Red Hat Fuse 7.10

Apache Karaf トランザクションガイド

ApacheKaraf コンテナーのトランザクションアプリケーション作成

Last Updated: 2023-07-29

Red Hat Fuse 7.10 Apache Karaf トランザクションガイド

ApacheKaraf コンテナーのトランザクションアプリケーション作成

法律上の通知

概要

Fuse のトランザクション対応アプリケーションの開発

目次

はじめに

本ガイドでは、Fuse トランザクションアプリケーションを実装するための情報および手順を提供します。本書では、以下の内容を取り上げます。

- 1章 *トランザクションの概要*
- 2章 *Karaf (OSGi) でのトランザクションの使用開始*
- 3章 *トランザクションマネージャーの設定および参照を行うインターフェイス*
- 4章 *Narayana トランザクションマネージャーの設定*
- 5章 *Narayana トランザクションマネージャーの使用*
- 6章 *JDBC データソースの使用*
- 7章 *JMS 接続ファクトリーの作成*
- 8章 *Java Connector Architecture (JCA) の概要*
- 9章 *トランザクションを使用する Camel アプリケーションの作成*

多様性を受け入れるオープンソースの強化

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

第1章 トランザクションの概要

この章では、いくつかの基本的なトランザクションの概念と、トランザクションマネージャーで重要なサービス品質について説明し、トランザクションを紹介します。本書では、以下の内容を取り上げます。

- 「トランザクションとは」
- 「トランザクションの ACID プロパティー」
- 「トランザクションクライアント」
- 「トランザクション用語の説明」
- 「複数のリソースを変更するトランザクションの管理」
- 「トランザクションとスレッドとの関係」
- 「トランザクションサービスの特性」

1.1. トランザクションとは

トランザクションのプロトタイプは、概念的には単一のステップ (例: アカウント A からアカウント B への送金) で設定される操作ですが、一連のステップとして実装する必要があります。このような操作は、障害によって一部のステップが未完了のままになり、システムが一貫性のない状態になる可能性があるので、システム障害に対して脆弱です。たとえば、口座 A から口座 B に送金する操作を考えてみます。口座 A (借方) に記入した後、口座 B (貸方) に記入する前に、システムに障害が発生したとします。その結果、一部のお金が失われます。

このような操作の信頼性を確保するために、この操作を **トランザクション** として実装します。トランザクションは、アトミックで一貫性があって分離されており、耐久性があるため、信頼性の高い実行が保証されます。これらのプロパティーは、トランザクションの ACID プロパティーと呼ばれます。

1.2. トランザクションの ACID プロパティー

トランザクションの **ACID** プロパティーは、次のように定義されています。

- **アトミック**: トランザクションは、all or nothing の手法です。トランザクションが完了すると、個々の更新が組み立てられ、同時にコミットまたは中止 (ロールバック) されます。
- **一貫性**: トランザクションは、システムをある一貫性のある状態から別の一貫性のある状態に移行する作業の単位です。
- **分離**: トランザクションの実行中、その部分的な結果は他のエンティティーには表示されません。
- **耐性**: トランザクションがコミットされた直後にシステムが失敗しても、トランザクションの結果は永続化されます。

1.3. トランザクションクライアント

トランザクションクライアント は、トランザクションを開始および終了できるようにする API またはオブジェクトです。通常、トランザクションクライアントはトランザクションの **開始**、**コミット**、または **ロールバック** などの操作を公開します。

標準の JavaEE アプリケーションでは、**javax.transaction.UserTransaction** インターフェイスはトランザクションクライアント API を公開します。Spring Framework である Spring Boot のコンテキストでは、**org.springframework.transaction.PlatformTransactionManager** インターフェイスが、トランザクションクライアント API を公開しています。

1.4. トランザクション用語の説明

以下の表は、重要なトランザクション用語を示しています。

用語	説明
境界	トランザクションの境界とは、トランザクションの開始および終了を指します。トランザクションの終了とは、トランザクションで実行された作業がコミットまたはロールバックされることを意味します。たとえば、境界は、トランザクションクライアント API を呼び出して明示的にすることも、トランザクションエンドポイントからメッセージがポーリングされるときはいつでも暗黙的にすることもできます。詳細は、*9章 トランザクションを使用する Camel アプリケーションの作成*を参照してください。
Resources	**リソース** は、永続化または完全な変更を行うことができるコンピューターシステムのコンポーネントです。実際には、リソースはほぼ常にデータベース上で階層化されたデータベースまたはサービスになります (例: 永続性のあるメッセージサービス)。ただし、他の種類のリソースも考えられます。たとえば、現金自動預払機 (ATM) はリソースの種類です。顧客がマシンから現金を物理的に受け取ると、トランザクションを元に戻すことはできません。
トランザクションマネージャー	**トランザクションマネージャー** は、1つ以上のリソースの間のトランザクションを調整するロールを果たします。多くの場合、トランザクションマネージャーはリソースに組み込まれています。たとえば、エンタープライズレベルのデータベースには通常、そのデータベースでコンテンツ変更が可能なトランザクションを管理できるトランザクションマネージャーが含まれます。**複数の** リソースを使用するトランザクションには、通常 **外部** トランザクションマネージャーが必要です。
トランザクションコンテキスト	**トランザクションコンテキスト** は、トランザクションの追跡に必要な情報をカプセル化するオブジェクトです。トランザクションコンテキストの形式は、関連するトランザクションマネージャーの実装によって異なります。少なくとも、トランザクションコンテキストには一意のトランザクション識別子が含まれます。
分散トランザクション	分散トランザクションは、トランザクションスコープが複数のネットワークノードにまたがる分散システムのトランザクションを指します。分散トランザクションをサポートするには基本的に、トランザクションコンテキストの送信をサポートするネットワークプロトコルが必須です。**分散トランザクションは Apache Camel トランザクションの対象範囲外です。**「分散トランザクションマネージャー」 も参照してください。

用語	説明
X/Open XA 標準	X/Open XA 標準は、リソースをトランザクションマネージャーと統合するためのインターフェイスを記述します。複数のリソースが含まれるトランザクションを管理するには、参加するリソースが XA 標準をサポートする必要があります。XA 標準をサポートするリソースは、**XA スイッチ** という特別なオブジェクトを公開します。このオブジェクトで、トランザクションマネージャー (またはトランザクション処理モニター) がリソースのトランザクションを制御できるようにします。XA 標準は、1 相コミットプロトコルと 2 相コミットプロトコルの両方をサポートします。

1.5. 複数のリソースを変更するトランザクションの管理

1つ のリソースを使用するトランザクションでは、通常、リソースに含まれるトランザクションマネージャーを使用できます。**複数** のリソースを使用するトランザクションには、外部トランザクションマネージャーまたはトランザクション処理 (TP) モニターを使用する必要があります。この場合、リソースは XA スイッチを登録してトランザクションマネージャーと統合する必要があります。

単一のリソースシステム上で動作するトランザクションと、複数のリソースシステムで動作するトランザクションのコミットに使用するトランザクションには、重要な違いがあります。

- **1相コミット**: シングルリソースシステム用このプロトコルは、1 つのステップでトランザクションをコミットします。
- **2相コミット**: 複数リソースシステム用このプロトコルは 2 つのステップでトランザクションをコミットします。

トランザクションに複数のリソースを含めると、リソース全部ではなく、一部のトランザクションをコミットすることが原因でシステム障害が発生するリスクが増加します。これにより、システムが一貫性のない状態のままになります。2 相コミットプロトコルは、このリスクを排除するために設計されています。これにより、システムを再起動すると、システムが **常に** 一貫した状態で復元できます。

1.6. トランザクションとスレッドとの関係

トランザクション処理を理解するには、トランザクションとスレッドの基本関係を把握することが重要です。**トランザクションはスレッド別になります。**つまり、トランザクションが開始されると、特定のスレッドに割り当てられます。(技術的には、**トランザクションコンテキスト** オブジェクトが作成され、現在のスレッドに関連付けられます)。スレッドのすべてのアクティビティーは、この時点からトランザクションが終了するまで、このトランザクションスコープ内で発生します。他のスレッドのアクティビティーが、このトランザクションのスコープ内に入ることは **ありません**。ただし、他のスレッドのアクティビティーが、他のトランザクションのスコープ内に入る可能性はあります。

トランザクションとスレッドのこの関係は、以下のようになります。

- 各トランザクションが別々のスレッドで作成されている限り、**アプリケーションは複数のトランザクションを同時に処理できます**。
- **トランザクション内でサブスレッドを作成すること** に注意してください。トランザクションの途中で、たとえば **threads()** Camel DSL コマンドを呼び出して、新しいスレッドプールを作成する場合、新しいスレッドは元のトランザクションのスコープに含まれ**ません**。

- 前述のポイントで示した理由と同じで、**新しいスレッドを暗黙的に作成する処理ステップに注意してください。**
- **トランザクションスコープは通常、ルートセグメント間では継承されません。**つまり、1つのルートセグメントが **JoinEndpoint** で終わり、別のルートセグメントが **JoinEndpoint** で始まる場合、これらのルートセグメントは通常同じトランザクションに属し **ません。**ただし、例外があります。

注記

一部の高度なトランザクションマネージャーの実装では、自由にトランザクションコンテキストをスレッドに対してデタッチしたり、アタッチしたりできます。たとえば、これにより、トランザクションコンテキストを別のスレッドに移動できます。場合によっては、1つのトランザクションコンテキストを複数のスレッドに割り当てることもできます。

1.7. トランザクションサービスの特性

トランザクションシステムを実装する製品を選択する場合に、さまざまなデータベース製品やトランザクションマネージャーが無料または商用で利用できます。これらはすべて、トランザクション処理のサポートが通常通りありますが、これらの製品がサポートするサービスの特性は各種あります。このセクションでは、さまざまなトランザクション製品の信頼性と洗練度を比較するときに考慮する必要のある機能の種類について簡単に説明します。

1.7.1. リソースが提供するサービスの特性

以下の機能は、リソースの QoS(Quality of Service) を決定します。

- 「トランザクション分離レベル」
- 「XA 標準のサポート」

1.7.1.1. トランザクション分離レベル

ANSI SQL は、以下のように 4 つの **トランザクション分離レベル** を定義します。

SERIALIZABLE

トランザクションは相互に完全に分離されます。つまり、1つのトランザクションはトランザクションがコミットされるまで他のトランザクションに影響を与えることはありません。この分離レベルは、すべてのトランザクションが次々に実行されたかのように影響を与えるので、**serializable** として記述されます。(ただし、実際には、リソースはアルゴリズムを最適化できることが多く、一部のトランザクションを同時に続行できます)。

REPEATABLE_READ

トランザクションがデータベースを読み取りまたは更新するたびに、読み取りまたは書き込みロックが取得され、トランザクションが終了するまで保持されます。これにより、ほぼ完全な分離が可能になります。ただし、分離が十分ではないというケースが1つあります。**WHERE** 句を使用して行の範囲を読み取る SQL の **SELECT** ステートメントについて考えてみましょう。最初のトランザクションの実行中に別のトランザクションがこの範囲に行を追加する場合、最初のトランザクションは **SELECT** の呼び出しを繰り返すと (**ファントム読み取り**)、この新しい行を確認できます。

READ_COMMITTED

書き込みロックは、トランザクションが終了するまで保持されます。読み取りロックは、トランザクションが終了するまで**保持されません。**その結果、他のトランザクションによってコミットされた更新が進行中のトランザクションに表示されるようになるため、読み取りを繰り返すと異なる結

果が得られる可能性があります。

READ_UNCOMMITTED

トランザクションが終了するまで、読み取りロックも書き込みロックも保持されません。そのため、ダーティリードが可能です。ダーティレディとは、他のトランザクションによって行われた変更の内、コミットされていない変更が進行中のトランザクションに表示される場合です。

データベースは通常、さまざまなトランザクション分離レベルのすべてをサポートしているわけではありません。たとえば、一部の無料データベースは **READ_UNCOMMITTED** のみをサポートします。また、一部のデータベースは、ANSI 規格とは微妙に異なる方法でトランザクション分離レベルを実装しています。分離は、データベースのパフォーマンスが犠牲にある複雑な問題です (たとえば、Wikipedia の分離 を参照してください)。

1.7.1.2. XA 標準のサポート

複数のリソースを含むトランザクションに参加するには、X/Open XA 標準をサポートする必要があります。XA 標準のリソースの実装が特別な制限を受けているかどうかを必ず確認してください。たとえば、XA 標準の一部の実装は、単一のデータベース接続に制限されており、一度に 1 つのスレッドのみがそのリソースを含むトランザクションを処理できることを意味します。

1.7.2. トランザクションマネージャーが提供するサービスの特性

以下の機能は、トランザクションマネージャーの QoS (Quality of Service) を決定します。

- 「一時停止/再開およびアタッチ/デタッチのサポート」.
- 「複数リソースのサポート」.
- 「分散トランザクション」.
- 「トランザクションの監視」.
- 「障害からの復旧」.

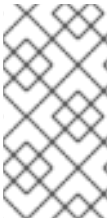
1.7.2.1. 一時停止/再開およびアタッチ/デタッチのサポート

トランザクションマネージャーの中には、以下のようにトランザクションコンテキストとアプリケーションスレッド間の関連付けを操作する高度な機能がサポートされます。

- **現在のトランザクションの一時停止/再開**: アプリケーションが現在のスレッドでトランザクション意外の作業を実行している間、現在のトランザクションコンテキストを一時的に停止できます。
- **トランザクションコンテキストのアタッチ/デタッチ**: トランザクションコンテキストをあるスレッドから別のスレッドに移動したり、トランザクションスコープを拡張して複数のスレッドを含めることができます。

1.7.2.2. 複数リソースのサポート

トランザクションマネージャーの主な違いは、複数のリソースをサポートする機能です。これには通常、XA 標準のサポートが必要であり、トランザクションマネージャーはリソースが XA スイッチを登録する方法を提供します。



注記

厳密に言えば、XA 標準は、複数リソースのサポートに使用できる唯一のアプローチではありませんが、最も実用的なアプローチです。代わり方法として、通常 XA スイッチによって提供されるアルゴリズムを実装するために、面倒な (および重要な) カスタムコードを記述する必要があります。

1.7.2.3. 分散トランザクション

一部のトランザクションマネージャーには、分散システム内の複数のノードがスコープに含まれるトランザクションを管理する機能があります。トランザクションコンテキストは、WS-AtomicTransactions または CORBA OTS などの特別なプロトコルを使用してノードからノードに伝播されます。

1.7.2.4. トランザクションの監視

通常、高度なトランザクションマネージャーは、保留中のトランザクションの状態を監視する視覚的なツールを提供します。この種のツールは、システム障害後に特に便利です。システム障害は、不確実な状態 (ヒューリスティックな例外) のままになっているトランザクションを特定して解決するのに役立ちます。

1.7.2.5. 障害からの復旧

トランザクションマネージャーはそれぞれ、システム障害 (クラッシュ) が発生した場合の堅牢性に大きな違いがあります。トランザクションマネージャーが使用する主なストラテジーは、トランザクションの各ステップを実行する前にデータを永続ログに書き込むことです。失敗した場合、ログのデータを使用してトランザクションを復元できます。一部のトランザクションマネージャーは、他のマネージャーよりも慎重にこのストラテジーを実装します。たとえば、ハイエンドトランザクションマネージャーは通常、永続的なトランザクションログを複製し、各ログを別のホストマシンに保存できるようにします。

第2章 KARAF (OSGI) でのトランザクションの使用開始

ここでは、トランザクションを使用して Artemis JMS ブローカーにアクセスする Camel アプリケーションについて説明します。本書では、以下の内容を取り上げます。

- 「前提条件」
- 「camel-jms プロジェクトのビルド」
- 「camel-jms プロジェクトの説明」

2.1. 前提条件

この Camel アプリケーションの実装には、以下の前提条件があります。

- 外部の AMQ 7 JMS メッセージブローカーが稼働している必要があります。
  以下のサンプルコードは、スタンドアロン (Docker 以外) バージョンの **amq-broker-7.1.0-bin.zip** を実行します。実行により **amq7** インスタンスが作成され、実行されます。

```
$ pwd
/data/servers/amq-broker-7.1.0

$ bin/artemis create --user admin --password admin --require-login amq7
Creating ActiveMQ Artemis instance at: /data/servers/amq-broker-7.1.0/amq7

Auto tuning journal ...
done! Your system can make 27.78 writes per millisecond, your journal-buffer-timeout will be 36000

You can now start the broker by executing:

   "/data/servers/amq-broker-7.1.0/amq7/bin/artemis" run

Or you can run the broker in the background using:

   "/data/servers/amq-broker-7.1.0/amq7/bin/artemis-service" start

$ amq7/bin/artemis run

        __  __  ____    ____          _
      /\   |  \/  |/ __ \  |  _ \        | |
     /  \  | \  / | |  | | | |_) |_ __ ___ | | _____ _ __
    / /\ \ | |\/| | |  | | |  _ <| '__/ _ \| |/ / _ \ '__|
   / _____ \| |  | | |__| | | |_) | | | (_) |   <  __/ |
  /_/    \_\_|  |_|\____\_\ |____/|_|  \___/|_|\_\___|_|

 Red Hat JBoss AMQ 7.1.0.GA


018-05-02 16:37:19,294 INFO  [org.apache.activemq.artemis.integration.bootstrap] AMQ101000: Starting ActiveMQ Artemis Server
...
```

- クライアントライブラリーが必要です。Artemis ライブラリーは Maven Central または Red Hat リポジトリーで利用できます。たとえば、以下を行うことができます。

- **mvn:org.apache.activemq/artemis-core-client/2.4.0.amq-710008-redhat-1**
- **mvn:org.apache.activemq/artemis-jms-client/2.4.0.amq-710008-redhat-1**

または、Artemis/AMQ 7 クライアントライブラリーを Karaf 機能としてインストールできます。以下に例を示します。

- **karaf@root()> feature:install artemis-jms-client artemis-core-client**

- Karaf シェルコマンドまたは専用の Artemis サポートを提供するサポート機能の一部が必要です。

```
karaf@root()> feature:install jms pax-jms-artemis pax-jms-config
```

- 必要な Camel 機能は次のとおりです。

```
karaf@root()> feature:install camel-jms camel-blueprint
```

2.2. CAMEL-JMS プロジェクトのビルド

Fuse Software Downloads ページから **quickstarts** をダウンロードします。

zip ファイルの内容をローカルフォルダーに展開します (例: **quickstarts** という名前の新規フォルダー)。

次に、/ **camel** / **camel-jms** の例を OSGi バンドルとしてビルドしてインストールできます。このバンドルには、AMQ 7 JMS キューへメッセージを送信する Camel ルートの Blueprint XML 定義が含まれます。

以下の例では、**$FUSE_HOME** が展開した Fuse ディストリビューションの場所になります。このプロジェクトをビルドするには、以下を実行します。

1. Maven を呼び出してプロジェクトをビルドします。

```
$ cd quickstarts

$ mvn clean install -f camel/camel-jms/
```

2. **javax.jms.ConnectionFactory** サービスが OSGi ランタイムに公開されるように、JMS 接続ファクトリー設定を作成します。これを行うには、 **quickstarts/camel/camel-jms/src/main/resources/etc/org.ops4j.connectionfactory-amq7.cfg** を **$FUSE_HOME/etc** ディレクトリーにコピーします。この設定は、機能する接続ファクトリーを作成するために処理されます。以下に例を示します。

```
$ cp camel/camel-jms/src/main/resources/etc/org.ops4j.connectionfactory-amq7.cfg ../etc/
```

3. 公開されている接続ファクトリーを確認します。

```
karaf@root()> service:list javax.jms.ConnectionFactory
[javax.jms.ConnectionFactory]
----------------------------
 felix.fileinstall.filename = file:$FUSE_HOME/etc/org.ops4j.connectionfactory-amq7.cfg
 name = artemis
 osgi.jndi.service.name = artemis
```

```
 password = admin
 pax.jms.managed = true
 service.bundleid = 251
 service.factoryPid = org.ops4j.connectionfactory
 service.id = 436
 service.pid = org.ops4j.connectionfactory.d6207fcc-3fe6-4dc1-a0d8-0e76ba3b89bf
 service.scope = singleton
 type = artemis
 url = tcp://localhost:61616
 user = admin
Provided by :
 OPS4J Pax JMS Config (251)

karaf@root()> jms:info -u admin -p admin artemis
Property │ Value
─────────┼──────────────────────────────────────────────────
product  │ ActiveMQ
version  │ 2.4.0.amq-711002-redhat-1

karaf@root()> jms:queues -u admin -p admin artemis
JMS Queues
──────────────────────────────────────────────────────
df2501d1-aa52-4439-b9e4-c0840c568df1
DLQ
ExpiryQueue
```

4. バンドルをインストールします。

```
karaf@root()> install -s mvn:org.jboss.fuse.quickstarts/camel-jms/7.0.0.redhat-SNAPSHOT
Bundle ID: 256
```

5. これが機能していることを確認します。

```
karaf@root()> camel:context-list
 Context             Status            Total #       Failed #     Inflight #   Uptime
 -------             ------            -------       --------     ----------   ------
 jms-example-context   Started               0              0              0   2 minutes
karaf@root()> camel:route-list
 Context             Route             Status            Total #       Failed #     Inflight #   Uptime
 -------             -----             ------            -------       --------     ----------   ------
 jms-example-context   file-to-jms-route   Started               0              0              0   2 minutes
 jms-example-context   jms-cbr-route       Started               0              0              0   2 minutes
```

6. Camel ルートが起動すると、即座に **work/jms/input** ディレクトリーが Fuse インストールに表示されます。このクイックスタートの **src/main/data** ディレクトリーにあるファイルを新しく作成した **work/jms/input** ディレクトリーにコピーします。

7. しばらく待つと、**work/jms/output** ディレクトリー以下に同じファイルが国別に分類されます。

   - **work/jms/output/others** の **order1.xml**、**order2.xml** および **order4.xml**
   - **work/jms/output/us** の **order3.xml** および **order5.xml**
   - **work/jms/output/fr** の **order6.xml**

8. ビジネスロギングを確認するには、ログを参照してください。

```
2018-05-02 17:20:47,952 | INFO  | ile://work/jms/input | file-to-jms-route                | 58 -
org.apache.camel.camel-core - 2.21.0.fuse-000077 | Receiving order order1.xml
2018-05-02 17:20:48,052 | INFO  | umer[incomingOrders] | jms-cbr-route                    | 58 -
org.apache.camel.camel-core - 2.21.0.fuse-000077 | Sending order order1.xml to another
country
2018-05-02 17:20:48,053 | INFO  | umer[incomingOrders] | jms-cbr-route                    | 58 -
org.apache.camel.camel-core - 2.21.0.fuse-000077 | Done processing order1.xml
```

9. キューが動的に作成されたことを確認します。

```
karaf@root()> jms:queues -u admin -p admin artemis
JMS Queues
————————————————————————————————————————
DLQ
17767323-937f-4bad-a403-07cd63311f4e
ExpiryQueue
incomingOrders
```

10. Camel ルートの統計を確認します。

```
karaf@root()> camel:route-info jms-example-context file-to-jms-route
Camel Route file-to-jms-route
 Camel Context: jms-example-context
 State: Started
 State: Started


Statistics
 Exchanges Total: 1
 Exchanges Completed: 1
 Exchanges Failed: 0
 Exchanges Inflight: 0
 Min Processing Time: 67 ms
 Max Processing Time: 67 ms
 Mean Processing Time: 67 ms
 Total Processing Time: 67 ms
 Last Processing Time: 67 ms
 Delta Processing Time: 67 ms
 Start Statistics Date: 2018-05-02 17:14:17
 Reset Statistics Date: 2018-05-02 17:14:17
 First Exchange Date: 2018-05-02 17:20:48
 Last Exchange Date: 2018-05-02 17:20:48
```

2.3. CAMEL-JMS プロジェクトの説明

Camel ルートは以下のエンドポイント URI を使用します。

```
<route id="file-to-jms-route">
...
    <to uri="jms:queue:incomingOrders?transacted=true" />
</route>
```

```
<route id="jms-cbr-route">
  <from uri="jms:queue:incomingOrders?transacted=true" />
...
</route>
```

jms コンポーネントは、以下のスニペットを使用して設定されます。

```
<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
  <property name="connectionFactory">
    <reference interface="javax.jms.ConnectionFactory" />
  </property>
  <property name="transactionManager" ref="transactionManager"/>
</bean>
```

transactionManager の参照は次のとおりです。

```
<reference id="transactionManager"
interface="org.springframework.transaction.PlatformTransactionManager" />
```

ご覧のとおり、JMS 接続ファクトリーと **PlatformTransactionManager** の Spring インターフェイスの両方が参照のみです。Blueprint XML で **定義** する必要はありません。これらの **サービス** は、Fuse 自体で公開されます。

javax.jms.ConnectionFactory が **etc/org.ops4j.connectionfactory-amq7.cfg** を使用して作成されたことはすでに確認できました。

トランザクションマネージャーは、以下のようになります。

```
karaf@root()> service:list org.springframework.transaction.PlatformTransactionManager
[org.springframework.transaction.PlatformTransactionManager]
----------------------------------------------------------
 service.bundleid = 21
 service.id = 527
 service.scope = singleton
Provided by :
 Red Hat Fuse :: Fuse Modules :: Transaction (21)
Used by:
 Red Hat Fuse :: Quickstarts :: camel-jms (256)
```

実際のトランザクションマネージャーが登録されている他のインターフェイスを確認します。

```
karaf@root()> headers 21

Red Hat Fuse :: Fuse Modules :: Transaction (21)
------------------------------------------------
...
Bundle-Name = Red Hat Fuse :: Fuse Modules :: Transaction
Bundle-SymbolicName = fuse-pax-transx-tm-narayana
Bundle-Vendor = Red Hat
...

karaf@root()> bundle:services -p 21

Red Hat Fuse :: Fuse Modules :: Transaction (21) provides:
----------------------------------------------------------
```

```
objectClass = [org.osgi.service.cm.ManagedService]
service.bundleid = 21
service.id = 519
service.pid = org.ops4j.pax.transx.tm.narayana
service.scope = singleton
 ----
objectClass = [javax.transaction.TransactionManager]
provider = narayana
service.bundleid = 21
service.id = 520
service.scope = singleton
 ----
objectClass = [javax.transaction.TransactionSynchronizationRegistry]
provider = narayana
service.bundleid = 21
service.id = 523
service.scope = singleton
 ----
objectClass = [javax.transaction.UserTransaction]
provider = narayana
service.bundleid = 21
service.id = 524
service.scope = singleton
 ----
objectClass = [org.jboss.narayana.osgi.jta.ObjStoreBrowserService]
provider = narayana
service.bundleid = 21
service.id = 525
service.scope = singleton
 ----
objectClass = [org.ops4j.pax.transx.tm.TransactionManager]
provider = narayana
service.bundleid = 21
service.id = 526
service.scope = singleton
 ----
objectClass = [org.springframework.transaction.PlatformTransactionManager]
service.bundleid = 21
service.id = 527
service.scope = singleton
```

トランザクションマネージャーは、以下のインターフェイスから利用できます。

- **javax.transaction.TransactionManager**
- **javax.transaction.TransactionSynchronizationRegistry**
- **javax.transaction.UserTransaction**
- **org.jboss.narayana.osgi.jta.ObjStoreBrowserService**
- **org.ops4j.pax.transx.tm.TransactionManager**
- **org.springframework.transaction.PlatformTransactionManager**

これらは、必要なコンテキストから任意のものを使用できます。たとえば、 **camel-jms** では、**org.apache.camel.component.jms.JmsConfiguration.transactionManager** フィールドを初期化する必要があります。これが、この例で以下が使用される理由です。

```
<reference id="transactionManager"
interface="org.springframework.transaction.PlatformTransactionManager" />
```

以下などは使用されません。

```
<reference id="transactionManager" interface="javax.transaction.TransactionManager" />
```

第3章 トランザクションマネージャーの設定および参照を行うインターフェイス

allInfacl および Spring Boot は、Fuse でトランザクションマネージャーを設定し、デプロイされたアプリケーションでトランザクションマネージャーを使用するためのトランザクションクライアントのインターフェイスを提供します。管理タスクである設定と開発タスクである参照には明確な違いがあります。アプリケーション開発者は、アプリケーションが以前に設定したトランザクションマネージャーを参照するように設定する必要があります。

- 「トランザクションマネージャーの機能」
- 「ローカル、グローバル、および分散トランザクションマネージャー」
- 「JavaEE トランザクションクライアントの使用」
- 「Spring Boot トランザクションクライアントの使用」
- 「トランザクションクライアントとトランザクションマネージャー間の OSGi インターフェイス」

3.1. トランザクションマネージャーの機能

トランザクションマネージャーは、1 つ以上のリソース間でトランザクションを調整するアプリケーションの一部です。トランザクションマネージャーのロールは以下のとおりです。

- 境界: begin、commit メソッド、および rollback メソッドを使用してトランザクションを開始および終了します。
- トランザクションコンテキストの管理: トランザクションコンテキストには、トランザクションマネージャーがトランザクションを追跡するために必要な情報が含まれます。トランザクションマネージャーは、トランザクションコンテキストを作成し、それらを現在のスレッドに割り当てます。
- 複数のリソース間でのトランザクションの調整: エンタープライズレベルのトランザクションマネージャーは、通常複数のリソース間でトランザクションを調整する機能があります。この機能には、2 相コミットプロトコルとリソースが XA プロトコルを使用して登録して管理する必要があります。「XA 標準のサポート」を参照してください。
  これは高度な機能で、すべてのトランザクションマネージャーでサポートされるわけではありません。
- 障害からの回復: システムに障害が発生した場合や、アプリケーションに問題があった場合にトランザクションマネージャーは、リソースの整合性のない状態のままにならないようにします。場合によっては、システムを一貫した状態に復元するために、手動による介入が必要になる場合があります。

3.2. ローカル、グローバル、および分散トランザクションマネージャー

トランザクションマネージャーは、ローカル、グローバル、または分散できます。

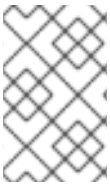
3.2.1. ローカルトランザクションマネージャー

ローカルトランザクションマネージャー は、1 つのリソースに対してのみトランザクションを調整できるトランザクションマネージャーです。ローカルトランザクションマネージャーの実装は通常、リソース自体に組み込まれ、アプリケーションが使用するトランザクションマネージャーは、この組み込みト

ランザクションマネージャーに関連するシンラッパーです。

たとえば、Oracle データベースには、デマケーション操作をサポートする組み込みトランザクションマネージャーがあり (SQL **BEGIN**、**COMMIT**、または **ROLLBACK** ステートメントを使用するか、ネイティブ Oracle API を使用)、さまざまなレベルのトランザクション分離があります。Oracle トランザクションマネージャーの制御は JDBC を介してエクスポートでき、この JDBC API は、アプリケーションがトランザクションの境界を定めるために使用します。

このコンテキストでは、リソース設定要素を理解することが重要です。たとえば、JMS 製品を使用している場合、JMS リソースは個別のキューやトピックではなく、JMS 製品の単一で実行中のインスタンスになります。さらに、同じ基になるリソースに異なる方法でアクセスする場合、複数のリソースのように見えるものが実際には単一のリソースである場合があります。たとえば、アプリケーションは、(JDBC を介して) 直接 (JDBC を介して) および間接的 (Hibernate などのオブジェクト関係マッピングツールを介して) リレーショナルデータベースにアクセスする可能性があります。この場合、基礎となるトランザクションマネージャーは同じため、これらのコードフラグメントの両方を同じトランザクションに登録できるはずです。



注記

すべてのケースで機能する保証はありません。原則では可能ですが、Spring Framework や他のラッパーレイヤーの設計の詳細により、実際には機能しなくなる可能性があります。

アプリケーションでは、多くの異なるローカルトランザクションマネージャーが互いに独立して動作させることができます。たとえば、JMS キューとトピックを操作する 1 つの Camel ルートがあり、JMS エンドポイントが JMS トランザクションマネージャーを参照しているとします。もう 1 つのルートは JDBC 経由でリレーショナルデータベースにアクセスできます。しかし、JDBC と JMS アクセスを同じルートで組み合わせて、両方を同じトランザクションに参加させることはできません。

3.2.2. グローバルトランザクションマネージャー

グローバルトランザクションマネージャーは、複数のリソースでトランザクションを調整できるトランザクションマネージャーです。これは、リソース自体に構築されたトランザクションマネージャーに依存できない場合に必要です。トランザクション処理モニター (TP モニター) と呼ばれる外部システムは、異なるリソース間のトランザクションを調整できます。

以下は、複数のリソースで動作するトランザクションの前提条件です。

- グローバルトランザクションマネージャーまたは TP モニター: 複数の XA リソースを調整するために 2 相コミットプロトコルを実装する外部トランザクションシステム。
- **XA 標準** をサポートするリソース: 2 相コミットに参加するには、リソースが XA 標準をサポートする必要があります。「XA 標準のサポート」を参照してください。実際には、これは、リソースが XA **スイッチ** オブジェクトをエクスポートできることを意味します。これにより、外部 TP モニターへのトランザクションを完全に制御できます。

ヒント

Spring Framework 自体は、グローバルトランザクションを管理する TP モニターを提供しているわけではありません。ただし、OSGi が提供する TP モニター、または (統合が JtaTransactionManager クラスによって実装される場所) との統合をサポートします。したがって、完全なトランザクションサポートを備えた OSGi コンテナーにアプリケーションをデプロイする場合、Spring で複数のトランザクションリソースを使用できます。

3.2.3. 分散トランザクションマネージャー

通常、サーバーはトランザクションに関連するリソースに直接接続します。ただし、分散システムでは、Web サービスを介して間接的にのみ公開されるリソースに接続する必要がある場合があります。この場合、分散トランザクションをサポートすることができる TP モニターが必要です。Web サービスの WS-AtomicTransactions 仕様など、さまざまな分散プロトコルのトランザクションをサポートする方法を記述するいくつかの標準が利用できます。

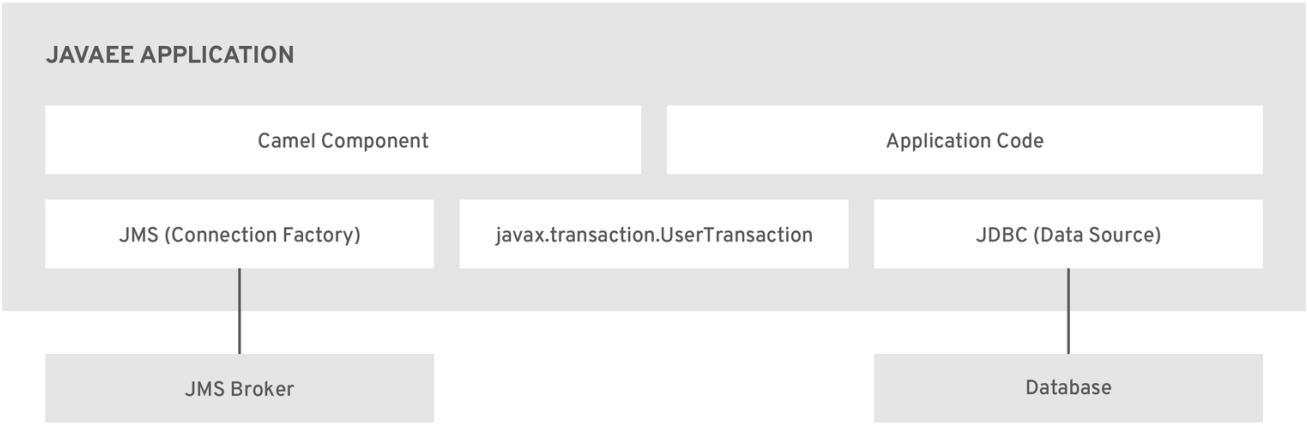
3.3. JAVAEE トランザクションクライアントの使用

JavaEE を使用する場合、トランザクションマネージャーと対話するための最も基本的で標準的な方法は、Java Transaction API (JTA) インターフェイス **javax.transaction.UserTransaction** です。正規の使用方法は以下のとおりです。

```
InitialContext context = new InitialContext();
UserTransaction ut = (UserTransaction) context.lookup("java:comp/UserTransaction");
ut.begin();

// Access transactional, JTA-aware resources such as database and/or message broker

ut.commit(); // or ut.rollback()
```

JNDI(Java Naming and Directory Interface) から **UserTransaction** インスタンスを取得する方法は、トランザクションクライアントを取得する方法の 1 つです。JavaEE 環境では、たとえば CDI (コンテキストおよび依存性注入) を使用してトランザクションクライアントにアクセスできます。

以下の図は、typica336 Camel アプリケーションを示しています。



以下の図は、Camel コードとアプリケーションコードの両方にアクセスできることを示しています。

- **javax.transaction.UserTransaction** インスタンスは、アプリケーションから直接トランザクションをデマケーションしたり、Spring の **TransactionTemplate** クラスを使用してトランザクション対応の Camel コンポーネントを介して内部でデマケーションできます。
- JDBC API 経由でのデータベースは直接的であるか、たとえば Spring の **JdbcTemplate** を使用するか、**camel-jdbc** コンポーネントを使用します。
- Spring の **JmsTemplate** クラスを使用するか、**camel-jms** コンポーネントを使用して直接 JMS API 経由のであるメッセージブローカー。

javax.transaction.UserTransaction オブジェクトを使用する場合、トランザクションクライアントのみを直接操作しているため、使用されている実際のトランザクションマネージャーを認識する必要はありません。(「トランザクションクライアント」 を参照してください。) Spring のトランザクション機能を内部で使用するため、Spring と Camel では異なるアプローチが使用されます。

JavaEE アプリケーション

一般的な JavaEE シナリオでは、アプリケーションは JavaEE アプリケーションサーバー (通常は **WAR** または **EAR** アーカイブ) にデプロイされます。JNDI または CDI により、アプリケーションは **javax.transaction.UserTransaction** サービスのインスタンスにアクセスできます。その後、このトランザクションクライアントインスタンスを使用してトランザクションを区別します。トランザクション内で、アプリケーションは JDBC または JMS アクセスを実行します。

Camel コンポーネントおよびアプリケーションコード

これらは、JMS/JDBC 操作を実行するコードを表します。Camel には、JMS/JDBC リソースにアクセスするための独自の高度なメソッドがあります。アプリケーションコードは、直接特定の API を使用できます。

JMS Connection Factory

これは、**javax.jms.Connection** のインスタンスの取得に使用される **javax.jms.ConnectionFactory** インターフェイス、および **javax.jms.Session** (または JMS 2.0 の **javax.jms.JmsContext**) です。これは、アプリケーションによって直接使用することも、**org.springframework.jms.core.JmsTemplate** で内部的に使用する場合もある Camel コンポーネントで間接的に使用することもできます。アプリケーションコードや Camel コンポーネントには、この接続ファクトリーの詳細は必要ありません。接続ファクトリーはアプリケーションサーバーで設定されます。この設定は、JavaEE サーバーで確認できます。Fuse などの OSGi サーバーも同様です。システム管理者は、アプリケーションとは別に接続ファクトリーを設定します。通常、接続ファクトリーはプーリング機能を実装します。

JDBC データソース

これは、**java.sql.Connection** のインスタンスを取得するために使用される **javax.sql.DataSource** インターフェイスです。JMS と同様に、このデータソースは直接的または間接的に使用できます。たとえば、**camel-sql** コンポーネントは **org.springframework.jdbc.core.JdbcTemplate** クラスを内部で使用します。JMS と同様に、アプリケーションコードや Camel にはこのデータソースの詳細は必要ありません。この設定は、*4章Narayana トランザクションマネージャーの設定* で説明されている方法を使用して、アプリケーションサーバー内または OSGi サーバー内で行われます。

3.4. SPRING BOOT トランザクションクライアントの使用

Spring Framework(および Spring Boot) の主な目的の 1 つとして、FIC API を簡単に使用できるようになります。すべての主要な JavaEE **vanilla** API は、Spring Framework(Spring Boot) に含まれています。これらは、特定の API の **代替** または **置換** ではなく、例外処理などで、より多くの設定オプションまたはより一貫した使用法を追加するラッパーです。

以下の表は、指定のブリック API を Spring 関連のインターフェイスと一致させます。

JavaEE API	Spring ユーティリティー	設定
JDBC	**org.springframework.jdbc.core.JdbcTemplate**	**javax.sql.DataSource**
JMS	**org.springframework.jms.core.JmsTemplate**	**javax.jms.ConnectionFactory**

JavaEE API	Spring ユーティリティー	設定
JTA	**org.springframework.transaction.support.TransactionTemplate**	**org.springframework.transaction.PlatformTransactionManager**

JdbcTemplate と **JmsTemplate** はそれぞれ **javax.sql.DataSource** と **javax.jms.ConnectionFactory** を直接使用します。しかし、**TransactionTemplate** は **PlatformTransactionManager** の Spring インターフェイスを使用します。これは、Spring が単に JavaEE を **改善する** のではなく、JavaEE クライアント API を独自のものに置き換える点です。

Spring は、**javax.transaction.UserTransaction** を実際のシナリオでは単純すぎるインターフェイスとして扱いま s. また、**javax.transaction.UserTransaction** はローカルの単一のリソーストランザクションとグローバルマルチリソーストランザクションを区別せず、**org.springframework.transaction.PlatformTransactionManager** の実装によって開発者はより自由になります。

以下は Spring Boot の正規 API の使用例です。

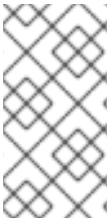
```
// Create or get from ApplicationContext or injected with @Inject/@Autowired.
JmsTemplate jms = new JmsTemplate(...);
JdbcTemplate jdbc = new JdbcTemplate(...);
TransactionTemplate tx = new TransactionTemplate(...);

tx.execute((status) -> {
   // Perform JMS operations within transaction.
   jms.execute((SessionCallback<Object>)(session) -> {
      // Perform operations on JMS session
      return ...;
   });
   // Perform JDBC operations within transaction.
   jdbc.execute((ConnectionCallback<Object>)(connection) -> {
      // Perform operations on JDBC connection.
      return ...;
   });
   return ...;
});
```

上記の例では、3 種類の **テンプレート** はすべて簡単にインスタンス化されますが、Spring の **ApplicationContext** からも取得したり、**@Autowired** アノテーションを使用してインジェクトしたりできます。

3.4.1. Spring PlatformTransactionManager インターフェイスの使用

前述のように、**javax.transaction.UserTransaction** は通常 JavaEE アプリケーションの JNDI から取得されます。ただし、Spring は多くのシナリオでこのインターフェイスの明示的な実装を提供します。常に完全な JTA シナリオが必要なく、アプリケーションは単一のリソース (JDBC など) のみにアクセスする必要がある場合があります。

通常、**org.springframework.transaction.PlatformTransactionManager** は **従来**のトランザクションクライアントの操作 (**begin**、**commit** および **rollback**) を提供する Spring トランザクションクライアント API です。つまり、このインターフェイスは実行時にトランザクションの制御に必要なメソッドを提供します。



注記

トランザクションシステムのもう 1 つの重要な要素は、トランザクションリソースを実装するための API です。しかし、トランザクションリソースは通常基盤のデータベースによって実装されるため、トランザクションプログラミングのこの要素はアプリケーションプログラマーにとって懸念されることはほとんどありません。

3.4.1.1. PlatformTransactionManager インターフェイスの定義

```
public interface PlatformTransactionManager {

    TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;

    void commit(TransactionStatus status) throws TransactionException;

    void rollback(TransactionStatus status) throws TransactionException;
}
```

3.4.1.2. TransactionDefinition インターフェイス

TransactionDefinition インターフェイスを使用して、新規に作成されたトランザクションの特性を指定します。新規トランザクションの分離レベルおよび伝播ポリシーを指定できます。詳細は、「トランザクション伝播ポリシー」 を参照してください。

3.4.1.3. TransactionStatus インターフェイスの定義

TransactionStatus インターフェイスを使用して、現在のトランザクション (つまり、現在のスレッドに関連するトランザクション) のステータスを確認し、現在のトランザクションをロールバック用にマーク付けできます。これはインターフェイス定義です。

```
public interface TransactionStatus extends SavepointManager, Flushable {

    boolean isNewTransaction();

    boolean hasSavepoint();

    void setRollbackOnly();

    boolean isRollbackOnly();

    void flush();

    boolean isCompleted();
}
```

3.4.1.4. PlatformTransactionManager インターフェイスで定義されるメソッド

PlatformTransactionManager インターフェイスは、以下のメソッドを定義します。

getTransaction()

新しいトランザクションを作成し、新しいトランザクションの特性を定義する **TransactionDefinition** オブジェクトを渡すことで、それを現在のスレッドに関連付けます。これは、数多くの他のトランザクションクライアント API の **begin()** メソッドと似ています。

commit()

現在のトランザクションをコミットし、登録されたリソースへの保留中の変更をすべて永続化します。

rollback()

現在のトランザクションをロールバックし、登録されたリソースに対して保留中の変更をすべて取り消します。

3.4.2. トランザクションマネージャーの使用手順

通常、**PlatformTransactionManager** インターフェイスを直接使用しません。Apache Camel では、通常以下のようにトランザクションマネージャーを使用します。

1. トランザクションマネージャーのインスタンスを作成します。Spring で利用できる実装がいくつかあります。「Spring Boot トランザクションクライアントの使用」を参照してください。

2. トランザクションマネージャーインスタンスを Apache Camel コンポーネントまたはルートの **transacted()** DSL コマンドに渡します。トランザクションコンポーネントまたは **transacted()** コマンドは、トランザクションを区切るロールを果たします。詳細は、*9章 トランザクションを使用する Camel アプリケーションの作成* を参照してください。

3.4.3. Spring PlatformTransactionManager 実装

本セクションでは、Spring Framework で提供されるトランザクションマネージャーの実装を概説します。この実装は、ローカルトランザクションマネージャーとグローバルトランザクションマネージャーの 2 つのカテゴリーに分類されます。

Camel から開始:

- **camel-jms** コンポーネントによって使用される **org.apache.camel.component.jms.JmsConfiguration** オブジェクトには、**org.springframework.transaction.PlatformTransactionManager** インターフェイスのインスタンスが必要です。

- **org.apache.camel.component.sql.SqlComponent** は **org.springframework.jdbc.core.JdbcTemplate** クラスを内部で使用し、この JDBC テンプレートは **org.springframework.transaction.PlatformTransactionManager** も統合します。

ご覧のとおり、このインターフェイスの実装が **いくつか** 必要です。シナリオによっては、必要なプラットフォームトランザクションマネージャーを設定できます。

3.4.3.1. ローカル PlatformTransactionManager 実装

以下の一覧は、Spring Framework で提供されるローカルトランザクションマネージャーの実装をまとめています。これらのトランザクションマネージャーは 1 つのリソースのみをサポートします。

org.springframework.jms.connection.JmsTransactionManager

このトランザクションマネージャーの実装は、**単一** の JMS リソースを管理できます。任意の数のキューまたはトピックに接続できますが、基礎となる JMS メッセージング製品インスタンスが同じものに所属する場合のみです。さらに、トランザクションの他のタイプのリソースを登録できません。

org.springframework.jdbc.datasource.DataSourceTransactionManager

このトランザクションマネージャーの実装では、**単一** の JDBC データベースリソースを管理できます。任意の数の異なるデータベーステーブルを更新することはできますが、それらは同じ基礎となるデータベースインスタンスに属する場合に **のみ** 更新できます。

org.springframework.orm.jpa.JpaTransactionManager

このトランザクションマネージャーの実装は、Java Persistence API (JPA) リソースを管理できます。ただし、トランザクション内の他の種類のリソースを同時に登録できません。

org.springframework.orm.hibernate5.HibernateTransactionManager

このトランザクションマネージャーの実装は、Hibernate リソースを管理できます。ただし、トランザクション内の他の種類のリソースを同時に登録できません。さらに、JPA API はネイティブの Hibernate API よりも優先されます。

また、他に使用頻度が低い **PlatformTransactionManager** の実装もあります。

3.4.3.2. グローバル PlatformTransactionManager 実装

Spring Framework は、OSGi ランタイムで使用するグローバルトランザクションマネージャーの実装を 1 つ提供します。**org.springframework.transaction.jta.JtaTransactionManager** は、トランザクションの複数のリソースに対する操作をサポートします。このトランザクションマネージャーは XA トランザクション API をサポートし、トランザクションに複数のリソースを登録できます。このトランザクションマネージャーを使用するには、OSGi コンテナーまたはブリックサーバーにアプリケーションをデプロイする必要があります。

PlatformTransactionManager の単一リソース実装は実際の**実装**ですが、**JtaTransactionManager** は標準の **javax.transaction.TransactionManager** の実際の実装に対するラッパーです。

PlatformTransactionManager の **JtaTransactionManager** 実装は、すでに設定済みの **javax.transaction.TransactionManager** インスタンスや通常 **javax.transaction.UserTransaction** に (JNDI または CDI で) アクセスできる環境で使用するほうが適切である理由です。通常、これらの JTA インターフェイスはどちらも単一のオブジェクト/サービスによって実装されます。

JtaTransactionManager の設定/使用例は次のとおりです。

```
InitialContext context = new InitialContext();
UserTransaction ut = (UserTransaction) context.lookup("java:comp/UserTransaction");
TransactionManager tm = (TransactionManager) context.lookup("java:/TransactionManager");

JtaTransactionManager jta = new JtaTransactionManager();
jta.setUserTransaction(ut);
jta.setTransactionManager(tm);

TransactionTemplate jtaTx = new TransactionTemplate(jta);

jtaTx.execute((status) -> {
    // Perform resource access in the context of global transaction.
    return ...;
});
```

上記の例では、JTA オブジェクトの実際のインスタンス (**UserTransaction** および **TransactionManager**) が JNDI から取得されます。OSGi では、OSGi サービスレジストリーからも取得することもできます。

3.5. トランザクションクライアントとトランザクションマネージャー間の OSGI インターフェイス

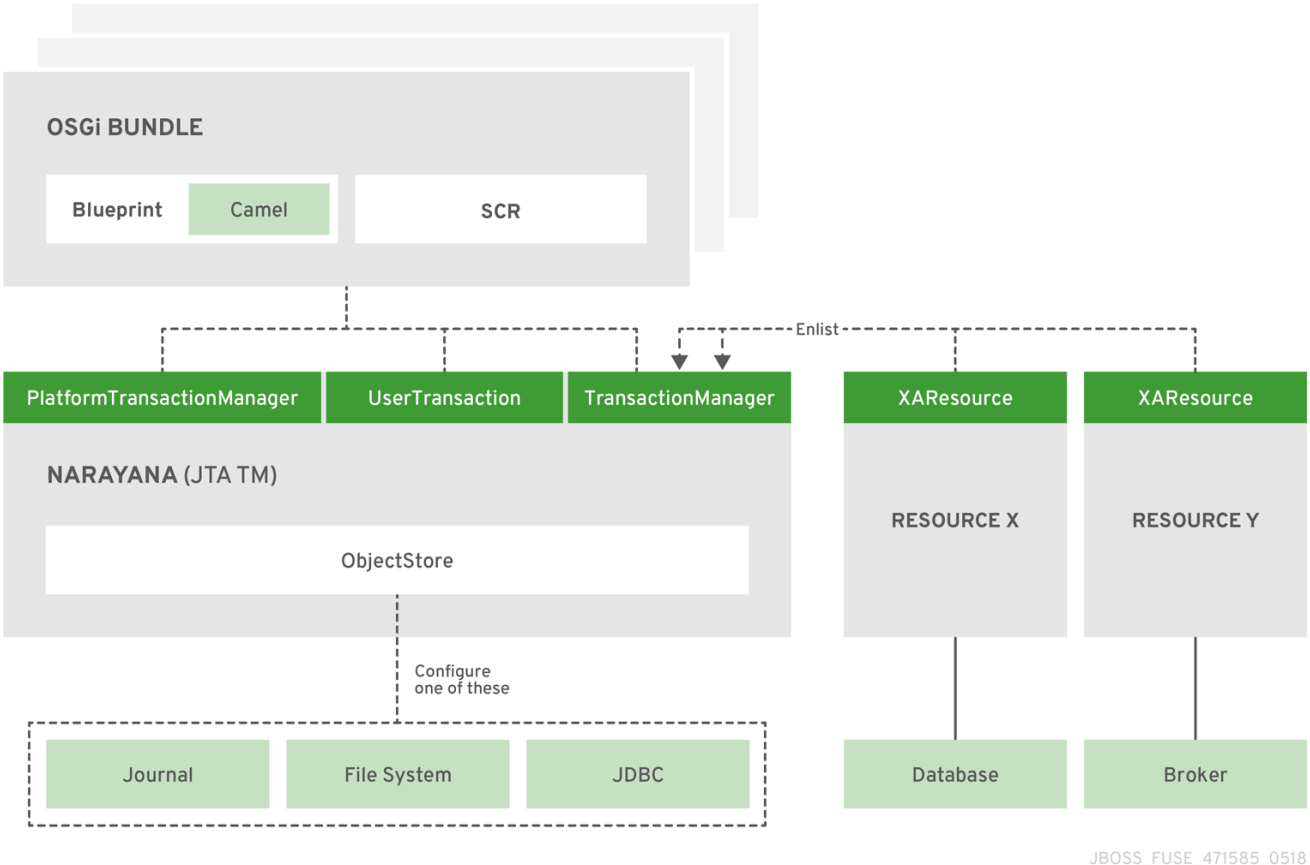
Warehouse トランザクションクライアント API と Spring Boot トランザクションクライアント API の記述後に、Fuse などの OSGi サーバー内の関係を確認すると便利です。OSGi の機能の 1 つは、グローバルサービスレジストリーで、以下を実行するために使用できます。

- フィルターまたはインターフェイスでサービスを検索します。
- 指定されたインターフェイスとプロパティーでサービスを登録します。

Java EE アプリケーションサーバーにデプロイされたアプリケーションが、JNDI (**サービスロケータ**メソッド) を使用して **javax.transaction.UserTransaction** への参照を取得したり、CDI (**依存性注入**メソッド) で注入したりするのと同じように、OSGi では、同じ参照 (直接的または間接的) を以下のいずれかの方法で取得できます。

- **org.osgi.framework.BundleContext.getServiceReference()** メソッド (**サービスロケーター**) を呼び出します。
- それらを Blueprint コンテナーに挿入します。
- Service Component Runtime(SCR) アノテーション (**依存性注入**) を使用します。

以下の図は、OSGi ランタイムにデプロイされた Fuse アプリケーションを示しています。アプリケーションコードや Camel コンポーネントは API を使用してトランザクションマネージャー、データソース、および接続ファクトリーへの参照を取得します。



アプリケーション (バンドル) は、OSGi レジストリーに登録されているサービスと対話します。**インターフェイス** を介してアクセスされ、これがアプリケーションに関連するすべてです。

Fuse では、(直接的または小さなラッパーを介して) トランザクションクライアントインターフェイスを実装する基本オブジェクトは **org.jboss.narayana.osgi.jta.internal.OsgiTransactionManager** です。以下のインターフェイスを使用してトランザクションマネージャーにアクセスできます。

- **javax.transaction.TransactionManager**
- **javax.transaction.UserTransaction**
- **org.springframework.transaction.PlatformTransactionManager**
- **org.ops4j.pax.transx.tm.TransactionManager**

これらのインターフェイスを直接使用するか、Camel などのフレームワークまたはライブラリーを選択して暗黙的に使用することができます。

Fuse で **org.jboss.narayana.osgi.jta.internal.OsgiTransactionManager** を設定する方法は、4 章*Narayana 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。このガイドの後半の章では、その章の情報に基づいて、JDBC データソースや JMS 接続ファクトリーなどの他のサービスを設定および使用する方法について説明します。

第4章 NARAYANA トランザクションマネージャーの設定

Fuse では、ビルトインのグローバルトランザクションマネージャーは JBoss Narayana Transaction Manager で、Enterprise Application Platform(EAP)7 によって使用されるトランザクションマネージャーと同じです。

Karaf の Fuse と同様に、OSGi ランタイムでは追加のインテグレーションレイヤーは PAX TRANSX プロジェクトによって提供されます。

以下のトピックでは、Narayana の設定について説明します。

- 「Narayana のインストール」
- 「サポートされているトランザクションプロトコル」
- 「Narayana 設定」
- 「ログストレージの設定」

4.1. NARAYANA のインストール

Narayana トランザクションマネージャーは、以下のインターフェイス、およびいくつかの追加のサポートインターフェイスの下で OSGi バンドルで使用できるように公開されています。

- **javax.transaction.TransactionManager**
- **javax.transaction.UserTransaction**
- **org.springframework.transaction.PlatformTransactionManager**
- **org.ops4j.pax.transx.tm.TransactionManager**

7.10.0.fuse-7_10_0-00010-redhat-00001 ディストリビューションでは、これらのインターフェイスを最初から利用できるようになります。

pax-transx-tm-narayana 機能には、Narayana を組み込むオーバーライドされたバンドルが含まれます。

```
karaf@root()> feature:info pax-transx-tm-narayana
Feature pax-transx-tm-narayana 0.3.0
Feature has no configuration
Feature has no configuration files
Feature depends on:
  pax-transx-tm-api 0.0.0
Feature contains followed bundles:
  mvn:org.jboss.fuse.modules/fuse-pax-transx-tm-narayana/7.0.0.fuse-000191-redhat-1 (overriden
from mvn:org.ops4j.pax.transx/pax-transx-tm-narayana/0.3.0)
Feature has no conditionals.
```

fuse-pax-transx-tm-narayana バンドルが提供するサービスは以下のとおりです。

```
karaf@root()> bundle:services fuse-pax-transx-tm-narayana

Red Hat Fuse :: Fuse Modules :: Transaction (21) provides:
----------------------------------------------------------
```

```
[org.osgi.service.cm.ManagedService]
[javax.transaction.TransactionManager]
[javax.transaction.TransactionSynchronizationRegistry]
[javax.transaction.UserTransaction]
[org.jboss.narayana.osgi.jta.ObjStoreBrowserService]
[org.ops4j.pax.transx.tm.TransactionManager]
[org.springframework.transaction.PlatformTransactionManager]
```

このバンドルは **org.osgi.service.cm.ManagedService** を登録するため、CM 設定の変更を追跡し、これに反応します。

```
karaf@root()> bundle:services -p fuse-pax-transx-tm-narayana

Red Hat Fuse :: Fuse Modules :: Transaction (21) provides:
----------------------------------------------------------
objectClass = [org.osgi.service.cm.ManagedService]
service.bundleid = 21
service.id = 232
service.pid = org.ops4j.pax.transx.tm.narayana
service.scope = singleton
...
```

デフォルトの **org.ops4j.pax.transx.tm.narayana** PID は以下のようになります。

```
karaf@root()> config:list '(service.pid=org.ops4j.pax.transx.tm.narayana)'
----------------------------------------------------------------
Pid:            org.ops4j.pax.transx.tm.narayana
BundleLocation: ?
Properties:
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.communicationStore.localOSRoot =
communicationStore
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.communicationStore.objectStoreDir =
/data/servers/7.10.0.fuse-7_10_0-00010-redhat-00001/data/narayana

com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.communicationStore.objectStoreType
= com.arjuna.ats.internal.arjuna.objectstore.ShadowNoFileLockStore
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.localOSRoot = defaultStore
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.objectStoreDir =
/data/servers/7.10.0.fuse-7_10_0-00010-redhat-00001/data/narayana
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.objectStoreType =
com.arjuna.ats.internal.arjuna.objectstore.ShadowNoFileLockStore
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.stateStore.localOSRoot = stateStore
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.stateStore.objectStoreDir =
/data/servers/7.10.0.fuse-7_10_0-00010-redhat-00001/data/narayana
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.stateStore.objectStoreType =
com.arjuna.ats.internal.arjuna.objectstore.ShadowNoFileLockStore
   com.arjuna.ats.arjuna.common.RecoveryEnvironmentBean.recoveryBackoffPeriod = 10
   felix.fileinstall.filename = file:/data/servers/7.10.0.fuse-7_10_0-00010-redhat-
00001/etc/org.ops4j.pax.transx.tm.narayana.cfg
   service.pid = org.ops4j.pax.transx.tm.narayana
```

つまり、以下のようになります。

- Karaf の Fuse には、フル機能のグローバル、Narayana トランザクションマネージャーが含まれます。

- トランザクションマネージャーは、さまざまなクライアントインターフェイス (JTA、Spring-tx、PAX JMS) で正しく公開されます。
- Narayana は、**org.ops4j.pax.transx.tm.narayana** で利用可能な標準の OSGi メソッドである Configuration Admin を使用して設定できます。
- デフォルト設定は **$FUSE_HOME/etc/org.ops4j.pax.transx.tm.narayana.cfg** で提供されます。

4.2. サポートされているトランザクションプロトコル

Narayana トランザクションマネージャー は、EAP で使用される JBoss/Red Hat 製品です。Narayana は、幅広い標準ベースのトランザクションプロトコルを使用して開発されたアプリケーションをサポートするトランザクションツールキットです。

- JTA
- JTS
- Web サービストランザクション
- REST トランザクション
- STM
- XATMI/TX

4.3. NARAYANA 設定

pax-transx-tm-narayana バンドルには **jbossts-properties.xml** ファイルが含まれており、これはトランザクションマネージャーのさまざまな側面のデフォルト設定を提供します。これらのプロパティーはすべて、**$FUSE_HOME/etc/org.ops4j.pax.transx.tm.narayana.cfg** ファイルに直接上書きしたり、Configuration Admin API を使用して上書きしたりできます。

Narayana の基本設定は、さまざまな **EnvironmentBean** オブジェクトによって行われます。このような Bean はすべて、接頭辞が異なるプロパティーを使用することで設定できます。以下の表は、使用される設定オブジェクトと接頭辞の概要を示しています。

設定 Bean	プロパティーの接頭辞
com.arjuna.ats.arjuna.common.CoordinatorEnvironmentBean	**com.arjuna.ats.arjuna.coordinator**
com.arjuna.ats.arjuna.common.CoreEnvironmentBean	**com.arjuna.ats.arjuna**
com.arjuna.ats.internal.arjuna.objectstore.hornetq.HornetqJournalEnvironmentBean	**com.arjuna.ats.arjuna.hornetqjournal**
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean	**com.arjuna.ats.arjuna.objectstore**
com.arjuna.ats.arjuna.common.RecoveryEnvironmentBean	**com.arjuna.ats.arjuna.recovery**

設定 Bean	プロパティーの接頭辞
com.arjuna.ats.jdbc.common.JDBCEnvironmentBean	**com.arjuna.ats.jdbc**
com.arjuna.ats.jta.common.JTAEnvironmentBean	**com.arjuna.ats.jta**
com.arjuna.ats.txoj.common.TxojEnvironmentBean	**com.arjuna.ats.txoj.lockstore**

接頭辞 をし用すると、設定を簡素化できます。ただし、通常、以下のいずれかの形式を使用する必要があります。

NameEnvironmentBean. propertyName (優先形式) または

fully-qualified-class-name.field-name

たとえば、**com.arjuna.ats.arjuna.common.CoordinatorEnvironmentBean.commitOnePhase** フィールドについて考えてみましょう。これは、**com.arjuna.ats.arjuna.common.CoordinatorEnvironmentBean.commitOnePhase** プロパティーを使用して設定するか、より簡単なフォーム **CoordinatorEnvironmentBean.commitOnePhase** (推奨) を使用して設定できます。プロパティーの設定方法と設定可能な Bean の詳細は、Narayana Product Documentation を参照してください。

ObjectStoreEnvironmentBean などの一部の Bean は、別の目的で設定を提供する **名前付き** インスタンスごとに複数回設定できます。この場合、インスタンスの名前は、接頭辞 (上記のいずれか) と **field-name** の間で使用されます。たとえば、**communicationStore** という名前の **ObjectStoreEnvironmentBean** インスタンスのオブジェクトストアのタイプは、名前付きプロパティーを使用して設定できます。

- **com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.communicationStore.objectStoreType**
- **ObjectStoreEnvironmentBean.communicationStore.objectStoreType**

4.4. ログストレージの設定

最も重要な設定は、オブジェクトログストレージのタイプおよび場所です。通常、**com.arjuna.ats.arjuna.objectstore.ObjectStoreAPI** インターフェイスには 3 つの実装があります。

com.arjuna.ats.internal.arjuna.objectstore.hornetq.HornetqObjectStoreAdaptor
: AMQ 7 の **org.apache.activemq.artemis.core.journal.Journal** ストレージを内部で使用します。

com.arjuna.ats.internal.arjuna.objectstore.jdbc.JDBCStore
: JDBC を使用して TX ログファイルを保持します。

com.arjuna.ats.internal.arjuna.objectstore.FileSystemStore **(および特別な実装)**
: カスタムファイルベースのログストレージを使用します。

デフォルトでは、Fuse は **com.arjuna.ats.internal.arjuna.objectstore.ShadowNoFileLockStore** を使用します。これは、**FileSystemStore** の特殊な実装です。

トランザクション/オブジェクトのログを保持する Narayana では、3 つの **ストア** が使用されます。

- **defaultStore**
- **communicationStore**
- **stateStore**

詳細は、Narayana ドキュメント の State management を参照してください。

これら 3 つの **ストア** のデフォルト設定は以下のとおりです。

```
# default store
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.objectStoreType =
com.arjuna.ats.internal.arjuna.objectstore.ShadowNoFileLockStore
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.objectStoreDir =
${karaf.data}/narayana
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.localOSRoot = defaultStore
# communication store
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.communicationStore.objectStoreType
= com.arjuna.ats.internal.arjuna.objectstore.ShadowNoFileLockStore
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.communicationStore.objectStoreDir =
${karaf.data}/narayana
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.communicationStore.localOSRoot =
communicationStore
# state store
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.stateStore.objectStoreType =
com.arjuna.ats.internal.arjuna.objectstore.ShadowNoFileLockStore
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.stateStore.objectStoreDir =
${karaf.data}/narayana
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.stateStore.localOSRoot = stateStore
```

ShadowNoFileLockStore は、ベースディレクトリー (**objectStoreDir**) および特定のストアディレクトリー (**localOSRoot**) で設定されます。

多くの設定オプションは、Narayana のドキュメントガイド に記載されています。ただし、Narayana ドキュメントには、設定オプションの正規参照はさまざまな **EnvironmentBean** クラスの Javadoc であると記載されています。

第5章 NARAYANA トランザクションマネージャーの使用

このセクションでは、**javax.transaction.UserTransaction** インターフェイス、**org.springframework.transaction.PlatformTransactionManager** インターフェイス、または **javax.transaction.Transaction** インターフェイスを実装して、Narayana トランザクションマネージャーを使用する詳細を説明します。使用するインターフェイスは、アプリケーションの要件によって異なります。本章の最後に、XA リソースを登録する問題の解決について扱います。本書では、以下の内容を取り上げます。

- 「UserTransaction オブジェクトの使用」
- 「TransactionManager オブジェクトの使用」
- 「トランザクションオブジェクトの使用」
- 「XA 登録の問題解決」

Java トランザクション API の詳細は、Java Transaction API(JTA)1.2 仕様と Javadoc を参照してください。

5.1. USERTRANSACTION オブジェクトの使用

トランザクションのマケーションの **javax.transaction.UserTransaction** インターフェイスを実装します。トランザクションを開始、コミット、またはロールバックする場合です。これは、アプリケーションコードで直接使用する可能性のある JTA インターフェイスです。ただし、**UserTransaction** インターフェイスはトランザクションを区切る方法の 1 つにすぎません。トランザクションの境界を定めるさまざまな方法の説明は、*9章 トランザクションを使用する Camel アプリケーションの作成* を参照してください。

5.1.1. UserTransaction インターフェイスの定義

JTA **UserTransaction** インターフェイスは以下のように定義されます。

```
public interface javax.transaction.UserTransaction {

    public void begin();

    public void commit();

    public void rollback();

    public void setRollbackOnly();

    public int getStatus();

    public void setTransactionTimeout(int seconds);
}
```

5.1.2. UserTransaction メソッドの説明

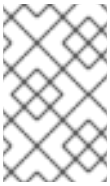
UserTransaction インターフェイスは以下のメソッドを定義します。

begin()

新しいトランザクションを開始し、これを現在のスレッドに関連付けます。XA リソースがこのトランザクションに関連する場合、トランザクションは暗黙的に XA トランザクションになります。

commit()

現在のトランザクションを正常に完了し、保留中のすべての変更が永続化されるようにします。コミット後、現在のスレッドに関連付けられているトランザクションはなくなります。



注記

ただし、現在のトランザクションがロールバックのみとしてマークされている場合は、**commit()** が呼び出されるとトランザクションは実際にはロールバックされます。

rollback()

トランザクションを直ちに中止し、保留中の変更を破棄します。ロールバック後、現在のスレッドに関連付けられているトランザクションはなくなります。

setRollbackOnly()

現在のトランザクションの状態を変更し、ロールバック以外に選択できる結果をなくしますが、ロールバックは実行しません。

getStatus()

現在のトランザクションのステータスを返します。これは、**javax.transaction.Status** インターフェイスで定義された以下の整数値のいずれかになります。

- **STATUS_ACTIVE**
- **STATUS_COMMITTED**
- **STATUS_COMMITTING**
- **STATUS_MARKED_ROLLBACK**
- **STATUS_NO_TRANSACTION**
- **STATUS_PREPARED**
- **STATUS_PREPARING**
- **STATUS_ROLLEDBACK**
- **STATUS_ROLLING_BACK**
- **STATUS_UNKNOWN**

setTransactionTimeout()

現在のトランザクションのタイムアウトを秒単位で指定してカスタマイズします。トランザクションが指定のタイムアウト内で解決されない場合、トランザクションマネージャーは自動的にこれをロールバックします。

5.2. TRANSACTIONMANAGER オブジェクトの使用

javax.transaction.TransactionManager オブジェクトを使用する最も一般的な方法は、フレームワーク API (例: Camel JMS コンポーネント) に渡すことです。これにより、フレームワークがトランザクションの境界を管理できるようになります。**TransactionManager** オブジェクトを直接使用する必要が

ある場合があります。これは、**suspend()** や **resume()** メソッドなどの高度なトランザクション API にアクセスする必要がある場合に役立ちます。

5.2.1. TransactionManager インターフェイスの定義

JTA TransactionManager インターフェイスの定義は以下のとおりです。

```
interface javax.transaction.TransactionManager {

    // Same as UserTransaction methods

    public void begin();

    public void commit();

    public void rollback();

    public void setRollbackOnly();

    public int getStatus();

    public void setTransactionTimeout(int seconds);

    // Extra TransactionManager methods

    public Transaction getTransaction();

    public Transaction suspend() ;

    public void resume(Transaction tobj);
}
```

5.2.2. TransactionManager メソッドの説明

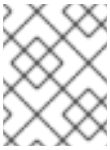
TransactionManager インターフェイスは、**UserTransaction** インターフェイスにあるすべてのメソッドをサポートします。トランザクションのデマケーションには **TransactionManager** オブジェクトを使用できます。さらに、**TransactionManager** オブジェクトは以下のメソッドをサポートします。

getTransaction()

現在のトランザクションへの参照を取得します。これは、現在のスレッドに関連付けられているトランザクションです。現在のトランザクションがない場合は、このメソッドは **null** を返します。

suspend()

現在のスレッドから現在のトランザクションをデタッチし、トランザクションへの参照を返します。このメソッドの呼び出し後、現在のスレッドにはトランザクションコンテキストがなくなります。この時点以降に行った作業は、トランザクションのコンテキストでは行われなくなります。



注記

すべてのトランザクションマネージャーがトランザクションの一時停止をサポートしているわけではありません。この機能は Narayana でサポートされています。

resume()

一時停止されたトランザクションを現在のスレッドコンテキストに再割り当てします。このメソッドを呼び出した後、トランザクションコンテキストが復元され、この時点以降に実行するすべての作業がトランザクションのコンテキストで実行されます。

5.3. トランザクションオブジェクトの使用

トランザクションを一時停止または再開する必要がある場合、またはリソースを明示的に登録する必要がある場合、**javax.transaction.Transaction** オブジェクトを直接使用する必要がある場合があります。で説明したように「XA 登録の問題解決」、フレームワークまたはコンテナーは通常、リソースの登録を自動的に処理します。

5.3.1. トランザクションインターフェイスの定義

JTA **Transaction** インターフェイスの定義は以下のとおりです。

```
interface javax.transaction.Transaction {

    public void commit();

    public void rollback();

    public void setRollbackOnly();

    public int getStatus();

    public boolean enlistResource(XAResource xaRes);

    public boolean delistResource(XAResource xaRes, int flag);

    public void registerSynchronization(Synchronization sync);
}
```

5.3.2. トランザクションメソッドの説明

commit()、**rollback()**、**setRollbackOnly()**、および **getStatus()** メソッドは、**UserTransaction** インターフェイスからの該当のメソッドと同じ動作をします。実際、**UserTransaction** オブジェクトは現在のトランザクションを取得し、**Transaction** オブジェクトで対応するメソッドを呼び出す便利なラッパーです。

また、**Transaction** インターフェイスは、**UserTransaction** インターフェイスにカウンターがない以下のメソッドを定義します。

enlistResource()

XA リソースを現在のトランザクションに関連付けます。

注記

この方法は、XA トランザクションのコンテキストで非常に重要です。XA トランザクションを特徴とする現在のトランザクションを使用して複数の XA リソースを登録する機能です。一方、リソースを明示的に登録するのは面倒であり、通常、フレームワークまたはコンテナーがこれを行うことを想定します。たとえば、「XA 登録の問題解決」 を参照してください。

delistResource()

トランザクションから、指定されたリソースの関連付けを解除します。flag 引数は、**javax.transaction.Transaction** インターフェイスで定義される以下の整数値のいずれかを取ることができます。

- **TMSUCCESS**
- **TMFAIL**
- **TMSUSPEND**

registerSynchronization()

javax.transaction.Synchronization オブジェクトを現在のトランザクションに登録します。**Synchronization** オブジェクトは、コミットの準備フェーズの直前にコールバックを受信し、トランザクションの完了直後にコールバックを受け取ります。

5.4. XA 登録の問題解決

XA リソースを登録する JTA の標準的な手順は、現在のトランザクションを表す現在の **javax.transaction.Transaction** オブジェクトに XA リソースを明示的に追加することです。つまり、新しいトランザクションが開始されるたびに XA リソースを明示的に登録する必要があります。

5.4.1. XA リソースの登録方法

XA リソースをトランザクションに登録するには、**Transaction** インターフェイスで **enlistResource()** メソッドを呼び出す必要があります。たとえば、**TransactionManager** オブジェクトおよび **XAResource** オブジェクトの場合は、以下のように **XAResource** オブジェクトを登録できます。

```
// Java
import javax.transaction.Transaction;
import javax.transaction.TransactionManager;
import javax.transaction.xa.XAResource;
...
// Given:
// 'tm' of type TransactionManager
// 'xaResource' of type XAResource

// Start the transaction
tm.begin();

Transaction transaction = tm.getTransaction();
transaction.enlistResource(xaResource);

// Do some work...
...

// End the transaction
tm.commit();
```

リソース登録が難しい点として、リソースを新しいトランザクション **ごと** に登録する必要があるので、リソースの使用を開始する前にリソースを登録する必要があります。リソースを明示的に登録すると、エラーが発生しやすいコードに **enlistResource()** 呼び出しがいくつもある状態になる可能性があります。さらに、適切な場所で **enlistResource()** を呼び出すことが困難なことがあります。たとえば、トランザクションの詳細の一部を非表示にするフレームワークを使用している場合などです。

5.4.2. 自動登録

XA リソースを明示的に登録する代わりに、XA リソースの自動登録をサポートする機能を使用するほうが簡単かつ安全です。たとえば、JMS および JDBC リソースを使用する場合に、標準の手法として自動登録に対応するラッパークラスを使用することが挙げられます。

JDBC および JMS アクセスの両方で一般的なパターンは次のとおりです。

1. アプリケーションコードは、JDBC アクセスには **javax.sql.DataSource** を、JMS での JDBC または JMS 接続には **javax.jms.ConnectionFactory** が必要です。

2. アプリケーション/OSGi サーバー内では、これらのインターフェイスのデータベースまたはブローカー固有の実装が登録されます。

3. アプリケーション/OSGi サーバーは、データベース/ブローカー固有のファクトリーを汎用のプーリングおよび登録ファクトリーに **ラップ** します。

このようにして、アプリケーションコードは引き続き **javax.sql.DataSource** および **javax.jms.ConnectionFactory** を使用しますが、内部的にこれらのコードがアクセスされると、通常は以下に関する追加機能があります。

- **接続プール**: データベース/メッセージブローカーへの新しいコネクションを作成する代わりに、事前に初期化された接続の **プール** が使用されます。**プーリング** には、接続の定期的な検証など、もう 1 つの側面があります。

- **JTA 登録** - **java.sql.Connection** (JDBC) または **javax.jms.Connection** (JMS) のインスタンスを返す前に、真の XA リソースであれば、実際の接続オブジェクトが登録されます。利用可能な場合、登録は JTA トランザクション内で行われます。

自動登録では、アプリケーションコードを変更する必要はありません。

JDBC データソースおよび JMS 接続ファクトリーのラッパーのプールとエンリストに関する詳細は、6章*JDBC データソースの使用* および 7章*JMS 接続ファクトリーの作成* を参照してください。

第6章 JDBC データソースの使用

以下のトピックでは、Fuse OSGi ランタイムでの JDBC データソースの使用について説明します。

- 「接続インターフェイスについて」
- 「JDBC データソースの概要」
- 「JDBC データソースの設定」
- 「OSGi JDBC サービスの使用」
- 「JDBC コンソールコマンドの使用」
- 「暗号化された設定値の使用」
- 「JDBC 接続プールの使用」
- 「アーティファクトとしてのデータソースのデプロイ」
- 「Java™ Persistence API でのデータソースの使用」

6.1. 接続インターフェイスについて

データ操作の実行に使用する最も重要な **オブジェクト** は、**java.sql.Connection** インターフェイスの実装です。Fuse 設定の観点からは、**Connection** オブジェクトの **取得** 方法を理解することが重要です。

関連するオブジェクトを含むライブラリーは以下のとおりです。

- PostgreSQL: **mvn:org.postgresql/postgresql/42.2.5**
- MySQL: **mvn:mysql/mysql-connector-java/5.1.34**

既存の実装 (**ドライバー JAR**に含まれる) は以下を提供します。

- PostgreSQL: **org.postgresql.jdbc.PgConnection**
- MySQL: **com.mysql.jdbc.JDBC4Connection** (**com.mysql.jdbc.Driver** のさまざまな **connect*()** メソッドも参照)

これらの実装には、DML、DDL、および簡単なトランザクション管理を実行するためのデータベース固有のロジックが含まれます。

理論的には、これらの接続オブジェクトを手動で作成することは可能ですが、詳細を非表示にしてよりクリーンな API を提供する 2 つの JDBC メソッドがあります。

- **java.sql.Driver.connect()** - このメソッドは、ずっと前にスタンドアロンアプリケーションで使用されていました。
- **javax.sql.DataSource.getConnection()** - **ファクトリー**パターンを使用する場合に推奨される方法です。同様の方法を使用して、JMS 接続ファクトリーから JMS 接続を取得します。

ドライバーマネージャー のアプローチについては、ここでは説明しません。このメソッドは、特定の接続オブジェクトのプレーンコンストラクターの上の小さな **レイヤー** であると記述するだけで十分です。

データベース固有の通信プロトコルを効果的に実装する **java.sql.Connection** に加えて、次の 2 つの特殊な**接続**インターフェイスがあります。

- **javax.sql.PooledConnection** は物理接続を表します。コードは、このプールされた接続と直接対話しません。代わりに、**getConnection()** メソッドから取得した接続が使用されます。この間接参照により、アプリケーションサーバーのレベルで接続プールを管理できます。通常、**getConnection()** を使用して取得した接続はプロキシーです。このようなプロキシー接続が閉じられると、物理接続は切断されず、管理接続プールで再度利用できるようになります。

- **javax.sql.XAConnection** では、XA 対応の接続に関連する **javax.transaction.xa.XAResource** オブジェクトを取得して、**javax.transaction.TransactionManager** と使用できます。**javax.sql.XAConnection** は **javax.sql.PooledConnection** を継承するので、`getConnection() メソッドも提供し、通常の DML/DQL メソッドで JDBC 接続オブジェクトにアクセスできます。

6.2. JDBC データソースの概要

JDBC 1.4 標準では、**java.sql.Connection** オブジェクトの **ファクトリー** として機能する **javax.sql.DataSource** インターフェイスが導入されました。通常、このようなデータソースは JNDI レジストリーにバインドされ、サーブレットや EJB などの JavaEE コンポーネント内に配置または挿入されていました。重要な側面は、これらのデータソースが **アプリケーションサーバー** 内で **設定** され、デプロイされたアプリケーションで名前で **参照** されていることです。

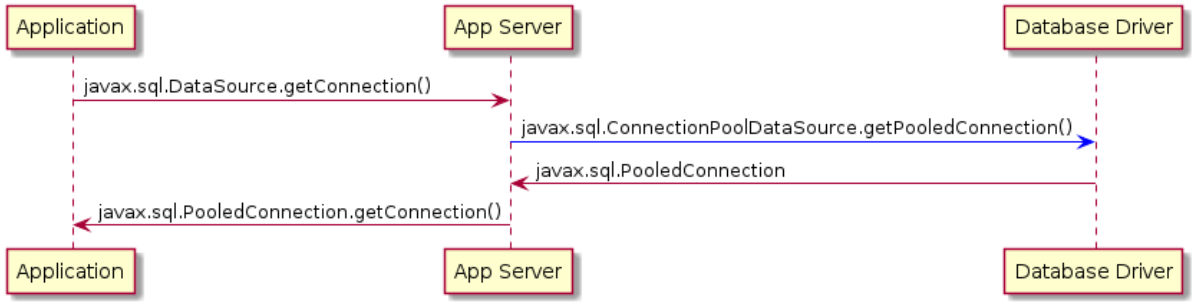
以下の **接続** オブジェクトには独自の **データソース** があります。

データソース	接続
javax.sql.DataSource	**java.sql.Connection**
javax.sql.ConnectionPoolDataSource	**javax.sql.PooledConnection**
javax.sql.XADataSource	**javax.sql.XAConnection**

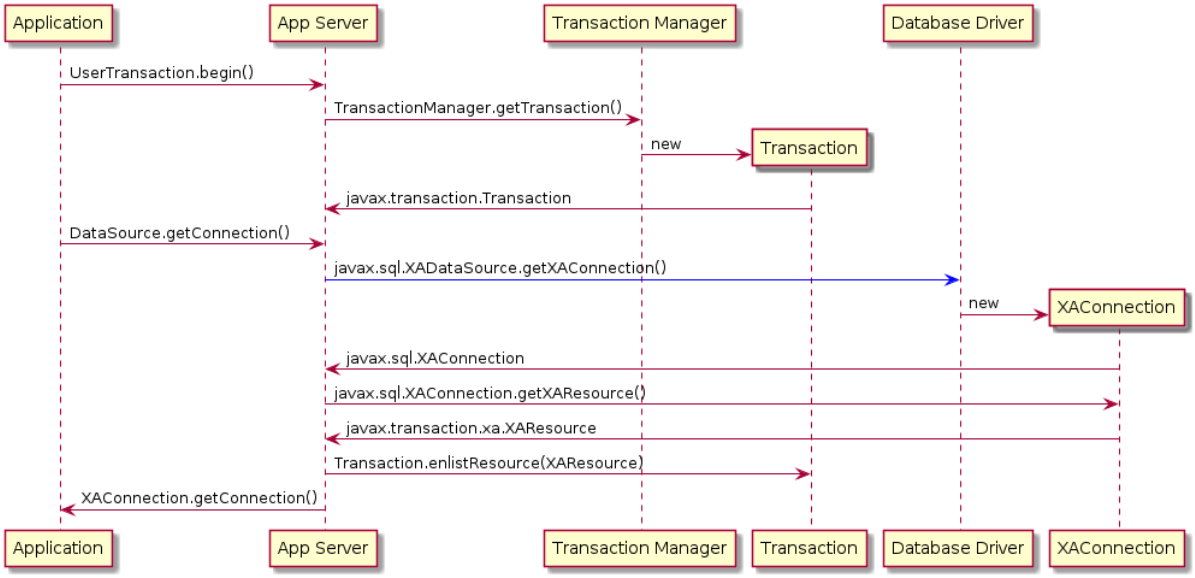
上記の各 **データソース** で最も重要な相違点は以下のとおりです。

- 最も重要なのは、**javax.sql.DataSource** は、**java.sql.Connection** インスタンスを取得するための **ファクトリー**のようなオブジェクトである点です。ほとんどの **javax.sql.DataSource** 実装が通常**接続プーリング**を実行することが事実であり、イメージを変更すべきではありません。これは、アプリケーションコードで使用する必要がある唯一のインターフェイスです。実装しているのはどれかは重要ではありません。
  - 直接 JDBC アクセス
  - JPA 永続ユニット設定 (**<jta-data-source> または<non-jta-data-source>**)
  - Apache Camel や Spring Framework などの一般的なライブラリー
- 最も重要なのは、**javax.sql.ConnectionPoolDataSource** は汎用 (データベース固有でない) 接続プール/データソースとデータベース固有のデータソースとの間の **ブリッジ** であることです。SPI インターフェイスとして処理できます。通常、アプリケーションコードは、JNDI から取得され、アプリケーションサーバーによって実装される汎用 **javax.sql.DataSource** オブジェクトに対応します (おそらく **commons-dbcp2** などのライブラリーを使用します)。一方、アプ

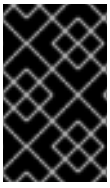
リケーションコードは **javax.sql.ConnectionPoolDataSource** と直接インターフェイスしません。これは、アプリケーションサーバーとデータベース固有のドライバーの間で使用されます。以下のシーケンス図は、これを示しています。



- **javax.sql.XADataSource** は、**javax.sql.XAConnection** と **javax.transaction.xa.XAResource** を取得する方法です。**javax.sql.ConnectionPoolDataSource** と同じで、これはアプリケーションサーバーとデータベース固有のドライバーとの間で使用されます。これは、今回は JTA トランザクションマネージャーなど、異なるアクターが含まれる図を若干変更しています。



上記の 2 つの図に示すように、**アプリケーションサーバー**と対話できます。これは一般的なエンティティーで、 **javax.sql.DataSource** および **javax.transaction.UserTransaction** インスタンスを設定できます。このようなインスタンスには、JNDI を使用するか、CDI や他の依存関係メカニズムを使用して注入することでアクセスできます。



重要

重要な点は、アプリケーションが XA トランザクションや接続プーリングを使用する場合でも、アプリケーションは他の 2 つの JDBC データソースインターフェイスではなく、**javax.sql.DataSource** と対話することです。

6.2.1. データベース固有のデータソースおよび汎用データソース

JDBC データソースの実装は 2 つのカテゴリーに分類されます。

- 以下のような汎用の **javax.sql.DataSource** 実装:
  - Apache Commons DBCP(2)
  - Apache Tomcat JDBC(DBCP ベース)

- **javax.sql.DataSource**、**javax.sql.XADataSource**、および **javax.sql.ConnectionPoolDataSource** のデータベース固有の実装

汎用 javax.sql.DataSource 実装によって独自のデータベース固有の接続を作成できないことが紛らわしい可能性があります。**汎用** データソースが **java.sql.Driver.connect()** または **java.sql.DriverManager.getConnection()** を使用できたとしても、データベース固有の **javax.sql.DataSource** 実装でこの **汎用** データソースを設定するのが通常、適切またはクリーンです。

汎用 データソースが JTA と対話する場合は、 **javax.sql.XADataSource** のデータベース固有の実装で設定する **必要** があります。

イメージを閉じるには、通常、**汎用** データソースは、接続プーリングを実行するために **javax.sql.ConnectionPoolDataSource** のデータベース固有の実装を必要**としません**。既存のプールは通常、標準の JDBC インターフェイス (**javax.sql.ConnectionPoolDataSource** および **javax.sql.PooledConnection**) なしでプーリングを処理し、代わりに独自のカスタム実装を使用します。

6.2.2. 一部の汎用データソース

サンプルのよく知られた汎用データソースである ApacheCommonsDBCP(2) を見ていきます。

javax.sql.XADataSource 実装

DBCP2 には、**javax.sql.XADataSource** の実装は含まれていないことが予想されます。

javax.sql.ConnectionPoolDataSource implementations

DBCP2 には **javax.sql.ConnectionPoolDataSource**: **org.apache.commons.dbcp2.cpdsadapter.DriverAdapterCPDS** の実装が**含まれます**。**java.sql.DriverManager.getConnection()** を呼び出して **javax.sql.PooledConnection** オブジェクトを作成します。このプールを直接使用することはできず、以下のドライバーの **アダプター** として扱う必要があります。

- 独自の **javax.sql.ConnectionPoolDataSource** 実装は指定しないでください。
- 接続プールに関する JDBC の**推奨事項** に応じて使用する必要があります。

上記のシーケンス図に示すように、ドライバーは **javax.sql.ConnectionPoolDataSource** を直接提供するか、**org.apache.commons.dbcp2.cpdsadapter.DriverAdapterCPDS アダプター** の助けを借りて提供します。一方、DBCP2 は、以下のいずれかで **アプリケーションサーバー** コントラクトを以下のいずれかで実装します。

- **org.apache.commons.dbcp2.datasources.PerUserPoolDataSource**
- **org.apache.commons.dbcp2.datasources.SharedPoolDataSource**

これらのプールは、設定段階で **javax.sql.ConnectionPoolDataSource** のインスタンスを取ります。

これは DBCP2 の最も重要で興味深い部分です。

javax.sql.DataSource implementations

接続プール機能を実装するには、JDBC の**推奨事項** に従って、**javax.sql.ConnectionPoolDataSource → javax.sql.PooledConnection** SPI を使用する必要はありません。

以下は、DBCP2 の **通常** のデータソースの一覧です。

- **org.apache.commons.dbcp2.BasicDataSource**

- **org.apache.commons.dbcp2.managed.BasicManagedDataSource**
- **org.apache.commons.dbcp2.PoolingDataSource**
- **org.apache.commons.dbcp2.managed.ManagedDataSource**

2 つの **軸** があります。

基本 vs プーリング

この**軸**は、**プーリング設定**の要素を決定します。

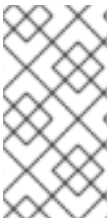
いずれの種類のデータソースも、**java.sql.Connection** オブジェクトの **プーリング** を実行します。**唯一**の違いは、以下の点です。

- **基本的な** データソースは、**org.apache.commons.pool2.impl.GenericObjectPool** の内部インスタンスを設定するために使用される **maxTotal** または **minIdle** などの Bean プロパティーを使用して設定されます。
- **プーリング** データソースは、外部作成/設定された **org.apache.commons.pool2.ObjectPool** で設定されます。

managed vs non-managed

この**軸**は、**接続作成**の要素と JTA の動作を決定します。

- **管理対象外の基本的な** データソースは、**java.sql.Driver.connect()** を内部で使用することで、**java.sql.Connection** インスタンスを作成します。
  管理対象外のプーリング データソースは、渡された **org.apache.commons.pool2.ObjectPool** オブジェクトを使用して **java.sql.Connection** インスタンスを作成します。
- **管理対象のプーリング**データソース
  は、**org.apache.commons.dbcp2.managed.ManagedConnection** オブジェクト内で **java.sql.Connection** インスタンスをラップし、JTA コンテキストで必要に応じて **javax.transaction.Transaction.enlistResource()** が呼び出されるようにします。ただし、ラップされる実際の接続は、プールが設定された **org.apache.commons.pool2.ObjectPool** オブジェクトから取得されます。
  managed basic データソースは、専用の **org.apache.commons.pool2.ObjectPool** の設定を解放します。代わりに、既存の実際のデータベース固有の **javax.sql.XADataSource** オブジェクトを設定すれば十分です。Bean プロパティー
  は、**org.apache.commons.pool2.impl.GenericObjectPool** の内部インスタンスを作成するために使用されます。これは、**管理対象プーリング**のデータソースの内部インスタンス (**org.apache.commons.dbcp2.managed.ManagedDataSource**) に渡されます。



注記

DBCP2 を実行できないのは **XA トランザクションの回復** です。DBCP2 はアクティブな JTA トランザクションで XAResources を正しく登録しますが、リカバリーは実行されません。これは個別に行う必要があり、設定は、通常選択されたトランザクションマネージャーの実装 (Narayana など) に固有のものです。

6.2.3. 使用するパターン

推奨のパターンは以下のとおりです。

- 接続/XA 接続を作成できるデータベース固有の設定 (URL、クレデンシャルなど) で、**データベース固有** の **javax.sql.DataSource** または **javax.sql.XADataSource** インスタンスを作成または取得します。
- データベース固有でない設定 (接続プーリング、トランザクションマネージャーなど) で**データベース固有でない javax.sql.DataSource** インスタンス (上記のデータベース固有のデータソースで内部的設定された) を作成および取得します。
- **javax.sql.DataSource** を使用して **java.sql.Connection** のインスタンスを取得し、JDBC 操作を実行します。

以下は **正規** の例です。

```
// Database-specific, non-pooling, non-enlisting javax.sql.XADataSource
PGXADataSource postgresql = new org.postgresql.xa.PGXADataSource();
// Database-specific configuration
postgresql.setUrl("jdbc:postgresql://localhost:5432/reportdb");
postgresql.setUser("fuse");
postgresql.setPassword("fuse");
postgresql.setCurrentSchema("report");
postgresql.setConnectTimeout(5);
// ...

// Non database-specific, pooling, enlisting javax.sql.DataSource
BasicManagedDataSource pool = new
org.apache.commons.dbcp2.managed.BasicManagedDataSource();
// Delegate to database-specific XADatasource
pool.setXaDataSourceInstance(postgresql);
// Delegate to JTA transaction manager
pool.setTransactionManager(transactionManager);
// Non database-specific configuration
pool.setMinIdle(3);
pool.setMaxTotal(10);
pool.setValidationQuery("select schema_name, schema_owner from
information_schema.schemata");
// ...

// JDBC code:
javax.sql.DataSource applicationDataSource = pool;

try (Connection c = applicationDataSource.getConnection()) {
    try (Statement st = c.createStatement()) {
        try (ResultSet rs = st.executeQuery("select ...")) {
            // ....
```

Fuse 環境では設定オプションが多くなり、DBCP2 を使用する必要はありません。

6.3. JDBC データソースの設定

OSGi トランザクションアーキテクチャー で説明されているように、一部のサービスを OSGi サービスレジストリーに登録する必要があります。**javax.transaction.UserTransaction** インターフェイスなどを使用してトランザクションマネージャーインスタンスを **検索** (**ルックアップ**) できるのと同様に、**javax.sql.DataSource** インターフェイスを使用して JDBC データソースで同じことができます。要件は以下のとおりです。

- ターゲットデータベースと通信できるデータベース固有のデータソース

- プーリングやトランザクション管理 (XA) を設定できる汎用データソース

OSGi 環境では、Fuse などの OSGi 環境では、OSGi サービスとして登録された場合に、アプリケーションからデータソースにアクセスできるようになります。基本的に、これは以下のように行われます。

```
org.osgi.framework.BundleContext.registerService(javax.sql.DataSource.class,
                                                 dataSourceObject,
                                                 properties);
org.osgi.framework.BundleContext.registerService(javax.sql.XADataSource.class,
                                                 xaDataSourceObject,
                                                 properties);
```

このようなサービスを登録する方法は 2 つあります。

- **jdbc:ds-create** Karaf コンソールコマンドを使用してデータソースを公開します。これは、**設定メソッド** です。
- Blueprint、OSGi Declarative Services (SCR)、または **BundleContext.registerService()** API 呼び出しなどの方法を使用してデータソースを公開します。この方法では、コードやメタデータを含む専用の OSGi バンドルが必要です。これは the_deployment method_ です。

6.4. OSGI JDBC サービスの使用

OSGi Enterprise R6 仕様の第 125 章には、 **org.osgi.service.jdbc** パッケージの単一のインターフェイスが定義されています。これは、OSGi がデータソースを処理する方法です。

```
public interface DataSourceFactory {

    java.sql.Driver createDriver(Properties props);

    javax.sql.DataSource createDataSource(Properties props);

    javax.sql.ConnectionPoolDataSource createConnectionPoolDataSource(Properties props);

    javax.sql.XADataSource createXADataSource(Properties props);
}
```

前述のように、**java.sql.Driver** からプレーン **java.sql.Connection** 接続を直接取得できる可能性があります。

Generic org.osgi.service.jdbc.DataSourceFactory

org.osgi.service.jdbc.DataSourceFactory の最も単純な実装は、**mvn:org.ops4j.pax.jdbc/pax-jdbc/1.3.0** バンドルが提供する **org.ops4j.pax.jdbc.impl.DriverDataSourceFactory** です。これはすべて、標準の Java™ **ServiceLoader** ユーティリティーの **/META-INF/services/java.sql.Driver** 記述子が含まれる可能性のあるバンドルを追跡するだけです。標準の JDBC ドライバーをインストールする場合、**pax-jdbc** バンドルは **java.sql.Driver.connect()** 呼び出しを使用して接続を取得するために (直接ではなく) 使用できる **DataSourceFactory** を登録します。

```
karaf@root()> install -s mvn:org.osgi/org.osgi.service.jdbc/1.0.0
Bundle ID: 223
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc/1.3.0
Bundle ID: 224
karaf@root()> install -s mvn:org.postgresql/postgresql/42.2.5
```

```
Bundle ID: 225
karaf@root()> install -s mvn:mysql/mysql-connector-java/5.1.34
Bundle ID: 226

karaf@root()> bundle:services -p org.postgresql.jdbc42

PostgreSQL JDBC Driver JDBC42 (225) provides:
----------------------------------------------
objectClass = [org.osgi.service.jdbc.DataSourceFactory]
osgi.jdbc.driver.class = org.postgresql.Driver
osgi.jdbc.driver.name = PostgreSQL JDBC Driver
osgi.jdbc.driver.version = 42.2.5
service.bundleid = 225
service.id = 242
service.scope = singleton

karaf@root()> bundle:services -p com.mysql.jdbc

Oracle Corporation's JDBC Driver for MySQL (226) provides:
----------------------------------------------------------
objectClass = [org.osgi.service.jdbc.DataSourceFactory]
osgi.jdbc.driver.class = com.mysql.jdbc.Driver
osgi.jdbc.driver.name = com.mysql.jdbc
osgi.jdbc.driver.version = 5.1.34
service.bundleid = 226
service.id = 243
service.scope = singleton
-----
objectClass = [org.osgi.service.jdbc.DataSourceFactory]
osgi.jdbc.driver.class = com.mysql.fabric.jdbc.FabricMySQLDriver
osgi.jdbc.driver.name = com.mysql.jdbc
osgi.jdbc.driver.version = 5.1.34
service.bundleid = 226
service.id = 244
service.scope = singleton

karaf@root()> service:list org.osgi.service.jdbc.DataSourceFactory
[org.osgi.service.jdbc.DataSourceFactory]
----------------------------------------
 osgi.jdbc.driver.class = org.postgresql.Driver
 osgi.jdbc.driver.name = PostgreSQL JDBC Driver
 osgi.jdbc.driver.version = 42.2.5
 service.bundleid = 225
 service.id = 242
 service.scope = singleton
Provided by :
 PostgreSQL JDBC Driver JDBC42 (225)

[org.osgi.service.jdbc.DataSourceFactory]
----------------------------------------
 osgi.jdbc.driver.class = com.mysql.jdbc.Driver
 osgi.jdbc.driver.name = com.mysql.jdbc
 osgi.jdbc.driver.version = 5.1.34
 service.bundleid = 226
 service.id = 243
 service.scope = singleton
```

```
Provided by :
 Oracle Corporation's JDBC Driver for MySQL (226)

[org.osgi.service.jdbc.DataSourceFactory]
----------------------------------------
 osgi.jdbc.driver.class = com.mysql.fabric.jdbc.FabricMySQLDriver
 osgi.jdbc.driver.name = com.mysql.jdbc
 osgi.jdbc.driver.version = 5.1.34
 service.bundleid = 226
 service.id = 244
 service.scope = singleton
Provided by :
 Oracle Corporation's JDBC Driver for MySQL (226)
```

上記のコマンドは、登録までの手順の 1 つで、**javax.sql.DataSource** サービスはまだ登録されません。上記の **org.osgi.service.jdbc.DataSourceFactory** の中間サービスを使用して以下を取得できます。

- **java.sql.Driver**
- **url**、**user** および **password** のプロパティーを **createDataSource()** メソッドに渡す **javax.sql.DataSource**。

データベース固有ではない **pax-jdbc** で作成された汎用の **org.osgi.service.jdbc.DataSourceFactory** から **javax.sql.ConnectionPoolDataSource** または **javax.sql.XADataSource** を取得できません。

注記

mvn:org.postgresql/postgresql/42.2.5 バンドルは OSGi JDBC 仕様を正しく実装し、XA および ConnectionPool データソースを作成するメソッドを含む、実装されているすべてのメソッドで **org.osgi.service.jdbc.DataSourceFactory** インスタンスを登録します。

データベース固有の org.osgi.service.jdbc.DataSourceFactory 専用実装

以下のような追加のバンドルがあります。

- **mvn:org.ops4j.pax.jdbc/pax-jdbc-mysql/1.3.0**
- **mvn:org.ops4j.pax.jdbc/pax-jdbc-db2/1.3.0**
- ...

これらのバンドルは、**javax.sql.ConnectionPoolDataSource** および **javax.sql.XADataSource** など、全種の **ファクトリー** を返すことができる、データベース固有の **org.osgi.service.jdbc.DataSourceFactory** サービスを登録します。以下に例を示します。

```
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-mysql/1.3.0
Bundle ID: 227

karaf@root()> bundle:services -p org.ops4j.pax.jdbc.mysql

OPS4J Pax JDBC MySQL Driver Adapter (227) provides:
---------------------------------------------------
objectClass = [org.osgi.service.jdbc.DataSourceFactory]
osgi.jdbc.driver.class = com.mysql.jdbc.Driver
osgi.jdbc.driver.name = mysql
```

```
service.bundleid = 227
service.id = 245
service.scope = singleton

karaf@root()> service:list org.osgi.service.jdbc.DataSourceFactory
...
[org.osgi.service.jdbc.DataSourceFactory]
----------------------------------------
 osgi.jdbc.driver.class = com.mysql.jdbc.Driver
 osgi.jdbc.driver.name = mysql
 service.bundleid = 227
 service.id = 245
 service.scope = singleton
Provided by :
 OPS4J Pax JDBC MySQL Driver Adapter (227)
```

6.4.1. PAX-JDBC 設定サービス

pax-jdbc (または **pax-jdbc-mysql**、**pax-jdbc-oracle** など) のバンドルでは、**org.osgi.service.jdbc.DataSourceFactory** サービスを登録でき、それを使用して、指定のデータベースのデータソースを取得できます (「データベース固有のデータソースおよび汎用データソース」参照)。しかし、実際のデータソースはまだありません。

mvn:org.ops4j.pax.jdbc/pax-jdbc-config/1.3.0 バンドルは、以下の 2 つを行う管理対象サービスファクトリーを提供します。

- メソッドを呼び出すために、**org.osgi.service.jdbc.DataSourceFactory** OSGi サービスを追跡します。

```
public DataSource createDataSource(Properties props);
public XADataSource createXADataSource(Properties props);
public ConnectionPoolDataSource createConnectionPoolDataSource(Properties props);
```

- **org.ops4j.datasource ファクトリー PID** を追跡し、上記のメソッドに必要なプロパティーを収集します。たとえば、**${karaf.etc}/org.ops4j.datasource-mysql.cfg** ファイルの作成など、Configuration Admin サービスで使用可能な任意の方法を使用して、**ファクトリー設定**を作成する場合は、最終的な手順を実行して、実際のデータベース固有のデータソースを公開できます。

以下は、Fuse の新規インストールを開始するために順をおって説明した **標準的な** ガイドです。

注記

機能の代わりにバンドルを明示的にインストールし、必要なバンドルを正確に表示します。便宜上、PAX JDBC プロジェクトは、複数のデータベース製品および設定アプローチの機能を提供します。

1. **/META-INF/services/java.sql.Driver** で JDBC ドライバーをインストールします。

```
karaf@root()> install -s mvn:mysql/mysql-connector-java/5.1.34
Bundle ID: 223
```

2. OSGi JDBC サービスバンドルと **intermediaryorg.osgi.service.jdbc.DataSourceFactory** を登録する **pax-jdbc-mysql** バンドルをインストールします。

```
karaf@root()> install -s mvn:org.osgi/org.osgi.service.jdbc/1.0.0
Bundle ID: 224
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-mysql/1.3.0
Bundle ID: 225

karaf@root()> service:list org.osgi.service.jdbc.DataSourceFactory
[org.osgi.service.jdbc.DataSourceFactory]
----------------------------------------
 osgi.jdbc.driver.class = com.mysql.jdbc.Driver
 osgi.jdbc.driver.name = mysql
 service.bundleid = 225
 service.id = 242
 service.scope = singleton
Provided by :
 OPS4J Pax JDBC MySQL Driver Adapter (225)
```

3. **pax-jdbc** バンドルおよび、**org.osgi.service.jdbc.DataSourceFactory** サービス **およびorg.ops4j.datasource ファクトリー PID** を追跡する **pax-jdbc-config** バンドルをインストールします。

```
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc/1.3.0
Bundle ID: 226
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-pool-common/1.3.0
Bundle ID: 227
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-config/1.3.0
Bundle ID: 228

karaf@root()> bundle:services -p org.ops4j.pax.jdbc.config

OPS4J Pax JDBC Config (228) provides:
-------------------------------------
objectClass = [org.osgi.service.cm.ManagedServiceFactory]
service.bundleid = 228
service.id = 245
service.pid = org.ops4j.datasource
service.scope = singleton
```

4. **ファクトリー設定** を作成します (MySQL サーバーが実行中であると想定します)。

```
karaf@root()> config:edit --factory --alias mysql org.ops4j.datasource
karaf@root()> config:property-set osgi.jdbc.driver.name mysql
karaf@root()> config:property-set dataSourceName mysqlds
karaf@root()> config:property-set url jdbc:mysql://localhost:3306/reportdb
karaf@root()> config:property-set user fuse
karaf@root()> config:property-set password fuse
karaf@root()> config:update

karaf@root()> config:list '(service.factoryPid=org.ops4j.datasource)'
----------------------------------------------------------------
Pid:            org.ops4j.datasource.a7941498-9b62-4ed7-94f3-8c7ac9365313
FactoryPid:     org.ops4j.datasource
BundleLocation: ?
Properties:
   dataSourceName = mysqlds
   felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.datasource-mysql.cfg
```

```
  osgi.jdbc.driver.name = mysql
  password = fuse
  service.factoryPid = org.ops4j.datasource
  service.pid = org.ops4j.datasource.a7941498-9b62-4ed7-94f3-8c7ac9365313
  url = jdbc:mysql://localhost:3306/reportdb
  user = fuse
```

5. **pax-jdbc-config** が設定を処理して **javax.sql.DataSource** サービスに指定しているかどうかを確認します。

```
karaf@root()> service:list javax.sql.DataSource
[javax.sql.DataSource]
----------------------
 dataSourceName = mysqlds
 felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.datasource-mysql.cfg
 osgi.jdbc.driver.name = mysql
 osgi.jndi.service.name = mysqlds
 password = fuse
 pax.jdbc.managed = true
 service.bundleid = 228
 service.factoryPid = org.ops4j.datasource
 service.id = 246
 service.pid = org.ops4j.datasource.a7941498-9b62-4ed7-94f3-8c7ac9365313
 service.scope = singleton
 url = jdbc:mysql://localhost:3306/reportdb
 user = fuse
Provided by :
 OPS4J Pax JDBC Config (228)
```

これで、実際のデータベース固有 (まだプールなし) データソースができました。必要な場所に注入することができます。たとえば、Karaf コマンドを使用してデータベースにクエリーすることができます。

```
karaf@root()> feature:install -v jdbc
Adding features: jdbc/[4.2.0.fuse-000237-redhat-1,4.2.0.fuse-000237-redhat-1]
...
karaf@root()> jdbc:ds-list
Mon May 14 08:46:22 CEST 2018 WARN: Establishing SSL connection without server's identity
verification is not recommended. According to MySQL 5.5.45+, 5.6.26+ and 5.7.6+ requirements SSL
connection must be established by default if explicit option isn't set. For compliance with existing
applications not using SSL the verifyServerCertificate property is set to 'false'. You need either to
explicitly disable SSL by setting useSSL=false, or set useSSL=true and provide truststore for server
certificate verification.
Name    | Product | Version | URL                          | Status
--------+---------+---------+------------------------------+--------
mysqlds | MySQL   | 5.7.21  | jdbc:mysql://localhost:3306/reportdb | OK

karaf@root()> jdbc:query mysqlds 'select * from incident'
Mon May 14 08:46:46 CEST 2018 WARN: Establishing SSL connection without server's identity
verification is not recommended. According to MySQL 5.5.45+, 5.6.26+ and 5.7.6+ requirements SSL
connection must be established by default if explicit option isn't set. For compliance with existing
applications not using SSL the verifyServerCertificate property is set to 'false'. You need either to
explicitly disable SSL by setting useSSL=false, or set useSSL=true and provide truststore for server
certificate verification.
date                | summary    | name   | details                 | id | email
```

```
2018-02-20 08:00:00.0 | Incident 1 | User 1 | This is a report incident 001 | 1 |
user1@redhat.com
2018-02-20 08:10:00.0 | Incident 2 | User 2 | This is a report incident 002 | 2 |
user2@redhat.com
2018-02-20 08:20:00.0 | Incident 3 | User 3 | This is a report incident 003 | 3 |
user3@redhat.com
2018-02-20 08:30:00.0 | Incident 4 | User 4 | This is a report incident 004 | 4 |
user4@redhat.com
```

上記の例では、MySQL の警告が表示されます。これは問題ではありません。すべてのプロパティー (OSGi JDBC 固有のものだけでなく) が提供されます。

```
karaf@root()> config:property-set --pid org.ops4j.datasource.a7941498-9b62-4ed7-94f3-
8c7ac9365313 useSSL false

karaf@root()> jdbc:ds-list
Name    | Product | Version | URL                          | Status
--------+---------+---------+------------------------------+-------
mysqlds | MySQL   | 5.7.21  | jdbc:mysql://localhost:3306/reportdb | OK
```

6.4.2. 処理されたプロパティーの概要

管理 **ファクトリー PID** の設定からのプロパティーは、関連する **org.osgi.service.jdbc.DataSourceFactory** 実装に渡されます。

Generic

org.ops4j.pax.jdbc.impl.DriverDataSourceFactory properties:

- **url**
- **user**
- **password**

DB2

org.ops4j.pax.jdbc.db2.impl.DB2DataSourceFactory プロパティーには、以下の実装クラスのすべての bean プロパティーが含まれます。

- **com.ibm.db2.jcc.DB2SimpleDataSource**
- **com.ibm.db2.jcc.DB2ConnectionPoolDataSource**
- **com.ibm.db2.jcc.DB2XADataSource**

PostgreSQL

Nnative **org.postgresql.osgi.PGDataSourceFactory** プロパティーには、**org.postgresql.PGProperty** に指定されたすべてのプロパティーが含まれます。

HSQLDB

org.ops4j.pax.jdbc.hsqldb.impl.HsqldbDataSourceFactory properties:

- **url**
- **user**
- **password**
- **databaseName**
- すべての Bean プロパティー
  - **org.hsqldb.jdbc.JDBCDataSource**
  - **org.hsqldb.jdbc.pool.JDBCPooledDataSource**
  - **org.hsqldb.jdbc.pool.JDBCXADataSource**

SQL Server および Sybase

org.ops4j.pax.jdbc.jtds.impl.JTDSDataSourceFactory プロパティーに
は、**net.sourceforge.jtds.jdbcx.JtdsDataSource** のすべての Bean プロパティーが含まれます。

SQL Server

org.ops4j.pax.jdbc.mssql.impl.MSSQLDataSourceFactory プロパティー:

- **url**
- **user**
- **password**
- **databaseName**
- **serverName**
- **portNumber**
- すべての Bean プロパティー
  - **com.microsoft.sqlserver.jdbc.SQLServerDataSource**
  - **com.microsoft.sqlserver.jdbc.SQLServerConnectionPoolDataSource**
  - **com.microsoft.sqlserver.jdbc.SQLServerXADataSource**

MySQL

org.ops4j.pax.jdbc.mysql.impl.MysqlDataSourceFactory properties:

- **url**
- **user**
- **password**
- **databaseName**
- **serverName**

- **portNumber**
- すべての Bean プロパティー
  - **com.mysql.jdbc.jdbc2.optional.MysqlDataSource**
  - **com.mysql.jdbc.jdbc2.optional.MysqlConnectionPoolDataSource**
  - **com.mysql.jdbc.jdbc2.optional.MysqlXADataSource**

Oracle

org.ops4j.pax.jdbc.oracle.impl.OracleDataSourceFactory properties:

- **url**
- **databaseName**
- **serverName**
- **user**
- **password**
- すべての Bean プロパティー
  - **oracle.jdbc.pool.OracleDataSource**
  - **oracle.jdbc.pool.OracleConnectionPoolDataSource**
  - **oracle.jdbc.xa.client.OracleXADataSource**

SQLite

org.ops4j.pax.jdbc.sqlite.impl.SqliteDataSourceFactory properties:

- **url**
- **databaseName**
- **org.sqlite.SQLiteDataSource** のすべての Bean プロパティー

6.4.3. pax-jdb-config バンドルがプロパティーを処理する方法

pax-jdbc-config バンドルは、**jdbc.** で始まるプロパティーを処理します。これらのプロパティーを使用すると、すべてこの接頭辞が削除され、残りの名前が引き継がれます。

以下に例を示します。こちらも、Fuse の新規インストールから始めます。

```
karaf@root()> install -s mvn:mysql/mysql-connector-java/5.1.34
Bundle ID: 223
karaf@root()> install -s mvn:org.osgi/org.osgi.service.jdbc/1.0.0
Bundle ID: 224
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-mysql/1.3.0
Bundle ID: 225
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc/1.3.0
Bundle ID: 226
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-pool-common/1.3.0
```

```
Bundle ID: 227
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-config/1.3.0
Bundle ID: 228

karaf@root()> config:edit --factory --alias mysql org.ops4j.datasource
karaf@root()> config:property-set osgi.jdbc.driver.name mysql
karaf@root()> config:property-set dataSourceName mysqlds
karaf@root()> config:property-set dataSourceType DataSource
karaf@root()> config:property-set jdbc.url jdbc:mysql://localhost:3306/reportdb
karaf@root()> config:property-set jdbc.user fuse
karaf@root()> config:property-set jdbc.password fuse
karaf@root()> config:property-set jdbc.useSSL false
karaf@root()> config:update

karaf@root()> config:list '(service.factoryPid=org.ops4j.datasource)'
----------------------------------------------------------------
Pid:            org.ops4j.datasource.7c3ee718-7309-46a0-ae3a-64b38b17a0a3
FactoryPid:     org.ops4j.datasource
BundleLocation: ?
Properties:
   dataSourceName = mysqlds
   dataSourceType = DataSource
   felix.fileinstall.filename = file:/data/servers/7.10.0.fuse-7_10_0-00010-redhat-
00001/etc/org.ops4j.datasource-mysql.cfg
   jdbc.password = fuse
   jdbc.url = jdbc:mysql://localhost:3306/reportdb
   jdbc.useSSL = false
   jdbc.user = fuse
   osgi.jdbc.driver.name = mysql
   service.factoryPid = org.ops4j.datasource
   service.pid = org.ops4j.datasource.7c3ee718-7309-46a0-ae3a-64b38b17a0a3

karaf@root()> service:list javax.sql.DataSource
[javax.sql.DataSource]
----------------------
 dataSourceName = mysqlds
 dataSourceType = DataSource
 felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.datasource-mysql.cfg
 jdbc.password = fuse
 jdbc.url = jdbc:mysql://localhost:3306/reportdb
 jdbc.user = fuse
 jdbc.useSSL = false
 osgi.jdbc.driver.name = mysql
 osgi.jndi.service.name = mysqlds
 pax.jdbc.managed = true
 service.bundleid = 228
 service.factoryPid = org.ops4j.datasource
 service.id = 246
 service.pid = org.ops4j.datasource.7c3ee718-7309-46a0-ae3a-64b38b17a0a3
 service.scope = singleton
Provided by :
 OPS4J Pax JDBC Config (228)
```

pax-jdbc-config バンドルには以下のプロパティーが必要です

- **osgi.jdbc.driver.name**

- **dataSourceName**
- **dataSourceType**

関連する **org.osgi.service.jdbc.DataSourceFactory** メソッドを見つけて呼び出すには、以下を実行します。**jdbc.** の接頭辞が付けられたプロパティーは (接頭辞の削除後)、**org.osgi.service.jdbc.DataSourceFactory.createDataSource(properties)** などに渡されます。ただし、これらのプロパティーは、**javax.sql.DataSource** OSGi サービスなどのプロパティーとして、接頭辞を削除せずに追加されます。

6.5. JDBC コンソールコマンドの使用

Fuse は、**jdbc:*** スコープのシェルコマンドが含まれる **jdbc** 機能を提供します。前述の例は、**jdbc:query** の使用を示しています。Configuration Admin 設定を作成する必要性を隠すコマンドもあります。

Fuse の新しいインスタンスから始めて、データベース固有のデータソースを汎用 **DataSourceFactory** サービスに登録するには、次のようにします。

```
karaf@root()> feature:install jdbc

karaf@root()> jdbc:ds-factories
Name │ Class │ Version
─────┼───────┼────────

karaf@root()> install -s mvn:mysql/mysql-connector-java/5.1.34
Bundle ID: 228

karaf@root()> jdbc:ds-factories
Name          │ Class                          │ Version
──────────────┼────────────────────────────────┼──────
com.mysql.jdbc │ com.mysql.jdbc.Driver                │ 5.1.34
com.mysql.jdbc │ com.mysql.fabric.jdbc.FabricMySQLDriver │ 5.1.34
```

MySQL 固有の **DataSourceFactory** サービスを登録する例を以下に示します。

```
karaf@root()> feature:repo-add mvn:org.ops4j.pax.jdbc/pax-jdbc-features/1.3.0/xml/features-gpl
Adding feature url mvn:org.ops4j.pax.jdbc/pax-jdbc-features/1.3.0/xml/features-gpl

karaf@root()> feature:install pax-jdbc-mysql

karaf@root()> la -l|grep mysql

232 │ Active   │  80 │ 5.1.34              │ mvn:mysql/mysql-connector-java/5.1.34

233 │ Active   │  80 │ 1.3.0               │ mvn:org.ops4j.pax.jdbc/pax-jdbc-mysql/1.3.0

karaf@root()> jdbc:ds-factories
Name          │ Class                          │ Version
──────────────┼────────────────────────────────┼──────
com.mysql.jdbc │ com.mysql.jdbc.Driver                │ 5.1.34
mysql         │ com.mysql.jdbc.Driver                │
com.mysql.jdbc │ com.mysql.fabric.jdbc.FabricMySQLDriver │ 5.1.34
```

上記の表は混乱を招くかもしれませんが、前述のように、 **pax-jdbc-database** バンドルの1つのみが、単に **java.sql.Driver.connect()** へ委任**しない** 標準/XA/接続プールデータソースを作成できる **org.osgi.service.jdbc.DataSourceFactory** インスタンスを登録する可能性があります。

以下の例では、MySQL データソースを作成して確認します。

```
karaf@root()> jdbc:ds-create -dt DataSource -dn mysql -url 'jdbc:mysql://localhost:3306/reportdb?
useSSL=false' -u fuse -p fuse mysqlds

karaf@root()> jdbc:ds-list
Name    │ Product │ Version │ URL                                          │ Status
────────┼─────────┼─────────┼──────────────────────────────────────────────┼──────
mysqlds │ MySQL   │ 5.7.21  │ jdbc:mysql://localhost:3306/reportdb?useSSL=false │ OK

karaf@root()> jdbc:query mysqlds 'select * from incident'
date                │ summary    │ name   │ details                │ id │ email
────────────────────┼────────────┼────────┼────────────────────────┼────┼──────
2018-02-20 08:00:00.0 │ Incident 1 │ User 1 │ This is a report incident 001 │ 1  │
user1@redhat.com
2018-02-20 08:10:00.0 │ Incident 2 │ User 2 │ This is a report incident 002 │ 2  │
user2@redhat.com
2018-02-20 08:20:00.0 │ Incident 3 │ User 3 │ This is a report incident 003 │ 3  │
user3@redhat.com
2018-02-20 08:30:00.0 │ Incident 4 │ User 4 │ This is a report incident 004 │ 4  │
user4@redhat.com

karaf@root()> config:list '(service.factoryPid=org.ops4j.datasource)'
----------------------------------------------------------------
Pid:            org.ops4j.datasource.55b18993-de4e-4e0b-abb2-a4c13da7f78b
FactoryPid:     org.ops4j.datasource
BundleLocation: mvn:org.ops4j.pax.jdbc/pax-jdbc-config/1.3.0
Properties:
   dataSourceName = mysqlds
   dataSourceType = DataSource
   osgi.jdbc.driver.name = mysql
   password = fuse
   service.factoryPid = org.ops4j.datasource
   service.pid = org.ops4j.datasource.55b18993-de4e-4e0b-abb2-a4c13da7f78b
   url = jdbc:mysql://localhost:3306/reportdb?useSSL=false
   user = fuse
```

ご覧のとおり、**org.ops4j.datasource** ファクトリー PID が作成されます。ただし、 **config:update** では可能ですが、これは自動的に **${karaf.etc}** に保存されません。

6.6. 暗号化された設定値の使用

pax-jdbc-config 機能は、値が暗号化された Configuration Admin 設定を処理できます。一般的なソリューションは、Blueprint でも使用される Jasypt 暗号化サービスを使用することです。

alias サービスプロパティーで OSGi に登録されている **org.jasypt.encryption.StringEncryptor** サービスプロパティーがある場合は、データソース**ファクトリー PID** で参照し、暗号化されたパスワードを使用することができます。以下に例を示します。

```
felix.fileinstall.filename = */etc/org.ops4j.datasource-mysql.cfg
dataSourceName = mysqlds
dataSourceType = DataSource
decryptor = my-jasypt-decryptor
osgi.jdbc.driver.name = mysql
url = jdbc:mysql://localhost:3306/reportdb?useSSL=false
user = fuse
password = ENC(<encrypted-password>)
```

復号化サービスの検索に使用するサービスフィルターは **(&(objectClass=org.jasypt.encryption.StringEncryptor)(alias=<alias>))** です。**<alias>** は、データソース設定の **ファクトリー PID** からの **decryptor** プロパティーの値になります。

6.7. JDBC 接続プールの使用

本セクションでは、JDBC 接続プールの使用の概要と、以下の接続プールモジュールの使用方法を説明します。

- pax-jdbc-pool-dbcp2
- pax-jdbc-pool-narayana
- pax-jdbc-pool-transx

重要

本章では、データソース管理の内部に関する包括的な情報を紹介しています。DBCP2 接続プール機能に関する情報が提供されていますが、この接続プールには適切な JTA 参加機能がありますが、**XA リカバリー** の機能はないので注意してください。

XA リカバリー が有効であることを確認するには、**pax-jdbc-pool-transx** または **pax-jdbc-pool-narayana** 接続プールモジュールを使用します。

6.7.1. JDBC 接続プールの使用について

以前の例では、データベース固有のデータソース **ファクトリー** を登録する方法を説明しました。**データソース** 自体は接続のファクトリーであるため、**org.osgi.service.jdbc.DataSourceFactory** は 3 種類のデータソースを生成できる**メタファクトリー** として扱われる可能性があります (さらに **java.sql.Driver**)。

- **javax.sql.DataSource**
- **javax.sql.ConnectionPoolDataSource**
- **javax.sql.XADataSource**

たとえば、**pax-jdbc-mysql** は、以下を作成する **org.ops4j.pax.jdbc.mysql.impl.MysqlDataSourceFactory** を登録します。

- **javax.sql.DataSource → com.mysql.jdbc.jdbc2.optional.MysqlDataSource**
- **javax.sql.ConnectionPoolDataSource → com.mysql.jdbc.jdbc2.optional.MysqlConnectionPoolDataSource**
- **javax.sql.XADataSource → com.mysql.jdbc.jdbc2.optional.MysqlXADataSource**

- **java.sql.Driver → com.mysql.jdbc.Driver**

PostgreSQL ドライバー自体は OSGi JDBC サービスを実装し、以下を生成します。

- **javax.sql.DataSource → org.postgresql.jdbc2.optional.PoolingDataSource** (プール関連のプロパティーが指定されている場合) または **org.postgresql.jdbc2.optional.SimpleDataSource**
- **javax.sql.ConnectionPoolDataSource → org.postgresql.jdbc2.optional.ConnectionPool**
- **javax.sql.XADataSource → org.postgresql.xa.PGXADataSource**
- **java.sql.Driver → org.postgresql.Driver**

標準の DataSource 例 に記載されているように、JTA 環境で機能する場合は、**プーリング**、**汎用** データソースは、(XA) 接続を実際に取得するには **データベース固有** のデータソースが必要です。

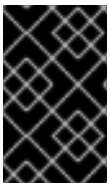
すでに後者があるので、実際の信頼性がある、汎用接続プールが必要です。

標準の DataSource 例 は、データベース固有のデータソースを使用して汎用プールを設定する方法を示しています。**pax-jdbc-pool-*** バンドルは上記の **org.osgi.service.jdbc.DataSourceFactory** サービスとスムーズに連携します。

OSGI Enterprise R6 JDBC 仕様が **org.osgi.service.jdbc.DataSourceFactory** 標準インターフェイスを提供するのと同じように、**pax-jdbc-pool-common** は **プロプライエタリーの org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory** インターフェイスを提供します。

```
public interface PooledDataSourceFactory {

    javax.sql.DataSource create(org.osgi.service.jdbc.DataSourceFactory dsf, Properties config)

}
```

このインターフェイスは、以前に提示されたこの重要なメモに完全に準拠しており、繰り返す価値があります。



重要

アプリケーションが XA トランザクションや接続プーリングを使用する場合でも、アプリケーションは他の 2 つの JDBC データソースインターフェイスではなく、**javax.sql.DataSource** と対話します。

このインターフェイスは、データベース固有の非プーリングデータからプーリングデータソースを作成するだけです。また、これは **データソースファクトリー**(**メタファクトリー**) で、データベース固有のデータソースのファクトリーをデータソースのプーリングファクトリーに変換します。

注記

javax.sql.DataSource オブジェクトのプールをすでに返している **org.osgi.service.jdbc.DataSourceFactory** サービスを使用して、アプリケーションで **javax.sql.DataSource** オブジェクトのプーリングが設定されないようにすることはできません。

以下の表は、プールされたデータソースファクトリーを登録するバンドルを示しています。この表では、**o.o.p.j.p** のインスタンスは **org.ops4j.pax.jdbc.pool** を表します。

バンドル	PooledDataSourceFactory	プールキー
pax-jdbc-pool-narayana	**o.o.p.j.p.narayana.impl.Dbcp(XA)PooledDataSourceFactory**	**narayana**
pax-jdbc-pool-dbcp2	**o.o.p.j.p.dbcp2.impl.Dbcp(XA)PooledDataSourceFactory**	**dbcp2**
pax-jdbc-pool-transx	**o.o.p.j.p.transx.impl.Transx(Xa)PooledDataSourceFactory**	**transx**

上記のバンドルはデータソースファクトリーのみをインストールし、データソース自体にはインストールしません。アプリケーションは **javax.sql.DataSource create(org.osgi.service.jdbc.DataSourceFactory dsf, Properties config)** メソッドを呼び出す何かが必要です。

6.7.2. dbcp2 接続プールモジュールの使用

一般的なデータソースのセクションでは、Apache Commons DBCP モジュール を使用し、設定する方法の例 を紹介します。ここでは、Fuse OSGi 環境で設定方法を説明します。

「PAX-JDBC 設定サービス」 バンドルについて見てみましょう。以下を追跡するだけでなく、ほかも追跡します。

- **org.osgi.service.jdbc.DataSourceFactory** services
- **org.ops4j.datasource** factory PIDs

このバンドルは、**pax-jdbc-pool-*** バンドルのいずれかによって登録された **org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory** のインスタンスも追跡します。

ファクトリー設定 に **pool** プロパティーが含まれる場合、**pax-jdbc-config** バンドルによって登録される最終データソースはデータベース固有のデータソースになりますが、**pool=dbcp2** の場合は以下のいずれかの内部でラッピングされます。

- **org.apache.commons.dbcp2.PoolingDataSource**
- **org.apache.commons.dbcp2.managed.ManagedDataSource**

これは、一般的なデータソースの例 と同じです。**pool** プロパティーと、非 xa または xa データソースを選択するブール値 **xa** プロパティーの他に、**org.ops4j.datasource ファクトリー PID** に **接頭辞** が付いたプロパティーが含まれることがあります。

- **pool.***
- **factory.***

各プロパティーが使用される場所は、使用される **pax-jdbc-pool-*** バンドルによって異なります。DBCP2 の場合は以下のようになります。

- **pool.***: **org.apache.commons.pool2.impl.GenericObjectPoolConfig** の bean プロパティー (xa と xa 以外の両方)

- **factory.***: **org.apache.commons.dbcp2.managed.PoolableManagedConnectionFactory** (xa) または **org.apache.commons.dbcp2.PoolableConnectionFactory** (xa 以外) の Bean プロパティー

6.7.2.1. BasicDataSource の設定プロパティー

以下の表は、BasicDataSource の汎用設定プロパティーを示しています。

パラメーター	デフォルト	説明
username		接続を確立するために JDBC ドライバーに渡される接続ユーザー名。
password		接続を確立するために JDBC ドライバーに渡される接続パスワード。
url		接続を確立するために JDBC ドライバーに渡される接続 URL。
driverClassName		使用する JDBC ドライバーの完全修飾 Java クラス名。
initialSize	0	プールの開始時に作成される接続の初期数。
maxTotal	8	このプールから同時に割り当てることができるアクティブな接続の最大数、または制限なしの場合は負の数。
maxIdle	8	追加の接続が解放されることなく、プール内でアイドル状態を維持できる接続の最大数、または無制限の場合は負の接続。
minIdle	0	追加の接続を作成せずにプール内でアイドル状態を維持できる接続の最小数、または作成しない場合はゼロ。
maxWaitMillis	無期限	例外を出力する前に接続が返されるまでプールが待機する最大ミリ秒数 (使用可能な接続がない場合)、または無期限に待機する場合は -1。
validationQuery		呼び出し元に接続を返す前に、このプールからの接続を検証するために使用される SQL クエリー。指定されている場合、このクエリーは、SQL SELECT ステートメントであり、最低でも 1 行を返す必要があります。指定のない場合は、isValid() メソッドを呼び出して接続が検証されます。

パラメーター	デフォルト	説明
validationQueryTimeout	タイムアウトなし	接続検証クエリーが失敗するまでのタイムアウト時間 (秒単位)。正の値に設定されている場合、この値は、検証クエリーの実行に使用されるステートメントの setQueryTimeout メソッドを介してドライバーに渡されます。
testOnCreate	false	作成後にオブジェクトが検証されるかどうか。オブジェクトの検証に失敗した場合には、オブジェクトの作成をトリガーした borrow の試行に失敗します。
testOnBorrow	true	プールからオブジェクトを借りる前に、オブジェクトを検証するかどうか。オブジェクトが検証に失敗すると、プールから破棄され、別のオブジェクトの借用を試みます。
testOnReturn	false	プールへ返す前にオブジェクトが検証されるかどうか。
testWhileIdle	false	アイドルオブジェクトのエビクターによってオブジェクトが検証されるかどうか (ある場合)。オブジェクトが検証できない場合には、プールからオブジェクトは削除されます。
timeBetweenEvictionRunsMillis	-1	アイドルオブジェクトのエビクタースレッドを次に実行するまでの間でどれだけ待機してからスリープするか (ミリ秒単位)。正の値以外の場合は、アイドル状態のオブジェクトのエビクタースレッドは実行されません。
numTestsPerEvictionRun	3	アイドルオブジェクトのエビクタースレッド (ある場合) のそれぞれの実行時に検査するオブジェクトの数。
minEvictableIdleTimeMillis	1000 * 60 * 30	アイドルオブジェクトのエビクターの対象となるまでにオブジェクトがプールでアイドル状態で待機できる最小時間 (ある場合)。

6.7.2.2. DBCP2 プールの設定方法の例

以下は、**jdbc.** が接頭辞として付けられたプロパティーと便利な構文を使用する DBCP2 プールの設定 (**org.ops4j.datasource-mysql**ファクトリー PID) の現実的な例です (**useSSL=false** を除く)。

```
# Configuration for pax-jdbc-config to choose and configure specific
```

```
org.osgi.service.jdbc.DataSourceFactory
dataSourceName = mysqlds
dataSourceType = DataSource
osgi.jdbc.driver.name = mysql
jdbc.url = jdbc:mysql://localhost:3306/reportdb
jdbc.user = fuse
jdbc.password = fuse
jdbc.useSSL = false

# Hints for pax-jdbc-config to use org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory
pool = dbcp2
xa = false

# dbcp2 specific configuration of org.apache.commons.pool2.impl.GenericObjectPoolConfig
pool.minIdle = 10
pool.maxTotal = 100
pool.initialSize = 8
pool.blockWhenExhausted = true
pool.maxWaitMillis = 2000
pool.testOnBorrow = true
pool.testWhileIdle = false
pool.timeBetweenEvictionRunsMillis = 120000
pool.evictionPolicyClassName = org.apache.commons.pool2.impl.DefaultEvictionPolicy

# dbcp2 specific configuration of org.apache.commons.dbcp2.PoolableConnectionFactory
factory.maxConnLifetimeMillis = 30000
factory.validationQuery  = select schema_name from information_schema.schemata
factory.validationQueryTimeout = 2
```

上記の設定では、**pool** と **xa** キーは **ヒント** (サービスフィルタープロパティー) で、登録された **org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory** サービスのいずれかを選択します。DBCP2 の場合は以下のようになります。

```
karaf@root()> feature:install pax-jdbc-pool-dbcp2

karaf@root()> bundle:services -p org.ops4j.pax.jdbc.pool.dbcp2

OPS4J Pax JDBC Pooling DBCP2 (230) provides:
----------------------------------------------
objectClass = [org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory]
pool = dbcp2
service.bundleid = 230
service.id = 337
service.scope = singleton
xa = false
-----
objectClass = [org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory]
pool = dbcp2
service.bundleid = 230
service.id = 338
service.scope = singleton
xa = true
```

以下は、前の例 に接続プール設定を追加した完全な例です。繰り返しになりますが、これは、Fuse の新規インストールから開始していることを前提としています。

1. JDBC ドライバーをインストールします。

```
karaf@root()> install -s mvn:mysql/mysql-connector-java/5.1.34
Bundle ID: 223
```

2. **jdbc**、**pax-jdbc-mysql** および **pax-jdbc-pool-dbcp2** 機能をインストールします。

```
karaf@root()> feature:repo-add mvn:org.ops4j.pax.jdbc/pax-jdbc-features/1.3.0/xml/features-
gpl
Adding feature url mvn:org.ops4j.pax.jdbc/pax-jdbc-features/1.3.0/xml/features-gpl

karaf@root()> feature:install jdbc pax-jdbc-mysql pax-jdbc-pool-dbcp2

karaf@root()> service:list org.osgi.service.jdbc.DataSourceFactory
...
[org.osgi.service.jdbc.DataSourceFactory]
----------------------------------------
 osgi.jdbc.driver.class = com.mysql.jdbc.Driver
 osgi.jdbc.driver.name = mysql
 service.bundleid = 232
 service.id = 328
 service.scope = singleton
Provided by :
 OPS4J Pax JDBC MySQL Driver Adapter (232)

karaf@root()> service:list org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory
[org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory]
----------------------------------------------------------
 pool = dbcp2
 service.bundleid = 233
 service.id = 324
 service.scope = singleton
 xa = false
Provided by :
 OPS4J Pax JDBC Pooling DBCP2 (233)

[org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory]
----------------------------------------------------------
 pool = dbcp2
 service.bundleid = 233
 service.id = 332
 service.scope = singleton
 xa = true
Provided by :
 OPS4J Pax JDBC Pooling DBCP2 (233)
```

3. **ファクトリー設定** を作成します。

```
karaf@root()> config:edit --factory --alias mysql org.ops4j.datasource
karaf@root()> config:property-set osgi.jdbc.driver.name mysql
karaf@root()> config:property-set dataSourceName mysqlds
karaf@root()> config:property-set dataSourceType DataSource
karaf@root()> config:property-set jdbc.url jdbc:mysql://localhost:3306/reportdb
karaf@root()> config:property-set jdbc.user fuse
karaf@root()> config:property-set jdbc.password fuse
```

```
karaf@root()> config:property-set jdbc.useSSL false
karaf@root()> config:property-set pool dbcp2
karaf@root()> config:property-set xa false
karaf@root()> config:property-set pool.minIdle 2
karaf@root()> config:property-set pool.maxTotal 10
karaf@root()> config:property-set pool.blockWhenExhausted true
karaf@root()> config:property-set pool.maxWaitMillis 2000
karaf@root()> config:property-set pool.testOnBorrow true
karaf@root()> config:property-set pool.testWhileIdle alse
karaf@root()> config:property-set pool.timeBetweenEvictionRunsMillis 120000
karaf@root()> config:property-set factory.validationQuery 'select schema_name from
information_schema.schemata'
karaf@root()> config:property-set factory.validationQueryTimeout 2
karaf@root()> config:update
```

4. **pax-jdbc-config** が設定を処理して **javax.sql.DataSource** サービスに指定しているかどうかを確認します。

```
karaf@root()> service:list javax.sql.DataSource
[javax.sql.DataSource]
----------------------
 dataSourceName = mysqlds
 dataSourceType = DataSource
 factory.validationQuery = select schema_name from information_schema.schemata
 factory.validationQueryTimeout = 2
 felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.datasource-mysql.cfg
 jdbc.password = fuse
 jdbc.url = jdbc:mysql://localhost:3306/reportdb
 jdbc.user = fuse
 jdbc.useSSL = false
 osgi.jdbc.driver.name = mysql
 osgi.jndi.service.name = mysqlds
 pax.jdbc.managed = true
 pool.blockWhenExhausted = true
 pool.maxTotal = 10
 pool.maxWaitMillis = 2000
 pool.minIdle = 2
 pool.testOnBorrow = true
 pool.testWhileIdle = alse
 pool.timeBetweenEvictionRunsMillis = 120000
 service.bundleid = 225
 service.factoryPid = org.ops4j.datasource
 service.id = 338
 service.pid = org.ops4j.datasource.fd7aa3a1-695b-4342-b0d6-23d018a46fbb
 service.scope = singleton
Provided by :
 OPS4J Pax JDBC Config (225)
```

5. データソースを使用します。

```
karaf@root()> jdbc:query mysqlds 'select * from incident'
date                  | summary    | name   | details                    | id | email
----------------------+------------+--------+----------------------------+----+-----------------------
2018-02-20 08:00:00.0 | Incident 1 | User 1 | This is a report incident 001 | 1  |
```

```
user1@redhat.com
2018-02-20 08:10:00.0 | Incident 2 | User 2 | This is a report incident 002 | 2 |
user2@redhat.com
2018-02-20 08:20:00.0 | Incident 3 | User 3 | This is a report incident 003 | 3 |
user3@redhat.com
2018-02-20 08:30:00.0 | Incident 4 | User 4 | This is a report incident 004 | 4 |
user4@redhat.com
```

6.7.3. narayana 接続プールモジュールの使用

pax-jdbc-pool-narayna モジュールはほぼすべてが **pax-jdbc-pool-dbcp2** として行われます。XA および XA 以外のシナリオ向けに DBCP2 固有の **org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory** をインストールします。**唯一** の違いは、XA のシナリオでは追加の統合ポイントがあることです。**org.jboss.tm.XAResourceRecovery** OSGi サービスは、Narayana トランザクションマネージャーに含まれる **com.arjuna.ats.arjuna.recovery.RecoveryManager** で取得されるように登録されます。

6.7.4. transx 接続プールモジュールの使用

pax-jdbc-pool-transx バンドルは、**org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory** サービスの実装を **pax-transx-jdbc** バンドルをもとにします。**pax-transx-jdbc** バンドルは、**org.ops4j.pax.transx.jdbc.ManagedDataSourceBuilder** 機能を使用して **javax.sql.DataSource** プールを作成します。これは JCA(Java™ Connector Architecture) ソリューションで、後 で説明します。

6.8. アーティファクトとしてのデータソースのデプロイ

この章では、OSGi JDBC サービスを紹介し、**pax-jdbc** バンドルがデータベース固有および汎用データソースの登録にどのように役立つのか、および OSGi サービスと設定管理者設定の観点から見たすべての仕組みについて説明します。データソースの両方のカテゴリー の設定は、Configuration Admin ファクトリー PID を使用して行うことができますが (**pax-jdbc-config** バンドルを利用して)、通常は **デプロイメント方法** を使用することが推奨されます。

デプロイメント方法 では、**javax.sql.DataSource** サービスは、通常 Blueprint コンテナー内でアプリケーションコードから直接登録されます。Blueprint XML は、通常の OSGi バンドルの一部で、**mvn:** URI を使用してインストールでき、Maven リポジトリー (ローカルまたはリモート) に保存されます。このようなバンドルを Configuration Admin 設定と比較することで、バージョン管理がはるかに容易になります。

pax-jdbc-config バンドルバージョン 1.3.0 は、データソース 設定の **デプロイメントメソッド**を追加します。アプリケーション開発者は (通常 Bluerpint XML を使用して) **javax.sql.(XA)DataSource** サービスを登録し、サービスプロパティーを指定します。**pax-jdbc-config** バンドルは、このような登録データベース固有のデータソースを検出し (サービスプロパティーを使用)、汎用的なデータベース固有でない接続プール内でサービスをラップします。

完全を期すため、Blueprint XML を使用する 3 つの **デプロイメントメソッド** を以下に示します。Fuse には、Fuse のさまざまな側面の例が含まれる **quickstarts** ダウンロードを提供します。**quickstarts** zip ファイルは Fuse Software Downloads ページからダウンロードできます。

クイックスタート zip ファイルの内容をローカルフォルダーに展開します。

以下の例では、**quickstarts/persistence** ディレクトリーは **$PQ_HOME** と呼ばれます。

- 「データソースの手動デプロイメント」

- 「データソースのファクトリーデプロイメント」
- 「データソースの混合デプロイメント」

6.8.1. データソースの手動デプロイメント

データソースの手動デプロイメントの例では、docker ベースの PostgreSQL インストールを使用します。この方法では、**pax-jdbc-config** は必要ありません。アプリケーションコードは、データベース固有および汎用データソースの両方の登録を行います。

これらの 3 つのバンドルが必要です。

- **mvn:org.postgresql/postgresql/42.2.5**
- **mvn:org.apache.commons/commons-pool2/2.5.0**
- **mvn:org.apache.commons/commons-dbcp2/2.1.1**

```
<!--
   Database-specific, non-pooling, non-enlisting javax.sql.XADataSource
-->
<bean id="postgresql" class="org.postgresql.xa.PGXADataSource">
   <property name="url" value="jdbc:postgresql://localhost:5432/reportdb" />
   <property name="user" value="fuse" />
   <property name="password" value="fuse" />
   <property name="currentSchema" value="report" />
   <property name="connectTimeout" value="5" />
</bean>

<!--
   Fuse/Karaf exports this service from fuse-pax-transx-tm-narayana bundle
-->
<reference id="tm" interface="javax.transaction.TransactionManager" />

<!--
   Non database-specific, generic, pooling, enlisting javax.sql.DataSource
-->
<bean id="pool" class="org.apache.commons.dbcp2.managed.BasicManagedDataSource">
   <property name="xaDataSourceInstance" ref="postgresql" />
   <property name="transactionManager" ref="tm" />
   <property name="minIdle" value="3" />
   <property name="maxTotal" value="10" />
   <property name="validationQuery" value="select schema_name, schema_owner from
information_schema.schemata" />
</bean>

<!--
   Expose datasource to use by application code (like Camel, Spring, ...)
-->
<service interface="javax.sql.DataSource" ref="pool">
   <service-properties>
      <entry key="osgi.jndi.service.name" value="jdbc/postgresql" />
   </service-properties>
</service>
```

上記の Blueprint XML フラグメントは 標準の DataSource 例 と一致します。以下は、その使用方法を示すシェルコマンドです。

```
karaf@root()> install -s mvn:org.postgresql/postgresql/42.2.5
Bundle ID: 233
karaf@root()> install -s mvn:org.apache.commons/commons-pool2/2.5.0
Bundle ID: 224
karaf@root()> install -s mvn:org.apache.commons/commons-dbcp2/2.1.1
Bundle ID: 225
karaf@root()> install -s blueprint:file://$PQ_HOME/databases/blueprints/postgresql-manual.xml
Bundle ID: 226

karaf@root()> bundle:services -p 226

Bundle 226 provides:
--------------------
objectClass = [javax.sql.DataSource]
osgi.jndi.service.name = jdbc/postgresql
osgi.service.blueprint.compname = pool
service.bundleid = 226
service.id = 242
service.scope = bundle
-----
objectClass = [org.osgi.service.blueprint.container.BlueprintContainer]
osgi.blueprint.container.symbolicname = postgresql-manual.xml
osgi.blueprint.container.version = 0.0.0
service.bundleid = 226
service.id = 243
service.scope = singleton

karaf@root()> feature:install jdbc

karaf@root()> jdbc:ds-list
Name            │ Product    │ Version                      │ URL
│ Status
────────────────┼────────────┼──────────────────────────────┼──────
────────────────────────────────────────┼──────
jdbc/postgresql │ PostgreSQL │ 10.3 (Debian 10.3-1.pgdg90+1) │
jdbc:postgresql://localhost:5432/reportdb?
prepareThreshold=5&preparedStatementCacheQueries=256&preparedStatementCacheSizeMiB=5&da
tabaseMetadataCacheFields=65536&databaseMetadataCacheFieldsMiB=5&defaultRowFetchSize=0&b
naryTransfer=true&readOnly=false&binaryTransferEnable=&binaryTransferDisable=&unknownLength=
2147483647&logUnclosedConnections=false&disableColumnSanitiser=false&tcpKeepAlive=false&login
meout=0&connectTimeout=5&socketTimeout=0&cancelSignalTimeout=10&receiveBufferSize=-
```

```
1&sendBufferSize=-1&ApplicationName=PostgreSQL JDBC
Driver&jaasLogin=true&useSpnego=false&gsslib=auto&sspiServiceClass=POSTGRES&allowEncodingC
hanges=false&currentSchema=report&targetServerType=any&loadBalanceHosts=false&hostRecheckSe
conds=10&preferQueryMode=extended&autosave=never&reWriteBatchedInserts=false | OK

karaf@root()> jdbc:query jdbc/postgresql 'select * from incident';
date                | summary    | name   | details                    | id | email
--------------------+------------+--------+----------------------------+----+-------------------
2018-02-20 08:00:00 | Incident 1 | User 1 | This is a report incident 001 | 1  | user1@redhat.com
2018-02-20 08:10:00 | Incident 2 | User 2 | This is a report incident 002 | 2  | user2@redhat.com
2018-02-20 08:20:00 | Incident 3 | User 3 | This is a report incident 003 | 3  | user3@redhat.com
2018-02-20 08:30:00 | Incident 4 | User 4 | This is a report incident 004 | 4  | user4@redhat.com
```

上記の一覧に示すように、Blueprint バンドルは、汎用的でデータベース固有でない接続プールである **javax.sql.DataSource** サービスをエクスポートします。Blueprint XML には明示的な **<service ref="postgresql">** 宣言がないため、データベース固有の **javax.sql.XADataSource** バンドルは OSGi サービスとして登録**されません**。

6.8.2. データソースのファクトリーデプロイメント

データソースのファクトリーデプロイメントは、**標準的** な方法で **pax-jdbc-config** バンドルを使用します。これは、サービスプロパティーとしてプーリング設定を指定する必要があった Fuse6.x で推奨されていた方法とは少し異なります。

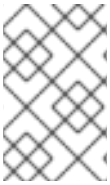
Blueprint XML の例を以下に示します。

```
<!--
   A database-specific org.osgi.service.jdbc.DataSourceFactory that can create
DataSource/XADataSource/
   /ConnectionPoolDataSource/Driver using properties. It is registered by pax-jdbc-* or for example
   mvn:org.postgresql/postgresql/42.2.5 bundle natively.
-->
<reference id="dataSourceFactory"
      interface="org.osgi.service.jdbc.DataSourceFactory"
      filter="(osgi.jdbc.driver.class=org.postgresql.Driver)" />

<!--
   Non database-specific org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory that can create
   pooled data sources using some org.osgi.service.jdbc.DataSourceFactory. dbcp2 pool is registered
   by pax-jdbc-pool-dbcp2 bundle.
-->
<reference id="pooledDataSourceFactory"
      interface="org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory"
      filter="(&amp;(pool=dbcp2)(xa=true))" />

<!--
   Finally, use both factories to expose pooled, xa-aware data source.
-->
<bean id="pool" factory-ref="pooledDataSourceFactory" factory-method="create">
   <argument ref="dataSourceFactory" />
   <argument>
      <props>
         <!--
            Properties needed by postgresql-specific org.osgi.service.jdbc.DataSourceFactory.
```

```
            Cannot prepend them with 'jdbc.' prefix as the DataSourceFactory is implemented directly
            by PostgreSQL driver, not by pax-jdbc-* bundle.
        -->
        <prop key="url" value="jdbc:postgresql://localhost:5432/reportdb" />
        <prop key="user" value="fuse" />
        <prop key="password" value="fuse" />
        <prop key="currentSchema" value="report" />
        <prop key="connectTimeout" value="5" />
        <!-- Properties needed by dbcp2-specific
org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory -->
        <prop key="pool.minIdle" value="2" />
        <prop key="pool.maxTotal" value="10" />
        <prop key="pool.blockWhenExhausted" value="true" />
        <prop key="pool.maxWaitMillis" value="2000" />
        <prop key="pool.testOnBorrow" value="true" />
        <prop key="pool.testWhileIdle" value="false" />
        <prop key="factory.validationQuery" value="select schema_name from
information_schema.schemata" />
        <prop key="factory.validationQueryTimeout" value="2" />
      </props>
   </argument>
</bean>

<!--
   Expose data source for use by application code (such as  Camel, Spring, ...).
-->
<service interface="javax.sql.DataSource" ref="pool">
   <service-properties>
      <entry key="osgi.jndi.service.name" value="jdbc/postgresql" />
   </service-properties>
</service>
```

この例では、データソース **ファクトリー** を使用してデータソースを作成するファクトリー Bean を使用します。XA 対応の **PooledDataSourceFactory** によって内部的に追跡されるため、**javax.transaction.TransactionManager** サービスを明示的に参照する必要はありません。

以下の例は Fuse/Karaf シェルと同じです。



注記

ネイティブの **org.osgi.service.jdbc.DataSourcFactory** バンドルを登録するには、**mvn:org.osgi/org.osgi.service.jdbc/1.0.0** をインストールしてから、PostgreSQL ドライバーをインストールします。

```
karaf@root()> feature:install jdbc pax-jdbc-config pax-jdbc-pool-dbcp2
karaf@root()> install -s mvn:org.postgresql/postgresql/42.2.5
Bundle ID: 232
karaf@root()> install -s blueprint:file://$PQ_HOME/databases/blueprints/postgresql-pax-jdbc-factory-
dbcp2.xml
Bundle ID: 233
karaf@root()> bundle:services -p 233

Bundle 233 provides:
--------------------
objectClass = [javax.sql.DataSource]
```

```
osgi.jndi.service.name = jdbc/postgresql
osgi.service.blueprint.compname = pool
service.bundleid = 233
service.id = 336
service.scope = bundle
-----
objectClass = [org.osgi.service.blueprint.container.BlueprintContainer]
osgi.blueprint.container.symbolicname = postgresql-pax-jdbc-factory-dbcp2.xml
osgi.blueprint.container.version = 0.0.0
service.bundleid = 233
service.id = 337
service.scope = singleton

karaf@root()> jdbc:ds-list
Name            │ Product    │ Version                      │ URL
│ Status
───────────────────────────┼────────────────────┼──────────────────────────────────────────────────┼──
─────────────────────────────────────────────────────────────────────────────────────────────────────
─────────────────────────────────────────────────────────────────────────────────────────────────────
─────────────────────────────────────────────────────────────────────────────────────────────────────
─────────────────────────────────────────────────────────────────────────────────────────────────────
─────────────────────────────────────────────────────────────────────────────────────────────────────
─────────────────────────────────────────────────────────────────────────────────────────────────────
─────────────────────────────────────────────────────────────────────────────────────────────────────
─────────────────────────────────────────────────────────────────────────────────────────────────────
─────────────────────────────────────────────────────────────────────────────────────────────────────
─────────────────────────────────────────────────────────────────────────────────────────────────────
─────────────────────────────────────────────────────────────────────────────────────────────────────
─────────────────────────────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────────┼────────────
jdbc/postgresql │ PostgreSQL │ 10.3 (Debian 10.3-1.pgdg90+1) │
jdbc:postgresql://localhost:5432/reportdb?
prepareThreshold=5&preparedStatementCacheQueries=256&preparedStatementCacheSizeMiB=5&da
tabaseMetadataCacheFields=65536&databaseMetadataCacheFieldsMiB=5&defaultRowFetchSize=0&b
naryTransfer=true&readOnly=false&binaryTransferEnable=&binaryTransferDisable=&unknownLength=
2147483647&logUnclosedConnections=false&disableColumnSanitiser=false&tcpKeepAlive=false&loginT
meout=0&connectTimeout=5&socketTimeout=0&cancelSignalTimeout=10&receiveBufferSize=-
1&sendBufferSize=-1&ApplicationName=PostgreSQL JDBC
Driver&jaasLogin=true&useSpnego=false&gsslib=auto&sspiServiceClass=POSTGRES&allowEncodingC
hanges=false&currentSchema=report&targetServerType=any&loadBalanceHosts=false&hostRecheckSe
conds=10&preferQueryMode=extended&autosave=never&reWriteBatchedInserts=false │ OK

karaf@root()> jdbc:query jdbc/postgresql 'select * from incident';
date                │ summary    │ name   │ details                      │ id │ email
──────────────────────────────────────┼─────────────────────────┼─────────────────┼──────────────────
──────────────────────┼─────────┼───────────────────────────
2018-02-20 08:00:00 │ Incident 1 │ User 1 │ This is a report incident 001 │ 1  │ user1@redhat.com
2018-02-20 08:10:00 │ Incident 2 │ User 2 │ This is a report incident 002 │ 2  │ user2@redhat.com
2018-02-20 08:20:00 │ Incident 3 │ User 3 │ This is a report incident 003 │ 3  │ user3@redhat.com
2018-02-20 08:30:00 │ Incident 4 │ User 4 │ This is a report incident 004 │ 4  │ user4@redhat.com
```

上記の一覧に示すように、Blueprint バンドルは、汎用的でデータベース固有でない接続プールである **javax.sql.DataSource** サービスをエクスポートします。Blueprint XML には明示的な **<service ref="postgresql">** 宣言がないため、データベース固有の **javax.sql.XADataSource** バンドルは OSGi サービスとして登録**されません**。

6.8.3. データソースの混合デプロイメント

データソースの混合デプロイメントでは、**pax-jdbc-config** 1.3.0 バンドルは、サービスプロパティーを使用して、プーリングデータソース内でデータベース固有のデータソースを **ラッピング** する別の方法を追加します。このメソッドは、Fuse 6.x で動作した方法と一致します。

以下は Blueprint XML の例です。

```
<!--
   Database-specific, non-pooling, non-enlisting javax.sql.XADataSource
-->
<bean id="postgresql" class="org.postgresql.xa.PGXADataSource">
   <property name="url" value="jdbc:postgresql://localhost:5432/reportdb" />
   <property name="user" value="fuse" />
   <property name="password" value="fuse" />
   <property name="currentSchema" value="report" />
   <property name="connectTimeout" value="5" />
</bean>

<!--
   Expose database-specific data source with service properties.
   No need to expose pooling, enlisting, non database-specific javax.sql.DataSource. It is registered
   automatically by pax-jdbc-config with the same properties as this <service>, but with higher
service.ranking.
-->
<service id="pool" ref="postgresql" interface="javax.sql.XADataSource">
   <service-properties>
      <!-- "pool" key is needed for pax-jdbc-config to wrap database-specific data source inside
connection pool -->
      <entry key="pool" value="dbcp2" />
      <entry key="osgi.jndi.service.name" value="jdbc/postgresql" />
      <!-- Other properties that configure given connection pool, as indicated by pool=dbcp2 -->
      <entry key="pool.minIdle" value="2" />
      <entry key="pool.maxTotal" value="10" />
      <entry key="pool.blockWhenExhausted" value="true" />
      <entry key="pool.maxWaitMillis" value="2000" />
      <entry key="pool.testOnBorrow" value="true" />
      <entry key="pool.testWhileIdle" value="false" />
      <entry key="factory.validationQuery" value="select schema_name from
information_schema.schemata" />
      <entry key="factory.validationQueryTimeout" value="2" />
   </service-properties>
</service>
```

上記の例では、データベース固有のデータソースのみが手動で登録されています。**pool=dbcp2** サービスプロパティーは、**pax-jdbc-config** バンドルによって管理されるデータソーストラッカーのヒントです。このサービスプロパティーを持つデータソースサービスは、プーリングデータソース内でラップされます (この例では **pax-jdbc-pool-dbcp2**)。

以下は、Fuse/Karaf シェルと同じ例です。

```
karaf@root()> feature:install jdbc pax-jdbc-config pax-jdbc-pool-dbcp2
karaf@root()> install -s mvn:org.postgresql/postgresql/42.2.5
Bundle ID: 232
karaf@root()> install -s blueprint:file://$PQ_HOME/databases/blueprints/postgresql-pax-jdbc-
discovery.xml
```

```
Bundle ID: 233
karaf@root()> bundle:services -p 233

Bundle 233 provides:
--------------------
factory.validationQuery = select schema_name from information_schema.schemata
factory.validationQueryTimeout = 2
objectClass = [javax.sql.XADataSource]
osgi.jndi.service.name = jdbc/postgresql
osgi.service.blueprint.compname = postgresql
pool = dbcp2
pool.blockWhenExhausted = true
pool.maxTotal = 10
pool.maxWaitMillis = 2000
pool.minIdle = 2
pool.testOnBorrow = true
pool.testWhileIdle = false
service.bundleid = 233
service.id = 336
service.scope = bundle
-----
objectClass = [org.osgi.service.blueprint.container.BlueprintContainer]
osgi.blueprint.container.symbolicname = postgresql-pax-jdbc-discovery.xml
osgi.blueprint.container.version = 0.0.0
service.bundleid = 233
service.id = 338
service.scope = singleton

karaf@root()> service:list javax.sql.XADataSource
[javax.sql.XADataSource]
------------------------
 factory.validationQuery = select schema_name from information_schema.schemata
 factory.validationQueryTimeout = 2
 osgi.jndi.service.name = jdbc/postgresql
 osgi.service.blueprint.compname = postgresql
 pool = dbcp2
 pool.blockWhenExhausted = true
 pool.maxTotal = 10
 pool.maxWaitMillis = 2000
 pool.minIdle = 2
 pool.testOnBorrow = true
 pool.testWhileIdle = false
 service.bundleid = 233
 service.id = 336
 service.scope = bundle
Provided by :
 Bundle 233
Used by:
 OPS4J Pax JDBC Config (224)

karaf@root()> service:list javax.sql.DataSource
[javax.sql.DataSource]
----------------------
 factory.validationQuery = select schema_name from information_schema.schemata
 factory.validationQueryTimeout = 2
 osgi.jndi.service.name = jdbc/postgresql
```

```
 osgi.service.blueprint.compname = postgresql
 pax.jdbc.managed = true
 pax.jdbc.service.id.ref = 336
 pool.blockWhenExhausted = true
 pool.maxTotal = 10
 pool.maxWaitMillis = 2000
 pool.minIdle = 2
 pool.testOnBorrow = true
 pool.testWhileIdle = false
 service.bundleid = 224
 service.id = 337
 service.ranking = 1000
 service.scope = singleton
Provided by :
 OPS4J Pax JDBC Config (224)

karaf@root()> jdbc:ds-list
Name            │ Product    │ Version                  │ URL
│ Status
────────────────┼────────────┼──────────────────────────┼──────────────────────────────────────────────────────────────────────────────────────────────┼─────────
jdbc/postgresql │ PostgreSQL │ 10.3 (Debian 10.3-1.pgdg90+1) │
jdbc:postgresql://localhost:5432/reportdb?
prepareThreshold=5&preparedStatementCacheQueries=256&preparedStatementCacheSizeMiB=5&da
tabaseMetadataCacheFields=65536&databaseMetadataCacheFieldsMiB=5&defaultRowFetchSize=0&b
naryTransfer=true&readOnly=false&binaryTransferEnable=&binaryTransferDisable=&unknownLength=
2147483647&logUnclosedConnections=false&disableColumnSanitiser=false&tcpKeepAlive=false&login
meout=0&connectTimeout=5&socketTimeout=0&cancelSignalTimeout=10&receiveBufferSize=-
1&sendBufferSize=-1&ApplicationName=PostgreSQL JDBC
Driver&jaasLogin=true&useSpnego=false&gsslib=auto&sspiServiceClass=POSTGRES&allowEncoding
hanges=false&currentSchema=report&targetServerType=any&loadBalanceHosts=false&hostRecheckS
conds=10&preferQueryMode=extended&autosave=never&reWriteBatchedInserts=false │ OK
jdbc/postgresql │ PostgreSQL │ 10.3 (Debian 10.3-1.pgdg90+1) │
jdbc:postgresql://localhost:5432/reportdb?
prepareThreshold=5&preparedStatementCacheQueries=256&preparedStatementCacheSizeMiB=5&da
tabaseMetadataCacheFields=65536&databaseMetadataCacheFieldsMiB=5&defaultRowFetchSize=0&b
naryTransfer=true&readOnly=false&binaryTransferEnable=&binaryTransferDisable=&unknownLength=
2147483647&logUnclosedConnections=false&disableColumnSanitiser=false&tcpKeepAlive=false&login
meout=0&connectTimeout=5&socketTimeout=0&cancelSignalTimeout=10&receiveBufferSize=-
1&sendBufferSize=-1&ApplicationName=PostgreSQL JDBC
Driver&jaasLogin=true&useSpnego=false&gsslib=auto&sspiServiceClass=POSTGRES&allowEncoding
hanges=false&currentSchema=report&targetServerType=any&loadBalanceHosts=false&hostRecheckS
conds=10&preferQueryMode=extended&autosave=never&reWriteBatchedInserts=false │ OK
```

```
karaf@root()> jdbc:query jdbc/postgresql 'select * from incident'
date                | summary    | name   | details                        | id | email
--------------------+------------+--------+--------------------------------+----+-------------------
2018-02-20 08:00:00 | Incident 1 | User 1 | This is a report incident 001  | 1  | user1@redhat.com
2018-02-20 08:10:00 | Incident 2 | User 2 | This is a report incident 002  | 2  | user2@redhat.com
2018-02-20 08:20:00 | Incident 3 | User 3 | This is a report incident 003  | 3  | user3@redhat.com
2018-02-20 08:30:00 | Incident 4 | User 4 | This is a report incident 004  | 4  | user4@redhat.com
```

このリストには、**jdbc:ds-list** 出力からわかるように、元のデータソースとラッパーデータソースという 2 つのデータソースがあります。

javax.sql.XADataSource は Blueprint バンドルから登録され、 **pool = dbcp2** プロパティーが宣言されています。

javax.sql.DataSource は **pax-jdbc-config** バンドルから登録されます。また、

- **pool = dbcp2** プロパティーがありません (ラッパーデータソースの登録時に削除されました)。
- **service.ranking = 1000** プロパティーがあるので、名前でデータソースを検索する場合などに、常に優先されるバージョンになります。
- **pax.jdbc.managed = true** プロパティーがあるので、再度ラップが試行されていません。
- **pax.jdbc.service.id.ref = 336** プロパティーがあり、接続プール内でラップされる元のデータソースサービスを示します。

6.9. JAVA™ PERSISTENCE API でのデータソースの使用

トランザクション管理の観点から、データソースが Java™Persistence API(JPA) でどのように使用されるかを理解することが重要です。このセクションでは、JPA 仕様自体の詳細や、最もよく知られている JPA 実装である Hibernate の詳細については説明しません。代わりに、このセクションでは、JPA 永続ユニットをデータソースにポイントする方法を示します。

6.9.1. データソースの参照について

META-INF/persistence.xml 記述子 (JPA 2.1 仕様 **8.2.1.5 jta-data-source, non-jta-data-source**を参照) は、2 種類のデータソース参照を定義します。

- **<jta-data-source>** - これは、**JTA** トランザクションで使用する JTA 対応データソースに対する JNDI 参照です。
- **<non-jta-data-source>** - これは、**JTA** トランザクション外で使用する JTA 対応データソースに対する JNDI 参照です。このデータソースは通常、Hibernate がデータベーススキーマを自動作成するように設定する **hibernate.hbm2ddl.auto** プロパティーなどと、初期化フェーズでも使用されます。

これらの 2 つのデータソースは、**javax.sql.DataSource** または **javax.sql.XADataSource** とは関連性が**ありません**。これは、JPA アプリケーションを開発する時によくある誤解です。両方の JNDI 名は JNDI でバインドされた **javax.sql.DataSource** サービスを参照する必要があります。

6.9.2. JNDI 名の参照

OSGi サービスを **osgi.jndi.service.name** プロパティーに登録すると、OSGi JNDI サービスで **バインド**されます。OSGi ランタイム (Fuse/Karaf など) では、JNDI は name → value ペアの単純なディク

ショナリーではありません。OSGi の JNDI 名を使用してオブジェクトを参照するには、サービスルックアップやその他の複雑な OSGi メカニズム (サービスフックなど) が必要です。

Fuse の新規インストールでは、以下のリストはデータソースが JNDI に登録される方法を示します。

```
karaf@root()> install -s mvn:mysql/mysql-connector-java/5.1.34
Bundle ID: 223
karaf@root()> install -s mvn:org.osgi/org.osgi.service.jdbc/1.0.0
Bundle ID: 224
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-mysql/1.3.0
Bundle ID: 225
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc/1.3.0
Bundle ID: 226
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-pool-common/1.3.0
Bundle ID: 227
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-config/1.3.0
Bundle ID: 228

karaf@root()> config:edit --factory --alias mysql org.ops4j.datasource
karaf@root()> config:property-set osgi.jdbc.driver.name mysql
karaf@root()> config:property-set dataSourceName mysqlds
karaf@root()> config:property-set osgi.jndi.service.name jdbc/mysqlds
karaf@root()> config:property-set dataSourceType DataSource
karaf@root()> config:property-set jdbc.url jdbc:mysql://localhost:3306/reportdb
karaf@root()> config:property-set jdbc.user fuse
karaf@root()> config:property-set jdbc.password fuse
karaf@root()> config:property-set jdbc.useSSL false
karaf@root()> config:update

karaf@root()> feature:install jndi

karaf@root()> jndi:names
JNDI Name                │ Class Name
─────────────────────────┼──────────────────────────────────────────────────
─────────────────
osgi:service/jndi        │ org.apache.karaf.jndi.internal.JndiServiceImpl
osgi:service/jdbc/mysqlds │ com.mysql.jdbc.jdbc2.optional.MysqlDataSource
```

ご覧のとおり、データソースは **osgi:service/jdbc/mysqlds** JNDI 名で利用できます。

しかし、OSGi の JPA の場合は、**完全な JNDI 名** を使用する必要があります。以下は、データソース参照を指定する **META-INF/persistence.xml** フラグメントの例です。

```
<jta-data-source>
    osgi:service/javax.sql.DataSource/(osgi.jndi.service.name=jdbc/mysqlds)
</jta-data-source>
<non-jta-data-source>
    osgi:service/javax.sql.DataSource/(osgi.jndi.service.name=jdbc/mysqlds)
</non-jta-data-source>
```

上記の設定がないと、以下のエラーが発生する可能性があります。

```
Persistence unit "pu-name" refers to a non OSGi service DataSource
```

第7章 JMS 接続ファクトリーの作成

本章では、OSGi で JMS 接続ファクトリーを使用する方法を説明します。基本的に、以下を使用して実行します。

```
org.osgi.framework.BundleContext.registerService(javax.jms.ConnectionFactory.class,
                                                 connectionFactoryObject,
                                                 properties);
org.osgi.framework.BundleContext.registerService(javax.jms.XAConnectionFactory.class,
                                                 xaConnectionFactoryObject,
                                                 properties);
```

このようなサービスを登録する方法は 2 つあります。

- **jms:create** Karaf コンソールコマンドを使用して接続ファクトリーを公開します。これは、**設定メソッド** です。
- Blueprint、OSGi Declarative Services (SCR)、または **BundleContext.registerService()** API 呼び出しなどのメソッドを使用して接続ファクトリーをパブリッシュします。この方法では、コードやメタデータを含む専用の OSGi バンドルが必要です。これが、**デプロイメントの方法** です。

詳細は以下を参照してください。

- 「OSGi JMS サービスについて」
- 「PAX-JMS 設定サービス」
- 「JMS コンソールコマンドの使用」
- 「暗号化された設定値の使用」
- 「JMS 接続プールの使用」
- 「接続ファクトリーのアーティファクトとしてのデプロイ」

7.1. OSGI JMS サービスについて

JDBC データソースを処理する OSGi 方法は、次の 2 つのインターフェイスに関連します。

- standard **org.osgi.service.jdbc.DataSourceFactory**
- proprietary **org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory**

JMS の場合、以下の推測を考慮します。

- **プロプライエタリー org.ops4j.pax.jms.service.ConnectionFactoryFactory**: **標準の** OSGi JDBC **org.osgi.service.jdbc.DataSourceFactory** と同じ目的。
- **プロプライエタリー org.ops4j.pax.jms.service.PooledConnectionFactoryFactory**: **プロプライエタリー** pax-jdbc **org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory** と同じ目的。

専用のブローカー固有の **org.ops4j.pax.jms.service.ConnectionFactoryFactory** 実装には、以下のようなバンドルがあります。

- **mvn:org.ops4j.pax.jms/pax-jms-artemis/1.0.0**
- **mvn:org.ops4j.pax.jms/pax-jms-ibmmq/1.0.0**
- **mvn:org.ops4j.pax.jms/pax-jms-activemq/1.0.0**

これらのバンドルは、ブローカー固有の **org.ops4j.pax.jms.service.ConnectionFactoryFactory** サービスを登録します。これは、**javax.jms.ConnectionFactory** と **javax.jms.XAConnectionFactory** など、JMS **ファクトリー** を返すことができます。以下に例を示します。

```
karaf@root()> feature:install pax-jms-artemis

karaf@root()> bundle:services -p org.ops4j.pax.jms.pax-jms-config

OPS4J Pax JMS Config (248) provides:
------------------------------------
objectClass = [org.osgi.service.cm.ManagedServiceFactory]
service.bundleid = 248
service.id = 328
service.pid = org.ops4j.connectionfactory
service.scope = singleton

karaf@root()> bundle:services -p org.ops4j.pax.jms.pax-jms-artemis

OPS4J Pax JMS Artemis Support (247) provides:
---------------------------------------------
objectClass = [org.ops4j.pax.jms.service.ConnectionFactoryFactory]
service.bundleid = 247
service.id = 327
service.scope = singleton
type = artemis
```

7.2. PAX-JMS 設定サービス

mvn:org.ops4j.pax.jms/pax-jms-config/1.0.0 バンドルは、以下の 3 つを行う Managed Service Factory を提供します。

- **org.ops4j.pax.jms.service.ConnectionFactoryFactory** OSGi サービスを追跡し、そのメソッドを呼び出します。

```
public ConnectionFactory createConnectionFactory(Map<String, Object> properties);

public XAConnectionFactory createXAConnectionFactory(Map<String, Object> properties);
```

- **org.ops4j.connectionfactory ファクトリー PID** を追跡し、上記のメソッドに必要なプロパティーを収集します。Configuration Admin サービスで利用可能な方法を使用して **ファクトリー設定** を作成する場合 (たとえば **${karaf.etc}/org.ops4j.connectionfactory-artemis.cfg** ファイルを作成)、最終的な手順を実行して、データベース固有のコネクションファクトリーを公開できます。
- **javax.jms.ConnectionFactory** サービスおよび **javax.jms.XAConnectionFactory** サービスを追跡し、これらをプーリング JMS 接続ファクトリー内にラップします。

詳細は以下を参照してください。

- 「AMQ 7.1 の接続ファクトリーの作成」
- 「IBM MQ 8 または IBM MQ 9 の接続ファクトリーの作成」
- 「処理されたプロパティーの概要」

7.2.1. AMQ 7.1 の接続ファクトリーの作成

以下は、Artemis ブローカーの接続ファクターを作成する方法を順をおって説明した **標準的** なガイドです。

1. **pax-jms-artemis** 機能と **pax-jms-config** 機能を使用して Artemis ドライバーをインストールします。

```
karaf@root()> feature:install pax-jms-artemis

karaf@root()> bundle:services -p org.ops4j.pax.jms.pax-jms-config

OPS4J Pax JMS Config (248) provides:
------------------------------------
objectClass = [org.osgi.service.cm.ManagedServiceFactory]
service.bundleid = 248
service.id = 328
service.pid = org.ops4j.connectionfactory
service.scope = singleton

karaf@root()> bundle:services -p org.ops4j.pax.jms.pax-jms-artemis

OPS4J Pax JMS Artemis Support (247) provides:
---------------------------------------------
objectClass = [org.ops4j.pax.jms.service.ConnectionFactoryFactory]
service.bundleid = 247
service.id = 327
service.scope = singleton
type = artemis
```

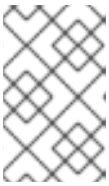
2. **ファクトリー設定** を作成します。

```
karaf@root()> config:edit --factory --alias artemis org.ops4j.connectionfactory
karaf@root()> config:property-set type artemis
karaf@root()> config:property-set osgi.jndi.service.name jms/artemis # "name" property may
be used too
karaf@root()> config:property-set connectionFactoryType ConnectionFactory # or
XAConnectionFactory
karaf@root()> config:property-set jms.url tcp://localhost:61616
karaf@root()> config:property-set jms.user admin
karaf@root()> config:property-set jms.password admin
karaf@root()> config:property-set jms.consumerMaxRate 1234
karaf@root()> config:update

karaf@root()> config:list '(service.factoryPid=org.ops4j.connectionfactory)'
----------------------------------------------------------------
Pid:            org.ops4j.connectionfactory.965d4eac-f5a7-4f65-ba1a-15caa4c72703
FactoryPid:     org.ops4j.connectionfactory
BundleLocation: ?
```

```
Properties:
  connectionFactoryType = ConnectionFactory
  felix.fileinstall.filename = file:${karar.etc}/org.ops4j.connectionfactory-artemis.cfg
  jms.consumerMaxRate = 1234
  jms.password = admin
  jms.url = tcp://localhost:61616
  jms.user = admin
  osgi.jndi.service.name = jms/artemis
  service.factoryPid = org.ops4j.connectionfactory
  service.pid = org.ops4j.connectionfactory.965d4eac-f5a7-4f65-ba1a-15caa4c72703
  type = artemis
```

3. **pax-jms-config** が設定を処理して **javax.jms.ConnectionFactory** サービスに指定しているかどうかを確認します。

```
karaf@root()> service:list javax.jms.ConnectionFactory
[javax.jms.ConnectionFactory]
----------------------------
 connectionFactoryType = ConnectionFactory
 felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.connectionfactory-artemis.cfg
 jms.consumerMaxRate = 1234
 jms.password = admin
 jms.url = tcp://localhost:61616
 jms.user = admin
 osgi.jndi.service.name = jms/artemis
 pax.jms.managed = true
 service.bundleid = 248
 service.factoryPid = org.ops4j.connectionfactory
 service.id = 342
 service.pid = org.ops4j.connectionfactory.965d4eac-f5a7-4f65-ba1a-15caa4c72703
 service.scope = singleton
 type = artemis
Provided by :
 OPS4J Pax JMS Config (248)
```



注記

追加の Artemis 設定 (特に **protocol=amqp**) を指定した場合は、Artemis JMS クライアントの代わりに QPID JMS ライブラリーが使用されます。その後、**amqp://** プロトコルを **jms.url** プロパティーに使用する必要があります。

4. 接続をテストします。

これで、必要に応じて挿入できるブローカー固有の (まだプールなし) 接続ファクトリーができました。たとえば、**jms** 機能から Karaf コマンドを使用できます。

```
karaf@root()> feature:install -v jms
Adding features: jms/[4.2.0.fuse-000237-redhat-1,4.2.0.fuse-000237-redhat-1]
...
karaf@root()> jms:connectionfactories
JMS Connection Factory
------------------------------

jms/artemis
```

```
karaf@root()> jms:info -u admin -p admin jms/artemis
Property │ Value
─────────┼──────────────────────────────────────────
product  │ ActiveMQ
version  │ 2.4.0.amq-711002-redhat-1

karaf@root()> jms:send -u admin -p admin jms/artemis DEV.QUEUE.1 "Hello Artemis"

karaf@root()> jms:browse -u admin -p admin jms/artemis DEV.QUEUE.1
Message ID                          │ Content       │ Charset │ Type │ Correlation ID │ Delivery Mode │
Destination              │ Expiration │ Priority │ Redelivered │ ReplyTo │ Timestamp
────────────────────────────────────┼───────────────┼─────────┼──────┼────────────────┼───────────────
──┼──────────────────────┼────────────┼──────────┼─────────────┼─────────┼──────────────────────────────
──┼──────────────────────┼────────────┼──────────┼─────────────┼─────────┼──────────────────────────────
──────────────────────────
ID:2b6ea56d-574d-11e8-971a-7ee9ecc029d4 │ Hello Artemis │ UTF-8   │      │                │ Persistent
│ ActiveMQQueue[DEV.QUEUE.1] │ Never      │ 4        │ false       │         │ Mon May 14 10:02:38
CEST 2018
```

次のリストは、プロトコルを切り替えたときに何が起こるかを示しています。

```
karaf@root()> config:list '(service.factoryPid=org.ops4j.connectionfactory)'
----------------------------------------------------------------
Pid:            org.ops4j.connectionfactory.965d4eac-f5a7-4f65-ba1a-15caa4c72703
FactoryPid:     org.ops4j.connectionfactory
BundleLocation: ?
Properties:
   connectionFactoryType = ConnectionFactory
   felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.connectionfactory-artemis.cfg
   jms.consumerMaxRate = 1234
   jms.password = fuse
   jms.url = tcp://localhost:61616
   jms.user = fuse
   osgi.jndi.service.name = jms/artemis
   service.factoryPid = org.ops4j.connectionfactory
   service.pid = org.ops4j.connectionfactory.965d4eac-f5a7-4f65-ba1a-15caa4c72703
   type = artemis

karaf@root()> config:edit org.ops4j.connectionfactory.312eb09a-d686-4229-b7e1-2ea38a77bb0f
karaf@root()> config:property-set protocol amqp
karaf@root()> config:property-delete user
karaf@root()> config:property-set username admin # mind the difference between artemis-jms-client
and qpid-jms-client
karaf@root()> config:property-set jms.url amqp://localhost:61616
karaf@root()> config:update

karaf@root()> jms:info -u admin -p admin jms/artemis
Property │ Value
─────────┼──────────────────────────
product  │ QpidJMS
version  │ 0.30.0.redhat-1

karaf@root()> jms:browse -u admin -p admin jms/artemis DEV.QUEUE.1
Message ID │ Content       │ Charset │ Type │ Correlation ID │ Delivery Mode │ Destination │
Expiration │ Priority │ Redelivered │ ReplyTo │ Timestamp
───────────┼───────────────┼─────────┼──────┼────────────────┼───────────────┼─────────────┼
```

```
──────────────┼─────────────┼────────────┼──────────┼──────────
───┼──────────┼──────────────────────────────
   │ Hello Artemis │ UTF-8  │   │        │ Persistent  │ DEV.QUEUE.1 │ Never     │ 4
│ false     │     │ Mon May 14 10:02:38 CEST 2018
```

7.2.2. IBM MQ 8 または IBM MQ 9 の接続ファクトリーの作成

このセクションでは、IBM MQ 8 および IBM MQ 9 に接続する方法を説明します。**pax-jms-ibmmq** は関連する **pax-jms** バンドルをインストールしますが、ライセンス上の理由から IBM MQ ドライバーはインストールされません。

1. https://developer.ibm.com/messaging/mq-downloads/ にアクセスします。
2. ログインします。
3. **IBM MQ 8.0 Client** または **IBM MQ 9.0 Client** など、インストールするバージョンをクリックします。
4. 表示されるページの下部にあるダウンロードバージョンの表で、目的のバージョンをクリックします。
5. 次のページで、接尾辞 **IBM-MQ-Install-Java-All** が付けられた最新バージョンを選択します。たとえば、**8.0.0.10-WS-MQ-Install-Java-All** or **9.0.0.4-IBM-MQ-Install-Java-All** をダウンロードします。
6. ダウンロードした JAR ファイルの内容を展開します。
7. **bundle:install** コマンドを実行します。たとえば、**/home/Downloads** ディレクトリーにコンテンツを展開した場合は、以下のようなコマンドを入力します。

   ```
   `bundle:install -s wrap:file:////home/Downloads/9.0.0.4-IBM-MQ-Install-Java-All/ibmmq9/wmq/JavaSE/com.ibm.mq.allclient.jar`.
   ```

8. 以下のように接続ファクトリーを作成します。
   a. **pax-jms-ibmmq** をインストールします。

      ```
      karaf@root()> feature:install pax-jms-ibmmq

      karaf@root()> bundle:services -p org.ops4j.pax.jms.pax-jms-ibmmq

      OPS4J Pax JMS IBM MQ Support (239) provides:
      --------------------------------------------
      objectClass = [org.ops4j.pax.jms.service.ConnectionFactoryFactory]
      service.bundleid = 239
      service.id = 346
      service.scope = singleton
      type = ibmmq
      ```

   b. **ファクトリー設定** を作成します。

      ```
      karaf@root()> config:edit --factory --alias ibmmq org.ops4j.connectionfactory
      karaf@root()> config:property-set type ibmmq
      karaf@root()> config:property-set osgi.jndi.service.name jms/mq9 # "name" property may be used too
      ```

```
karaf@root()> config:property-set connectionFactoryType ConnectionFactory # or
XAConnectionFactory
karaf@root()> config:property-set jms.queueManager FUSEQM
karaf@root()> config:property-set jms.hostName localhost
karaf@root()> config:property-set jms.port 1414
karaf@root()> config:property-set jms.transportType 1 #
com.ibm.msg.client.wmq.WMQConstants.WMQ_CM_CLIENT
karaf@root()> config:property-set jms.channel DEV.APP.SVRCONN
karaf@root()> config:property-set jms.CCSID 1208 #
com.ibm.msg.client.jms.JmsConstants.CCSID_UTF8
karaf@root()> config:update

karaf@root()> config:list '(service.factoryPid=org.ops4j.connectionfactory)'
----------------------------------------------------------------
Pid:            org.ops4j.connectionfactory.eee4a757-a95d-46b8-b8b6-19aa3977d863
FactoryPid:     org.ops4j.connectionfactory
BundleLocation: ?
Properties:
   connectionFactoryType = ConnectionFactory
   felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.connectionfactory-ibmmq.cfg
   jms.CCSID = 1208
   jms.channel = DEV.APP.SVRCONN
   jms.hostName = localhost
   jms.port = 1414
   jms.queueManager = FUSEQM
   jms.transportType = 1
   osgi.jndi.service.name = jms/mq9
   service.factoryPid = org.ops4j.connectionfactory
   service.pid = org.ops4j.connectionfactory.eee4a757-a95d-46b8-b8b6-19aa3977d863
   type = ibmmq
```

c. **pax-jms-config** が設定を処理して **javax.jms.ConnectionFactory** サービスに指定しているかどうかを確認します。

```
karaf@root()> service:list javax.jms.ConnectionFactory
[javax.jms.ConnectionFactory]
------------------------------
 connectionFactoryType = ConnectionFactory
 felix.fileinstall.filename = file:/data/servers/7.10.0.fuse-7_10_0-00010-redhat-
00001/etc/org.ops4j.connectionfactory-ibmmq.cfg
 jms.CCSID = 1208
 jms.channel = DEV.APP.SVRCONN
 jms.hostName = localhost
 jms.port = 1414
 jms.queueManager = FUSEQM
 jms.transportType = 1
 osgi.jndi.service.name = jms/mq9
 pax.jms.managed = true
 service.bundleid = 237
 service.factoryPid = org.ops4j.connectionfactory
 service.id = 347
 service.pid = org.ops4j.connectionfactory.eee4a757-a95d-46b8-b8b6-19aa3977d863
 service.scope = singleton
 type = ibmmq
Provided by :
 OPS4J Pax JMS Config (237)
```

d. 接続をテストします。

```
karaf@root()> feature:install -v jms
Adding features: jms/[4.2.0.fuse-000237-redhat-1,4.2.0.fuse-000237-redhat-1]
...
karaf@root()> jms:connectionfactories
JMS Connection Factory
------------------------------
jms/mq9

karaf@root()> jms:info -u app -p fuse jms/mq9
Property | Value
---------+-----------------------------
product  | IBM MQ JMS Provider
version  | 8.0.0.0

karaf@root()> jms:send -u app -p fuse jms/mq9 DEV.QUEUE.1 "Hello IBM MQ 9"

karaf@root()> jms:browse -u app -p fuse jms/mq9 DEV.QUEUE.1
Message ID                                | Content          | Charset | Type |
Correlation ID | Delivery Mode | Destination       | Expiration | Priority | Redelivered
| ReplyTo | Timestamp
------------------------------------------+------------------+---------+------+
----------------+---------------+-------------------+------------+----------+------------
---+---------+----------------------------
ID:414d512046555345514d202020202020c940f95a038b3220 | Hello IBM MQ 9
| UTF-8   |      |                | Persistent    | queue:///DEV.QUEUE.1 | Never      | 4
| false       |         | Mon May 14 10:17:01 CEST 2018
```

メッセージが IBM MQ Explorer から送信されたのか、Web コンソールから送信されたのかを確認することもできます。

7.2.3. Fuse on Apache Karaf での JBossA-MQ6.3 クライアントの使用

Fuse の Software Downloads ページから Fuse **quickstarts** をダウンロードできます。

クイックスタート zip ファイルの内容をローカルフォルダーに展開します (例: **quickstarts** という名前のフォルダー)。

次に、**quickstarts/camel/camel-jms** の例を OSGi バンドルとしてビルドしてインストールできます。このバンドルには、メッセージを JBoss A-MQ 6.3 JMS キューに送信する Camel ルートの Blueprint XML 定義が含まれます。JBoss A-MQ 6.3 ブローカーの接続ファクトリーを作成する手順は次のとおりです。

7.2.3.1. 前提条件

- Maven 3.3.1 以降がインストールされている。
- Red Hat Fuse がマシンにインストールされている。
- JBoss A-MQ Broker 6.3 がマシンにインストールされている。

- カスタマーポータル から Fuse on Karaf クイックスタートの zip ファイルをダウンロードして展開している。

7.2.3.2. 手順

1. **quickstarts/camel/camel-jms/src/main/resources/OSGI-INF/blueprint/** ディレクトリーに移動します。

2. **camel-context.xml** ファイルで id="jms" のある以下の Bean を見つけます。

```
<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
  <property name="connectionFactory">
    <reference interface="javax.jms.ConnectionFactory" />
  </property>
  <property name="transactionManager" ref="transactionManager"/>
</bean>
```

   以下のセクションを使用して、JBoss A-MQ 6.3 接続ファクトリーをインスタンス化します。

```
<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
    <property name="connectionFactory" ref="activemqConnectionFactory"/>
    <property name="transactionManager" ref="transactionManager"/>
</bean>
<bean id="activemqConnectionFactory"
class="org.apache.activemq.ActiveMQConnectionFactory">
 <property name="brokerURL" value="tcp://localhost:61616"/>
 <property name="userName" value="admin"/>
 <property name="password" value="admin"/>
</bean>
```

   JBoss A-MQ 6.3 接続ファクトリーは、**tcp://localhost:61616** でリッスンするブローカーに接続するよう設定されます。デフォルトでは、JBoss A-MQ は IP ポート値 **61616** を使用します。接続ファクトリーはユーザー名/パスワードの認証情報で admin/admin を使用するよう設定されています。このユーザーがブローカーの設定で有効にされていることを確認します (または、ここでこれらの設定をブローカー設定に合わせてカスタマイズすることもできます)。

3. **camel-context.xml** ファイルを保存します。

4. **camel-jms** クイックスタートをビルドします。

```
$ cd quickstarts/camel/camel-jms
$ mvn install
```

5. クイックスタートが正常にインストールされた後、**$FUSE_HOME/** ディレクトリーに移動し、以下のコマンドを実行して Fuse on Apache Karaf サーバーを起動します。

```
$ ./bin/fuse
```

6. Fuse on Apache Karaf インスタンスで、**activemq-client** 機能と **camel-jms** 機能をインストールします。

```
karaf@root()> feature:install activemq-client
karaf@root()> feature:install camel-jms
```

7. **camel-jms** クイックスタートバンドルをインストールします。

```
karaf@root()> install -s mvn:org.jboss.fuse.quickstarts/camel-jms/{$fuseversion}
```

{$fuseversion} は、ビルドした Maven アーティファクトの実際のバージョンに置き換えます (camel-jms クイックスタートの README ファイルを参照)。

8. **JBoss A-MQ 6.3** ブローカーを起動します (これに JBoss A-MQ 6.3 のインストールが必要です)。別のターミナルウィンドウを開き、JBOSS_AMQ_63_INSTALLDIR ディレクトリーに移動します。

```
$ cd JBOSS_AMQ_63_INSTALLDIR
$ ./bin/amq
```

9. Camel ルートが起動すると、即座に **work/jms/input** ディレクトリーが Fuse インストールに表示されます。このクイックスタートの **src/main/data** ディレクトリーにあるファイルを新しく作成した **work/jms/input** ディレクトリーにコピーします。

10. しばらく待つと、**work/jms/output** ディレクトリー以下に同じファイルが国別に分類されます。

```
order1.xml, order2.xml and order4.xml in work/jms/output/others
order3.xml and order5.xml in work/jms/output/us
order6.xml in work/jms/output/fr
```

11. **log:display** を使用してビジネスロギングを確認します。

```
Receiving order order1.xml

Sending order order1.xml to another country

Done processing order1.xml
```

7.2.4. 処理されたプロパティーの概要

Configuration Admin **ファクトリー PID** からのプロパティーは、関連する **org.ops4j.pax.jms.service.ConnectionFactoryFactory** 実装に渡されます。

- ActiveMQ
  org.ops4j.pax.jms.activemq.ActiveMQConnectionFactoryFactory (JMS 1.1 のみ)

  org.apache.activemq.ActiveMQConnectionFactory.buildFromMap() メソッドに渡されるプロパティー

- Artemis
  org.ops4j.pax.jms.artemis.ArtemisConnectionFactoryFactory

  protocol=amqp の場合、プロパティーは **org.apache.qpid.jms.util.PropertyUtil.setProperties()** メソッドに渡され、**org.apache.qpid.jms.JmsConnectionFactory** インスタンスを設定します。

  そうでない場合は、**org.apache.activemq.artemis.utils.uri.BeanSupport.setData()** が **org.apache.activemq.artemis.jms.client.ActiveMQConnectionFactory** インスタンスに対して呼び出されます。

- IBM MQ
  org.ops4j.pax.jms.ibmmq.MQConnectionFactoryFactory

  com.ibm.mq.jms.MQConnectionFactory または
  com.ibm.mq.jms.MQXAConnectionFactory の Bean プロパティーが処理されます。

7.3. JMS コンソールコマンドの使用

Apache Karaf は、**jms:*** スコープのシェルコマンドが含まれる **jms** 機能を提供します。これらのコマンドを使用して手動で設定された接続ファクトリーをチェックする例をすでにいくつか見てきました。Configuration Admin 設定を作成する必要性を隠すコマンドもあります。

Fuse の新しいインスタンス以降、ブローカー固有の接続ファクトリーを登録できます。以下のリストは、Karaf からの **jms** 機能のインストールと、**pax-jms** からの **pax-jms-artemis** のインストールを示しています。

```
karaf@root()> feature:install jms pax-jms-artemis

karaf@root()> jms:connectionfactories
JMS Connection Factory
──────────────────────
karaf@root()> service:list javax.jms.ConnectionFactory # should be empty

karaf@root()> service:list org.ops4j.pax.jms.service.ConnectionFactoryFactory
[org.ops4j.pax.jms.service.ConnectionFactoryFactory]
----------------------------------------------------
 service.bundleid = 250
 service.id = 326
 service.scope = singleton
 type = artemis
Provided by :
 OPS4J Pax JMS Artemis Support (250)
```

以下は、Artemis 接続ファクトリーを作成して確認する方法を示しています。

```
karaf@root()> jms:create -t artemis -u admin -p admin --url tcp://localhost:61616 artemis

karaf@root()> jms:connectionfactories
JMS Connection Factory
──────────────────────
jms/artemis

karaf@root()> jms:info -u admin -p admin jms/artemis
Property │ Value
─────────┼───────────────────────────
product  │ ActiveMQ
version  │ 2.4.0.amq-711002-redhat-1

karaf@root()> jms:send -u admin -p admin jms/artemis DEV.QUEUE.1 "Hello Artemis"

karaf@root()> jms:browse -u admin -p admin jms/artemis DEV.QUEUE.1
Message ID                          │ Content       │ Charset │ Type │ Correlation ID │ Delivery Mode │
Destination              │ Expiration │ Priority │ Redelivered │ ReplyTo │ Timestamp
──────────────────────────────────────────────────────────────────┼─────────────────────────┼──────────
───┼──────────┼───────────────────────────┼──────────────────────────────────
```

```
──────────┼──────────┼──────────┼──────────┼──────────┼──────────
──────────────────────────

ID:7a944470-574f-11e8-918e-7ee9ecc029d4 │ Hello Artemis │ UTF-8  │      │             │ Persistent
│ ActiveMQQueue[DEV.QUEUE.1] │ Never      │ 4       │ false      │         │ Mon May 14 10:19:10
CEST 2018

karaf@root()> config:list '(service.factoryPid=org.ops4j.connectionfactory)'
----------------------------------------------------------------
Pid:            org.ops4j.connectionfactory.9184db6f-cb5f-4fd7-b5d7-a217090473ad
FactoryPid:     org.ops4j.connectionfactory
BundleLocation: mvn:org.ops4j.pax.jms/pax-jms-config/1.0.0
Properties:
   name = artemis
   osgi.jndi.service.name = jms/artemis
   password = admin
   service.factoryPid = org.ops4j.connectionfactory
   service.pid = org.ops4j.connectionfactory.9184db6f-cb5f-4fd7-b5d7-a217090473ad
   type = artemis
   url = tcp://localhost:61616
   user = admin
```

ご覧のとおり、**org.ops4j.connectionfactory** ファクトリー PID が作成されます。ただし、**config:update** では可能ですが、これは自動的に **${karaf.etc}** に保存されません。他のプロパティーを指定することはできませんが、後で追加することもできます。

7.4. 暗号化された設定値の使用

pax-jdbc-config バンドルと同様に、Jacsypt を使用してプロパティーを暗号化できます。

OSGi に **alias** サービスプロパティーで登録される **org.jasypt.encryption.StringEncryptor** サービスがある場合は、接続 ファクトリーの**ファクトリー PID** で参照し、暗号化されたパスワードを使用できます。以下に例を示します。

```
felix.fileinstall.filename = */etc/org.ops4j.connectionfactory-artemis.cfg
name = artemis
type = artemis
decryptor = my-jasypt-decryptor
url = tcp://localhost:61616
user = fuse
password = ENC(<encrypted-password>)
```

復号化サービスの検索に使用するサービスフィルターは **(&(objectClass=org.jasypt.encryption.StringEncryptor)(alias=<alias>))** です。**<alias>** は、接続ファクトリー設定の **ファクトリー PID** からの **decryptor** プロパティーの値になります。

7.5. JMS 接続プールの使用

本セクションでは、JMS 接続/セッションプーリングオプションを説明します。JDBC の場合よりも選択肢が少なくなります。情報は次のトピックに分けてまとめられています。

- 「JMS 接続プールの使用」
- 「pax-jms-pool-pooledjms 接続プールモジュールの使用」
- 「pax-jms-pool-narayana 接続プールモジュールの使用」

- 「pax-jms-pool-transx 接続プールモジュールの使用」

重要

XA リカバリーを使用するには、**pax-jms-pool-transx** または **pax-jms-pool-narayana** 接続プールモジュールを使用する必要があります。

7.5.1. JMS 接続プールの使用

これまで、ブローカー固有の接続 **ファクトリー** を登録しました。接続ファクトリー自体は **接続ファクトリー** のファクトリーであるため、**org.ops4j.pax.jms.service.ConnectionFactoryFactory** サービスは**メタファクトリー**として扱われる可能性があります。2 種類の接続ファクトリーを生成できるはずです。

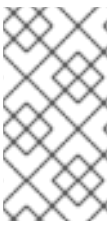
- **javax.jms.ConnectionFactory**
- **javax.jms.XAConnectionFactory**

pax-jms-pool-* バンドルは、**org.ops4j.pax.jms.service.ConnectionFactoryFactory** サービスとスムーズに連携します。これらのバンドルは、**org.ops4j.pax.jms.service.PooledConnectionFactoryFactory** の実装を提供し、これで、プロパティーと、元の **org.ops4j.pax.jms.service.ConnectionFactoryFactory** を wrapper のように使用して、プール接続ファクトリーを作成できます。以下に例を示します。

```
public interface PooledConnectionFactoryFactory {

    ConnectionFactory create(ConnectionFactoryFactory cff, Map<String, Object> props);

}
```

以下の表は、プールされた接続ファクトリーファクトリーを登録するバンドルを示しています。この表では、**o.o.p.j.p** は **org.ops4j.pax.jms.pool** を表します。

バンドル	PooledConnectionFactoryFactory	プールキー
pax-jms-pool-pooledjms	**o.o.p.j.p.pooledjms.PooledJms(XA) PooledConnectionFactoryFactory**	**pooledjms**
pax-jms-pool-narayana	**o.o.p.j.p.narayana.PooledJms(XA)P ooledConnectionFactoryFactory**	**narayana**
pax-jms-pool-transx	**o.o.p.j.p.transx.Transx(XA)PooledC onnectionFactoryFactory**	**transx**



注記

pax-jms-pool-narayana ファクトリーは **PooledJms(XA)PooledConnectionFactoryFactory** と呼ばれ、**pooled-jms** ライブラリーに基づいています。これにより、XA リカバリーのための Narayana トランザクションマネージャーとの統合が追加されます。

上記のバンドルは接続ファクトリーファクトリーのみをインストールします。接続ファクトリー自体を

インストールしないバンドルです。そのため、**javax.jms.ConnectionFactory** **org.ops4j.pax.jms.service.PooledConnectionFactoryFactory.create()** メソッドを呼び出すものが必要になります。

7.5.2. pax-jms-pool-pooledjms 接続プールモジュールの使用

pax-jms-pool-pooledjms バンドルの使用方法を理解することで、**pax-jms-pool-pooledjms** バンドルだけでなく、**pax-jms-pool-narayna** バンドルも使用できるので、**pax-jms-pool-pool-pooledjms** としてほぼすべてが機能します。

pax-jms-config バンドルは以下を追跡します。

- **org.ops4j.pax.jms.service.ConnectionFactoryFactory** services
- **org.ops4j.connectionfactory** factory PIDs
- **pax-jms-pool-*** バンドルのいずれかによって登録される **org.ops4j.pax.jms.service.PooledConnectionFactoryFactory** のインスタンス。

ファクトリー設定 に **pool** プロパティーが含まれる場合、**pax-jms-config** バンドルによって登録される最終的な接続ファクトリーはブローカー固有の接続ファクトリーです。**pool=pooledjms** の場合、接続ファクトリーは以下のいずれかの内部でラップされます。

- **org.messaginghub.pooled.jms.JmsPoolConnectionFactory** (**xa=false**)
- **org.messaginghub.pooled.jms.JmsPoolXAConnectionFactory** (**xa=true**)

pool プロパティー (および非 xa/xa 接続ファクトリーの 1 つを選択するブール値 **xa** プロパティー) のほかに、**org.ops4j.connectionfactory ファクトリー** PID には **pool.** で始まるプロパティーが含まれる場合があります。

pooled-jms ライブラリーでは、これらの接頭辞が付いたプロパティーを使用して (接頭辞を削除した後に)、以下のインスタンスを設定します。

- **org.messaginghub.pooled.jms.JmsPoolConnectionFactory** または
- **org.messaginghub.pooled.jms.JmsPoolXAConnectionFactory**

以下のリストは、**jms.** で始まるプロパティーと便利な構文を使用する **pooled-jms** プール (**org.ops4j.connectionfactory-artemis ファクトリー** PID) の現実的な設定です。

```
# configuration for pax-jms-config to choose and configure specific
org.ops4j.pax.jms.service.ConnectionFactoryFactory
name = jms/artemis
connectionFactoryType = ConnectionFactory
jms.url = tcp://localhost:61616
jms.user = fuse
jms.password = fuse
# org.apache.activemq.artemis.jms.client.ActiveMQConnectionFactory specific coniguration
jms.callTimeout = 12000
# ...

# hints for pax-jms-config to use selected org.ops4j.pax.jms.service.PooledConnectionFactoryFactory
pool = pooledjms
xa = false
```

```
# pooled-jms specific configuration of org.messaginghub.pooled.jms.JmsPoolConnectionFactory
pool.idleTimeout = 10
pool.maxConnections = 100
pool.blockIfSessionPoolIsFull = true
# ...
```

上記の設定では、**pool** と **xa** キーは **ヒント** (サービスフィルタープロパティー) で、登録された **org.ops4j.pax.jms.service.PooledConnectionFactoryFactory** サービスのいずれかを選択します。**pooled-jms** ライブラリーの場合、以下が行われます。

```
karaf@root()> feature:install pax-jms-pool-pooledjms

karaf@root()> bundle:services -p org.ops4j.pax.jms.pax-jms-pool-pooledjms

OPS4J Pax JMS MessagingHub JMS Pool implementation (252) provides:
----------------------------------------------------------------
objectClass = [org.ops4j.pax.jms.service.PooledConnectionFactoryFactory]
pool = pooledjms
service.bundleid = 252
service.id = 331
service.scope = singleton
xa = false
-----
objectClass = [org.ops4j.pax.jms.service.PooledConnectionFactoryFactory]
pool = pooledjms
service.bundleid = 252
service.id = 335
service.scope = singleton
xa = true
```

以下は、接続プールを作成して設定する手順の完全な例です。

1. 必要な機能をインストールします。

```
karaf@root()> feature:install -v pax-jms-pool-pooledjms pax-jms-artemis
Adding features: pax-jms-pool-pooledjms/[1.0.0,1.0.0]
...
```

2. **jms** 機能をインストールします。

```
karaf@root()> feature:install jms

karaf@root()> service:list org.ops4j.pax.jms.service.ConnectionFactoryFactory
[org.ops4j.pax.jms.service.ConnectionFactoryFactory]
---------------------------------------------------
 service.bundleid = 249
 service.id = 327
 service.scope = singleton
 type = artemis
Provided by :
 OPS4J Pax JMS Artemis Support (249)

karaf@root()> service:list org.ops4j.pax.jms.service.PooledConnectionFactoryFactory
[org.ops4j.pax.jms.service.PooledConnectionFactoryFactory]
----------------------------------------------------------
```

```
 pool = pooledjms
 service.bundleid = 251
 service.id = 328
 service.scope = singleton
 xa = false
Provided by :
 OPS4J Pax JMS MessagingHub JMS Pool implementation (251)

[org.ops4j.pax.jms.service.PooledConnectionFactoryFactory]
----------------------------------------------------------
 pool = pooledjms
 service.bundleid = 251
 service.id = 333
 service.scope = singleton
 xa = true
Provided by :
 OPS4J Pax JMS MessagingHub JMS Pool implementation (251)
```

3. **ファクトリー設定** を作成します。

```
karaf@root()> config:edit --factory --alias artemis org.ops4j.connectionfactory
karaf@root()> config:property-set connectionFactoryType ConnectionFactory
karaf@root()> config:property-set osgi.jndi.service.name jms/artemis
karaf@root()> config:property-set type artemis
karaf@root()> config:property-set protocol amqp # so we switch to
org.apache.qpid.jms.JmsConnectionFactory
karaf@root()> config:property-set jms.url amqp://localhost:61616
karaf@root()> config:property-set jms.username admin
karaf@root()> config:property-set jms.password admin
karaf@root()> config:property-set pool pooledjms
karaf@root()> config:property-set xa false
karaf@root()> config:property-set pool.idleTimeout 10
karaf@root()> config:property-set pool.maxConnections 123
karaf@root()> config:property-set pool.blockIfSessionPoolIsFull true
karaf@root()> config:update
```

4. **pax-jms-config** が設定を処理して **javax.jms.ConnectionFactory** サービスに指定しているかどうかを確認します。

```
karaf@root()> service:list javax.jms.ConnectionFactory
[javax.jms.ConnectionFactory]
-----------------------------
 connectionFactoryType = ConnectionFactory
 felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.connectionfactory-artemis.cfg
 jms.password = admin
 jms.url = amqp://localhost:61616
 jms.username = admin
 osgi.jndi.service.name = jms/artemis
 pax.jms.managed = true
 pool.blockIfSessionPoolIsFull = true
 pool.idleTimeout = 10
 pool.maxConnections = 123
 protocol = amqp
 service.bundleid = 250
 service.factoryPid = org.ops4j.connectionfactory
```

```
 service.id = 347
 service.pid = org.ops4j.connectionfactory.fc1b9e85-91b4-421b-aa16-1151b0f836f9
 service.scope = singleton
 type = artemis
Provided by :
 OPS4J Pax JMS Config (250)
```

5. 接続ファクトリーを使用します。

```
karaf@root()> jms:connectionfactories
JMS Connection Factory
───────────────────────
jms/artemis

karaf@root()> jms:info -u admin -p admin jms/artemis
Property │ Value
─────────┼──────────────
product  │ QpidJMS
version  │ 0.30.0.redhat-1

karaf@root()> jms:send -u admin -p admin jms/artemis DEV.QUEUE.1 "Hello Artemis"

karaf@root()> jms:browse -u admin -p admin jms/artemis DEV.QUEUE.1
Message ID                              │ Content       │ Charset │ Type │ Correlation ID │
Delivery Mode │ Destination │ Expiration │ Priority │ Redelivered │ ReplyTo │ Timestamp
────────────────────────────────────────┼───────────────┼─────────┼──────┼────────────────┼
──────────────┼─────────────┼────────────┼──────────┼─────────────┼─────────┼────────────
────────────────────────────────────
ID:64842f99-5cb2-4850-9e88-f50506d49d20:1:1:1-1 │ Hello Artemis │ UTF-8 │      │
│ Persistent    │ DEV.QUEUE.1 │ Never      │ 4        │ false       │         │ Mon May 14
12:47:13 CEST 2018
```

7.5.3. pax-jms-pool-narayana 接続プールモジュールの使用

pax-jms-pool-narayna モジュールはほぼすべてが **pax-jms-pool-pooledjms** として行われます。XA と XA 以外のシナリオ向けに、pooled-jms 固有の **org.ops4j.pax.jms.service.PooledConnectionFactoryFactory** をインストールします。**唯一** の違いは XA シナリオでは、追加の統合ポイントがあることです。**org.jboss.tm.XAResourceRecovery** OSGi サービスは、**com.arjuna.ats.arjuna.recovery.RecoveryManager** によって取得されるように登録されます。

7.5.4. pax-jms-pool-transx 接続プールモジュールの使用

pax-jms-pool-transx モジュールは、**pax-transx-jms** バンドルに基づく **org.ops4j.pax.jms.service.PooledConnectionFactoryFactory** サービスの実装を提供します。**pax-transx-jms** バンドルは、**org.ops4j.pax.transx.jms.ManagedConnectionFactoryBuilder** 機能を使用して **javax.jms.ConnectionFactory** プールを作成します。これは、「pax-transx プロジェクト」 で説明されている JCA(Java™ Connector Architecture) ソリューションです。

7.6. 接続ファクトリーのアーティファクトとしてのデプロイ

このトピックでは、実際の推奨事項について説明します。

デプロイメント方法 では、**javax.jms.ConnectionFactory** サービスはアプリケーションコードによって直接登録されます。通常、このコードは Blueprint コンテナー内にあります。Blueprint XML は、通常の OSGi バンドルの一部である可能性があり、**mvn:** URI を使用してインストールでき、Maven リポジトリー (ローカルまたはリモート) に保存されます。Configuration Admin 設定と比較して、このようなバンドルのバージョン管理は簡単です。

pax-jms-config バージョン 1.0.0 バンドルは、接続ファクトリー設定の **デプロイメント方法** を追加します。アプリケーション開発者は (通常 Bluerpint XML を使用して) **javax.jms.(XA)ConnectionFactory** サービスを登録し、サービスプロパティーを指定します。次に、**pax-jms-config** は、登録されたブローカー固有の接続ファクトリーを検出し、(サービスプロパティーを使用して) 汎用のブローカー固有でない接続プール内でサービスをラップします。

以下は、Blueprint XML を使用する 3 つの **デプロイメント方法** です。

- 「接続ファクトリーの手動デプロイメント」
- 「接続ファクトリーのファクトリーデプロイメント」
- 「接続ファクトリーの混合デプロイメント」

7.6.1. 接続ファクトリーの手動デプロイメント

この方法では、**pax-jms-config** バンドルは必要ありません。アプリケーションコードは、ブローカー固有および汎用接続プールの両方の登録を行います。

```
<!--
   Broker-specific, non-pooling, non-enlisting javax.jms.XAConnectionFactory
-->
<bean id="artemis" class="org.apache.activemq.artemis.jms.client.ActiveMQXAConnectionFactory">
   <argument value="tcp://localhost:61616" />
   <property name="callTimeout" value="2000" />
   <property name="initialConnectAttempts" value="3" />
</bean>

<!--
   Fuse exports this service from fuse-pax-transx-tm-narayana bundle.
-->
<reference id="tm" interface="javax.transaction.TransactionManager" />

<!--
   Non broker-specific, generic, pooling, enlisting javax.jms.ConnectionFactory
-->
<bean id="pool" class="org.messaginghub.pooled.jms.JmsPoolXAConnectionFactory">
   <property name="connectionFactory" ref="artemis" />
   <property name="transactionManager" ref="tm" />
   <property name="maxConnections" value="10" />
   <property name="idleTimeout" value="10000" />
</bean>

<!--
   Expose connection factory for use by application code (such as Camel, Spring, ...)
-->
<service interface="javax.jms.ConnectionFactory" ref="pool">
   <service-properties>
      <!-- Giving connection factory a name using one of these properties makes identification easier
in jms:connectionfactories: -->
```

```
        <entry key="osgi.jndi.service.name" value="jms/artemis" />
        <!--<entry key="name" value="jms/artemis" />-->
        <!-- Without any of the above, name will fall back to "service.id" -->
    </service-properties>
</service>
```

以下は、その使用方法を示すシェルコマンドです。

```
karaf@root()> feature:install artemis-core-client artemis-jms-client
karaf@root()> install -s mvn:org.apache.commons/commons-pool2/2.5.0
Bundle ID: 244
karaf@root()> install -s mvn:org.messaginghub/pooled-jms/0.3.0
Bundle ID: 245
karaf@root()> install -s blueprint:file://$PQ_HOME/message-brokers/blueprints/artemis-manual.xml
Bundle ID: 246

karaf@root()> bundle:services -p 246

Bundle 246 provides:
--------------------
objectClass = [javax.jms.ConnectionFactory]
osgi.jndi.service.name = jms/artemis
osgi.service.blueprint.compname = pool
service.bundleid = 246
service.id = 340
service.scope = bundle
-----
objectClass = [org.osgi.service.blueprint.container.BlueprintContainer]
osgi.blueprint.container.symbolicname = artemis-manual.xml
osgi.blueprint.container.version = 0.0.0
service.bundleid = 246
service.id = 341
service.scope = singleton

karaf@root()> feature:install jms

karaf@root()> jms:connectionfactories
JMS Connection Factory
------------------------------------
jms/artemis

karaf@root()> jms:info -u admin -p admin jms/artemis
Property │ Value
─────────┼──────────────────────────────────────────────
product  │ ActiveMQ
version  │ 2.4.0.amq-711002-redhat-1
```

上記の一覧に示すように、Blueprint バンドルは、汎用的でブローカー固有でない接続プールである **javax.jms.ConnectionFactory** サービスをエクスポートします。Blueprint XML には明示的な **<service ref="artemis">** 宣言がないため、ブローカー固有の **javax.jms.XAConnectionFactory** は OSGi サービスとして登録**されません**。

7.6.2. 接続ファクトリーのファクトリーデプロイメント

このメソッドは、**標準** の方法での **pax-jms-config** の使用方法を示しています。これは、Fuse 6.x で推奨されていた方法とは少し異なります。この方法では、サービスプロパティーとしてプーリング設定を指定する必要がありました。

Blueprint XML の例を以下に示します。

```
<!--
    A broker-specific org.ops4j.pax.jms.service.ConnectionFactoryFactory that can create
(XA)ConnectionFactory
    using properties. It is registered by pax-jms-* bundles
-->
<reference id="connectionFactoryFactory"
        interface="org.ops4j.pax.jms.service.ConnectionFactoryFactory"
        filter="(type=artemis)" />

<!--
    Non broker-specific org.ops4j.pax.jms.service.PooledConnectionFactoryFactory that can create
    pooled connection factories with the help of org.ops4j.pax.jms.service.ConnectionFactoryFactory

    For example, pax-jms-pool-pooledjms bundle registers
org.ops4j.pax.jms.service.PooledConnectionFactoryFactory
    with these properties:
     - pool = pooledjms
     - xa = true|false (both are registered)
-->
<reference id="pooledConnectionFactoryFactory"
        interface="org.ops4j.pax.jms.service.PooledConnectionFactoryFactory"
        filter="(&amp;(pool=pooledjms)(xa=true))" />

<!--
    When using XA connection factories, javax.transaction.TransactionManager service is not needed
here.
    It is used internally by xa-aware pooledConnectionFactoryFactory.
-->
<!--<reference id="tm" interface="javax.transaction.TransactionManager" />-->

<!--
    Finally, use both factories to expose the pooled, xa-aware, connection factory.
-->
<bean id="pool" factory-ref="pooledConnectionFactoryFactory" factory-method="create">
    <argument ref="connectionFactoryFactory" />
    <argument>
        <props>
            <!--
                Properties needed by artemis-specific org.ops4j.pax.jms.service.ConnectionFactoryFactory
            -->
            <prop key="jms.url" value="tcp://localhost:61616" />
            <prop key="jms.callTimeout" value="2000" />
            <prop key="jms.initialConnectAttempts" value="3" />
            <!-- Properties needed by pooled-jms-specific
org.ops4j.pax.jms.service.PooledConnectionFactoryFactory -->
            <prop key="pool.maxConnections" value="10" />
            <prop key="pool.idleTimeout" value="10000" />
        </props>
    </argument>
</bean>
```

```
<!--
   Expose connection factory for use by application code (such as Camel, Spring, ...)
-->
<service interface="javax.jms.ConnectionFactory" ref="pool">
   <service-properties>
      <!-- Giving connection factory a name using one of these properties makes identification easier
in jms:connectionfactories: -->
      <entry key="osgi.jndi.service.name" value="jms/artemis" />
      <!--<entry key="name" value="jms/artemis" />-->
      <!-- Without any of the above, name will fall back to "service.id" -->
   </service-properties>
</service>
```

前述の例は、接続ファクトリーのファクトリーを使用して、接続ファクトリーを作成する**ファクトリー** Bean を使用します。XA 対応の **PooledConnectionFactoryFactory** によって内部的に追跡されるため、**javax.transaction.TransactionManager** サービスを明示的に参照する必要はありません。

Fuse/Karaf シェルでは、以下のようになります。

```
karaf@root()> feature:install jms pax-jms-artemis pax-jms-pool-pooledjms

karaf@root()> install -s blueprint:file://$PQ_HOME/message-brokers/blueprints/artemis-pax-jms-
factory-pooledjms.xml
Bundle ID: 253
karaf@root()> bundle:services -p 253

Bundle 253 provides:
--------------------
objectClass = [javax.jms.ConnectionFactory]
osgi.jndi.service.name = jms/artemis
osgi.service.blueprint.compname = pool
service.bundleid = 253
service.id = 347
service.scope = bundle
-----
objectClass = [org.osgi.service.blueprint.container.BlueprintContainer]
osgi.blueprint.container.symbolicname = artemis-pax-jms-factory-pooledjms.xml
osgi.blueprint.container.version = 0.0.0
service.bundleid = 253
service.id = 348
service.scope = singleton

karaf@root()> jms:connectionfactories
JMS Connection Factory
------------------------------
jms/artemis

karaf@root()> jms:info -u admin -p admin jms/artemis
Property │ Value
─────────┼──────────────────────────────────────────────────
product  │ ActiveMQ
version  │ 2.4.0.amq-711002-redhat-1
```

上記の一覧に示すように、Blueprint バンドルは、汎用的でブローカー固有でない接続プールである **javax.jms.ConnectionFactory** サービスをエクスポートします。Blueprint XML には明示的な **<service**

ref="artemis"> 宣言がないため、ブローカー固有の **javax.jms.XAConnectionFactory** は OSGi サービスとして登録**されません**。

7.6.3. 接続ファクトリーの混合デプロイメント

pax-jms-config 1.0.0 バンドルは、サービスプロパティーを使用して、プーリング接続ファクトリー内にブローカー固有の接続ファクトリーを **ラッピング** する別の方法を追加します。このメソッドは、Fuse 6.x での作業に使用した方法と同じです。

Blueprint XML の例を以下に示します。

```
<!--
    Broker-specific, non-pooling, non-enlisting javax.jms.XAConnectionFactory
-->
<bean id="artemis" class="org.apache.activemq.artemis.jms.client.ActiveMQXAConnectionFactory">
    <argument value="tcp://localhost:61616" />
    <property name="callTimeout" value="2000" />
    <property name="initialConnectAttempts" value="3" />
</bean>

<!--
    Expose broker-specific connection factory with service properties.
    No need to expose pooling, enlisting, non broker-specific javax.jms.XAConnectionFactory. It will be registered
    automatically by pax-jms-config with the same properties as this <service>, but with a higher service.ranking
-->
<service id="pool" ref="artemis" interface="javax.jms.XAConnectionFactory">
    <service-properties>
        <!-- "pool" key is needed for pax-jms-config to wrap broker-specific connection factory inside connection pool -->
        <entry key="pool" value="pooledjms" />
        <!-- <service>/@id attribute does not propagate, but name of the connection factory is required using one of: -->
        <entry key="osgi.jndi.service.name" value="jms/artemis" />
        <!-- or: -->
        <!--<entry key="name" value="jms/artemis" />-->
        <!-- Other properties, that normally by e.g., pax-jms-pool-pooledjms -->
        <entry key="pool.maxConnections" value="10" />
        <entry key="pool.idleTimeout" value="10000" />
    </service-properties>
</service>
```

上記の例では、ブローカー固有の接続ファクトリーのみを手動で登録できます。**pool=pooledjms** サービスプロパティーは、**pax-jms-config** バンドルによって管理される接続ファクトリートラッカーのヒントです。このサービスプロパティーを持つ接続ファクトリーサービスは、プール接続ファクトリー内でラップされます (この例では **pax-jms-pool-pooledjms** 。

Fuse/Karaf シェルでは、以下のようになります。

```
karaf@root()> feature:install jms pax-jms-config pax-jms-artemis pax-jms-pool-pooledjms

karaf@root()> install -s blueprint:file://$PQ_HOME/message-brokers/blueprints/artemis-pax-jms-discovery.xml
Bundle ID: 254
```

```
karaf@root()> bundle:services -p 254

Bundle 254 provides:
--------------------
objectClass = [javax.jms.XAConnectionFactory]
osgi.jndi.service.name = jms/artemis
osgi.service.blueprint.compname = artemis
pool = pooledjms
pool.idleTimeout = 10000
pool.maxConnections = 10
service.bundleid = 254
service.id = 349
service.scope = bundle
-----
objectClass = [org.osgi.service.blueprint.container.BlueprintContainer]
osgi.blueprint.container.symbolicname = artemis-pax-jms-discovery.xml
osgi.blueprint.container.version = 0.0.0
service.bundleid = 254
service.id = 351
service.scope = singleton

karaf@root()> service:list javax.jms.XAConnectionFactory
[javax.jms.XAConnectionFactory]
-------------------------------
 osgi.jndi.service.name = jms/artemis
 osgi.service.blueprint.compname = artemis
 pool = pooledjms
 pool.idleTimeout = 10000
 pool.maxConnections = 10
 service.bundleid = 254
 service.id = 349
 service.scope = bundle
Provided by :
 Bundle 254
Used by:
 OPS4J Pax JMS Config (251)

karaf@root()> service:list javax.jms.ConnectionFactory
[javax.jms.ConnectionFactory]
-----------------------------
 osgi.jndi.service.name = jms/artemis
 osgi.service.blueprint.compname = artemis
 pax.jms.managed = true
 pax.jms.service.id.ref = 349
 pool.idleTimeout = 10000
 pool.maxConnections = 10
 service.bundleid = 251
 service.id = 350
 service.ranking = 1000
 service.scope = singleton
Provided by :
 OPS4J Pax JMS Config (251)

karaf@root()> jms:connectionfactories
JMS Connection Factory
```

```
──────────────────────────────
jms/artemis

karaf@root()> jms:info -u admin -p admin jms/artemis
Property │ Value
─────────┼──────────────────────────────────────────
product  │ ActiveMQ
version  │ 2.4.0.amq-711002-redhat-1
```

前述の例では、このコマンドは重複した名前を削除するため、**jms:connectionfactories** は 1 つのサービスのみを示しています。データソースの混合デプロイメントにおいて、2 つのサービスが **jdbc:ds-list** によって表されました。

javax.jms.XAConnectionFactory は Blueprint バンドルから登録され、 **pool = pooledjms** プロパティーが宣言されています。

javax.jms.ConnectionFactory は **pax-jms-config** バンドルから登録されます。また、

- **pool = pooledjms** プロパティーがありません。ラッパー接続ファクトリーの登録時に削除されました。
- **service.ranking = 1000** プロパティーがあるため、名前で接続ファクトリーを検索する場合など、常に優先されるバージョンになります。
- **pax.jms.managed = true** プロパティーがあるので、再度ラップが試行されていません。
- これには、接続プール内でラップされる元の接続ファクトリーサービスを示す **pax.jms.service.id.ref = 349** プロパティーがあります。

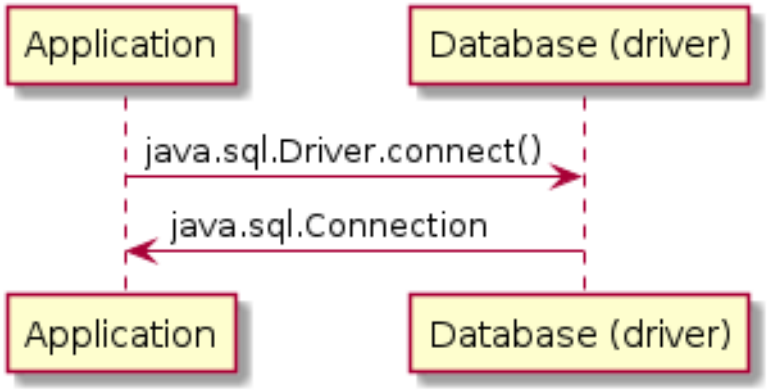
第8章 JAVA CONNECTOR ARCHITECTURE (JCA) の概要

JCA 仕様は、以下の 3 つの参加者を含むシナリオを **一般化** するために作成されました。

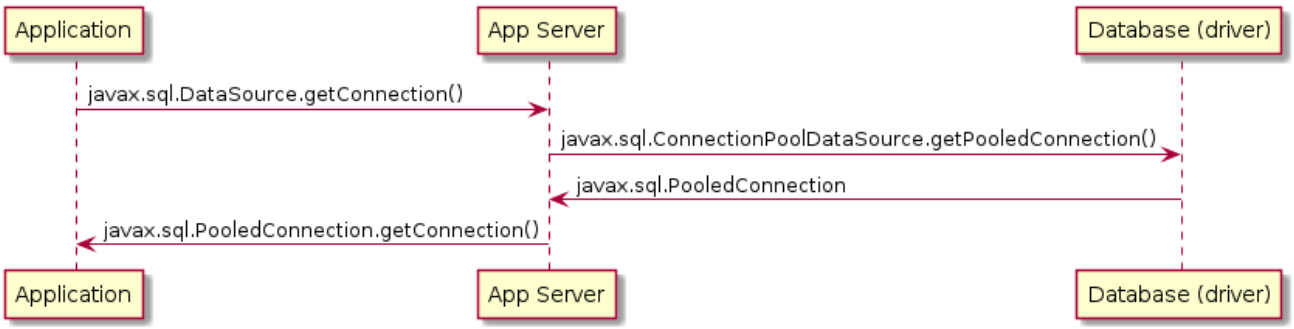
- データベースなどの外部システムまたは **通常は** EIS システム
- アプリケーションサーバー
- デプロイされたアプリケーション

8.1. SIMPLE JDBC ANALOGY

アプリケーションとデータベースしかない最も単純なシナリオでは、次のようになります。



javax.sql.DataSource を公開するアプリケーションサーバーを追加すると、次のようになります (XA などのデータソースのさまざまな側面を再呼び出しすることなく)。



8.2. JCA の使用についての概要

JCA は、**ドライバーとアプリケーションサーバーとの間の双方向通信を追加して、データベース ドライバー** の概念を一般化します。ドライバーは、**javax.resource.spi.ResourceAdapter** で表される **リソースアダプター** になります。

以下の 2 つの重要なインターフェイスがあります。

- **javax.resource.spi.ManagedConnectionFactory** implemented by a resource adapter.
- アプリケーションサーバーで実装される **javax.resource.spi.ConnectionManager**

ManagedConnectionFactory インターフェイスには 2 つの目的があります。

- **Object createConnectionFactory(ConnectionManager cxManager)** メソッドは、アプリケーションコードが使用できる指定の EIS (またはデータベースまたはメッセージブローカー) の **接続ファクトリー** を生成するために使用できます。返されたオブジェクト **メソッド**

  - 汎用 **javax.resource.cci.ConnectionFactory** (詳細は JCA 1.6, chapter 17: **Common Client Interface**を参照)

  - よく知られている **javax.sql.DataSource** または **javax.jms.ConnectionFactory** などの EIS 固有の接続ファクトリー。これは、**pax-transx-jdbc** および **pax-transx-jms** バンドルによって使用される **接続ファクトリー** のタイプです。

- **アプリケーションサーバー** によって使用される **javax.resource.spi.ManagedConnection ManagedConnectionFactory.createManagedConnection()** メソッドでは、EIS/database/broker への実際の物理接続を作成します。

ConnectionManager は **アプリケーションサーバー** によって実装され、**リソースアダプター** によって使用されます。これは、最初に QoS 操作 (プーリング、セキュリティー、トランザクション管理) を実行し、最終的に **リソースアダプター** の **ManagedConnectionFactory** に委譲して **ManagedConnection** インスタンスを作成する **アプリケーションサーバー** です。フローは以下のようになります。

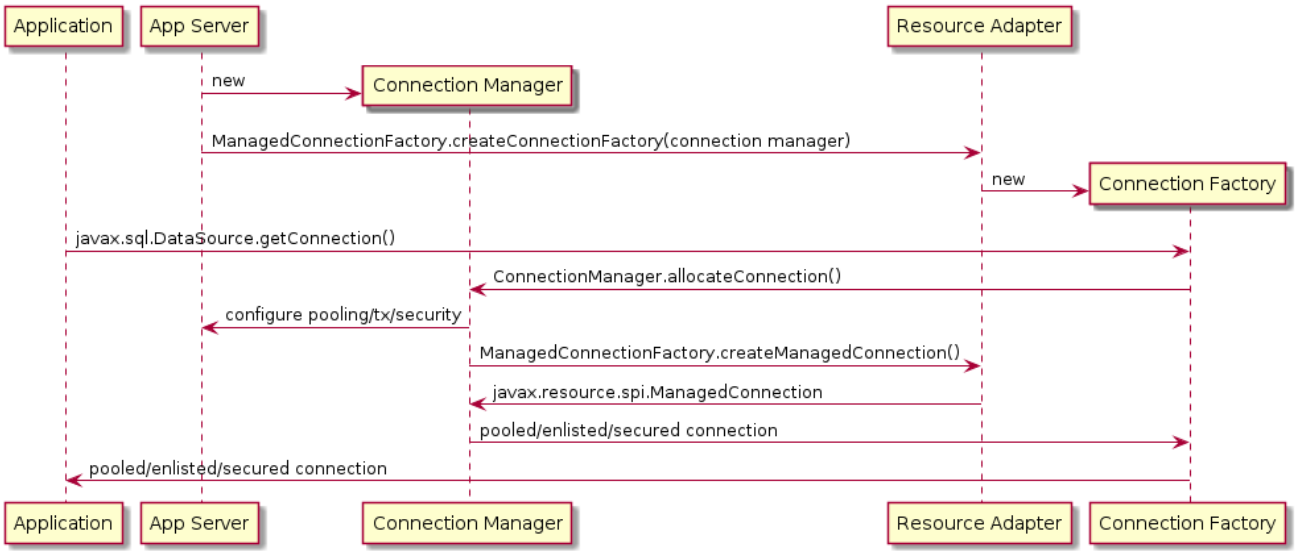
1. アプリケーションコードは、**ManagedConnectionFactory.createConnectionFactory()** から返されるオブジェクトを使用して **アプリケーションサーバー** で作成され、公開される **接続ファクトリー** を使用します。これは一般的な CCI インターフェイス (例: **javax.sql.DataSource**) である可能性があります。

2. この **接続ファクトリー** は、独自の接続で **接続** を作成しません。代わりに、**リソースアダプター**固有の **ManagedConnectionFactory** を渡す **ConnectionManager.allocateConnection()** に委譲します。

3. **アプリケーションサーバー** によって実装される **ConnectionManager** は、**サポートするオブジェクト** の作成、トランザクションプーリングの管理などを行い、最終的に渡された **ManagedConnectionFactory** から **物理 (管理) 接続** を取得します。

4. アプリケーションコードは、通常、**リソースアダプター**の特定の **物理接続** に委譲する **アプリケーションサーバー** によって作成されるラッパー/プロキシーである**コネクション** を取得します。

以下は、**アプリケーションサーバー** が EIS 固有の EIS 接続ファクトリー以外の **接続ファクトリー** を作成した図です。EIS (ここではデータベース) へのアクセスは **javax.sql.DataSource** インターフェイスを使用して行われます。ドライバーのタスクは **物理接続**を提供することで、**アプリケーションサーバー** はプーリング/登録/セキュリティーを行うプロキシー (通常) 内でラップします。



8.3. PAX-TRANSX プロジェクト

pax-transx プロジェクトは、OSGi での JTA/JTS トランザクション管理、および JDBC および JMS のリソースプーリングのサポートを提供します。**pax-jdbc** と **pax-jms** の間の差異をなくします。

- **pax-jdbc** は、**javax.sql.(XA)ConnectionFactory** サービスの設定オプションと検出を追加し、JDBC プーリング実装の一部が提供されます。
- **pax-jms** は、**javax.jms.(XA)ConnectionFactory** サービスと同じことを行い、一部の JMS プーリング実装を提供します。
- **pax-transx** は **javax.transaction.TransactionManager** 実装の設定オプションおよび検出を追加し、(最終的に) JCA ベースの JDBC/JMS 接続管理をプーリングおよびトランザクションサポートとともに提供します。

JDBC 接続プール と JMS 接続プール のセクションは引き続き有効です。JCA ベースのプールを使用するために必要な変更は、JDBC データソースと JMS 接続ファクトリーの登録時に **pool=transx** プロパティーを使用することだけです。

- **pax-jdbc-pool-transx** は **pax-transx-jdbc** の **org.ops4j.pax.transx.jdbc.ManagedDataSourceBuilder** を使用します。
- **pax-jms-pool-transx** は **pax-transx-jms** の **org.ops4j.pax.transx.jms.ManagedConnectionFactoryBuilder** を使用します。

プールされたデータソース/接続ファクトリーは **ビルダースタイル** (Java™ Bean プロパティーなし) で作成されますが、JDBC ではこれらのプロパティーがサポートされます。

- **name**
- **userName**
- **password**
- **commitBeforeAutocommit**
- **preparedStatementCacheSize**
- **transactionIsolationLevel**

- **minIdle**
- **maxPoolSize**
- **aliveBypassWindow**
- **houseKeepingPeriod**
- **connectionTimeout**
- **idleTimeout**
- **maxLifetime**

これらのプロパティーは JMS でサポートされます。

- **name**
- **userName**
- **password**
- **clientID**
- **minIdle**
- **maxPoolSize**
- **aliveBypassWindow**
- **houseKeepingPeriod**
- **connectionTimeout**
- **idleTimeout**
- **maxLifetime**

XA リカバリーが機能するには、**userName** と **password** プロパティーが必要です (Fuse 6.x の **aries.xa.username** プロパティーおよび **aries.xa.password** プロパティーと同様)。

Blueprint でのこの JDBC 設定 (**pool=transx** を考慮) は次のとおりです。

```
<!--
    Database-specific, non-pooling, non-enlisting javax.sql.XADataSource
-->
<bean id="postgresql" class="org.postgresql.xa.PGXADataSource">
    <property name="url" value="jdbc:postgresql://localhost:5432/reportdb" />
    <property name="user" value="fuse" />
    <property name="password" value="fuse" />
    <property name="currentSchema" value="report" />
    <property name="connectTimeout" value="5" />
</bean>

<!--
    Expose database-specific data source with service properties
    No need to expose pooling, enlisting, non database-specific javax.sql.DataSource - it'll be
```

```
registered
    automatically by pax-jdbc-config with the same properties as this <service>, but with higher
service.ranking
-->
<service id="pool" ref="postgresql" interface="javax.sql.XADataSource">
    <service-properties>
        <!-- "pool" key is needed for pax-jdbc-config to wrap database-specific data source inside
connection pool -->
        <entry key="pool" value="transx" />
        <!-- <service>/@id attribute doesn't propagate, but name of the datasource is required using one
of: -->
        <entry key="osgi.jndi.service.name" value="jdbc/postgresql" />
        <!-- or: -->
        <!--<entry key="dataSourceName" value="jdbc/postgresql" />-->
        <!-- Other properties, that normally are needed by e.g., pax-jdbc-pool-transx -->
        <entry key="pool.maxPoolSize" value="13" />
        <entry key="pool.userName" value="fuse" />
        <entry key="pool.password" value="fuse" />
    </service-properties>
</service>
```

Blueprint でのこの JMS 設定 (**pool=transx** を考慮) は次のとおりです。

```
<!--
    Broker-specific, non-pooling, non-enlisting javax.jms.XAConnectionFactory
-->
<bean id="artemis" class="org.apache.activemq.artemis.jms.client.ActiveMQXAConnectionFactory">
    <argument index="0" value="tcp://localhost:61616" />
    <!-- credentials needed for JCA-based XA-recovery -->
    <argument index="1" value="admin" />
    <argument index="2" value="admin" />
    <property name="callTimeout" value="2000" />
    <property name="initialConnectAttempts" value="3" />
</bean>

<!--
    Expose broker-specific connection factory with service properties
    No need to expose pooling, enlisting, non broker-specific javax.jms.XAConnectionFactory - it'll be
registered
    automatically by pax-jms-config with the same properties as this <service>, but with higher
service.ranking
-->
<service id="pool" ref="artemis" interface="javax.jms.XAConnectionFactory">
    <service-properties>
        <!-- "pool" key is needed for pax-jms-config to wrap broker-specific connection factory inside
connection pool -->
        <entry key="pool" value="transx" />
        <!-- <service>/@id attribute doesn't propagate, but name of the connection factory is required
using one of: -->
        <entry key="osgi.jndi.service.name" value="jms/artemis" />
        <!-- or: -->
        <!--<entry key="name" value="jms/artemis" />-->
        <!-- Other properties, that normally are needed e.g., pax-jms-pool-transx -->
        <entry key="pool.maxPoolSize" value="13" />
        <entry key="pool.userName" value="admin" />
```

```
        <entry key="pool.password" value="admin" />
    </service-properties>
</service>
```

JCA ベースのリソース管理を活用する JDBC データソースと JMS 接続ファクトリーが登録されます。transx ベースのプールは XA リカバリーに関して **pax-transx-tm-narayana** と適切に統合されます。

必要な機能は以下のとおりです。

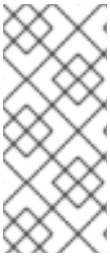
- **pax-jdbc-pool-tranx**
- **pax-jms-pool-tranx**
- **pax-transx-jdbc**
- **pax-transx-jms**
- **pax-jms-artemis** (A-MQ 7 を使用する場合)

第9章 トランザクションを使用する CAMEL アプリケーションの作成

参照可能な 3 つのタイプのサービスを設定すると、アプリケーションを作成する準備が整います。以下の 3 つのタイプのサービスがあります。

- 以下のインターフェイスのいずれかを実装する 1 つのトランザクションマネージャー。
  - **javax.transaction.UserTransaction**
  - **javax.transaction.TransactionManager**
  - **org.springframework.transaction.PlatformTransactionManager**
- **javax.sql.DataSource**. インターフェイスを実装する JDBC データソースが少なくとも 1 つ。多くの場合、複数のデータソースが存在します。
- **javax.jms.ConnectionFactory** インターフェイスを実装する JMS 接続ファクトリーが少なくとも 1 つ。多くの場合、複数の値が存在します。

ここでは、トランザクション、データソース、および接続ファクトリーの管理に関連する Camel 固有の設定を説明します。



注記

ここでは、**SpringTransactionPolicy** などの Spring 関連の概念をいくつか説明します。**SpringXMLDSL** と **BlueprintXMLDSL** には明確な違いがあり、どちらも Camel コンテキストを定義する XML 言語です。Spring XML DSL は Fuse で **非推奨** になりました。ただし、Camel トランザクションメカニズムは引き続き内部で Spring ライブラリーを使用します。

ここでの情報のほとんどは、使用される **PlatformTransactionManager** の種類に依存していません。**PlatformTransactionManager** が Narayana トランザクションマネージャーの場合、完全な JTA トランザクションが使用されます。**PlatformTransactionManager** がローカルの Blueprint **<bean>** として定義されている場合 (例:**org.springframework.jms.connection.JmsTransactionManager**)、ローカルトランザクションが使用されます。

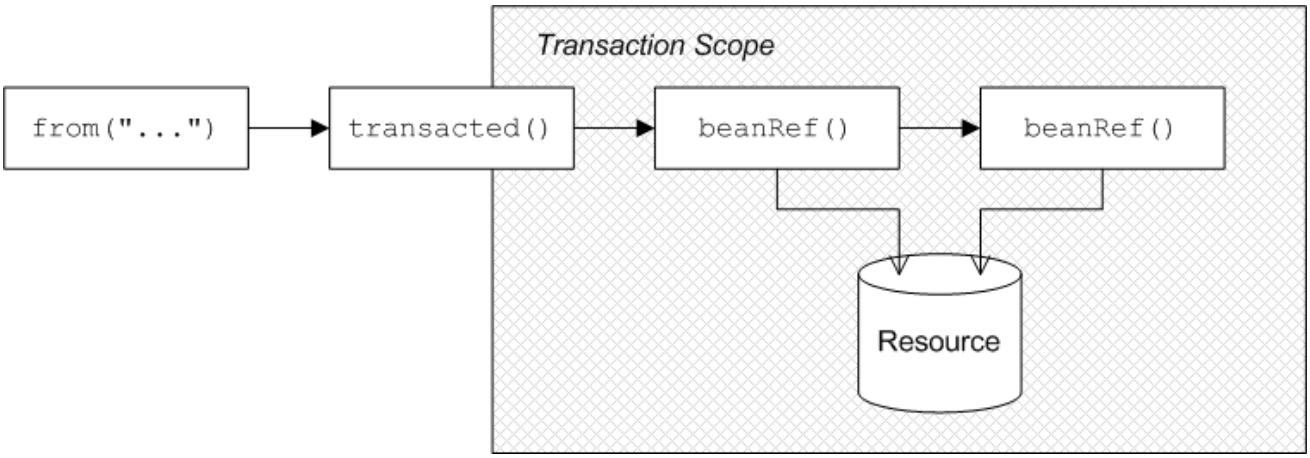
トランザクション境界とは、トランザクションを開始、コミット、およびロールバックする手順を指します。本セクションでは、プログラミングと設定の両方でトランザクション境界を制御することができるメカニズムを説明します。

- 「ルートをマークすることによるトランザクションの境界」
- 「トランザクションエンドポイントによる境界」
- 「宣言型トランザクションによる境界」
- 「トランザクション伝播ポリシー」
- 「エラー処理およびロールバック」

9.1. ルートをマークすることによるトランザクションの境界

Apache Camel は、ルートでトランザクションを開始する簡単なメカニズムを提供します。Java DSL で **transacted()** コマンドを挿入するか、XML DSL で **<transacted/>** タグを挿入します。

図9.1 ルートをマークすることによる境界



トランザクション処理されたプロセッサーは、トランザクションを次のように区切ります。

1. 交換がトランザクションプロセッサーに入ると、トランザクションプロセッサーはデフォルトのトランザクションマネージャーを呼び出し、トランザクションを開始して、現在のスレッドにトランザクションをアタッチします。

2. エクスチェンジが残りのルートの最後に到達すると、処理されたプロセッサーはトランザクションマネージャーを呼び出して現在のトランザクションをコミットします。

9.1.1. JDBC リソースを使用したルートのサンプル

図9.1「ルートをマークすることによる境界」 は、ルートに **transacted()** DSL コマンドを追加することで、トランザクションになるルートの例を示しています。**transacted()** ノードに続くすべてのルートノードはトランザクションスコープに含まれます。この例では、以下の 2 つのノードが JDBC リソースにアクセスします。

9.1.2. Java DSL でのルート定義

以下の Java DSL の例は、**transacted()** DSL コマンドでルートをマークしてトランザクションルートを定義する方法を示しています。

```
import org.apache.camel.builder.RouteBuilder;

class MyRouteBuilder extends RouteBuilder {
    public void configure() {
        from("file:src/data?noop=true")
                .transacted()
                .bean("accountService","credit")
                .bean("accountService","debit");
    }
}
```

この例では、ファイルエンドポイントは、あるアカウントから別のアカウントへの資金の移動を記述するいくつかの XML 形式のファイルを読み取ります。最初の **bean()** 呼び出しでは、指定された金額が受取人の口座に加算され、2 回目の **bean()** 呼び出しでは、支払人の口座からから指定の金額が差し引かれます。どちらの **bean()** 呼び出しでも、データベースリソースに対する更新が行われます。データベースリソースがトランザクションマネージャーを介してトランザクションにバインドされることを前提とします (例:6章*JDBC データソースの使用*)。

9.1.3. Blueprint XML でのルート定義

上記のルートは Blueprint XML で表現することもできます。**<transacted />** タグは、以下の XML のようにルートをトランザクションとしてマークします。

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ...>

   <camelContext xmlns="http://camel.apache.org/schema/blueprint">
      <route>
         <from uri="file:src/data?noop=true" />
         <transacted />
         <bean ref="accountService" method="credit" />
         <bean ref="accountService" method="debit" />
      </route>
   </camelContext>

</blueprint>
```

9.1.4. デフォルトの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ポリシー

トランザクションを区別するには、トランザクションプロセッサーを特定のトランザクションマネージャーインスタンスに関連付ける必要があります。**transacted()** を呼び出すたびにトランザクションマネージャーを指定しなくても、トランザクションプロセッサーが適切なデフォルトを自動的に選択します。たとえば、設定にトランザクションマネージャーのインスタンスが 1 つしかない場合、transacted プロセッサーはこのトランザクションマネージャーを暗黙的に選択し、トランザクションを区別するために使用されます。

トランザクションプロセッサーは、**TransactedPolicy** タイプのトランザクションポリシーで設定することもできます。これは、伝播ポリシーとトランザクションマネージャーがカプセル化します (詳細は「トランザクション伝播ポリシー」 を参照してください)。デフォルトのトランザクションマネージャーまたはトランザクションポリシーを選択するには、以下のルールが使用されます。

1. **org.apache.camel.spi.TransactedPolicy** タイプの Bean が 1 つしかない場合は、この Bean を使用します。

注記

TransactedPolicy タイプは、「トランザクション伝播ポリシー」 で説明されている **SpringTransactionPolicy** 型のベースタイプです。そのため、ここで参照される Bean は **SpringTransactionPolicy** Bean である可能性があります。

2. **ID** や **PROPAGATION_REQUIRED** が含まれる、**org.apache.camel.spi.TransactedPolicy** の Bean タイプがある場合には、この Bean を使用します。

3. **org.springframework.transaction.PlatformTransactionManager** タイプの Bean が 1 つしかない場合は、この Bean を使用します。

Bean ID を **transacted()** に引数として指定することで、Bean を明示的に指定するオプションもあります。「Java DSL の **PROPAGATION_NEVER** ポリシーを使用したルートの例」 を参照してください。

9.1.5. トランザクションスコープ

トランザクションプロセッサーをルートに挿入すると、トランザクションマネージャーは、エクスチェンジがこのノードを通過するたびに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スコープは以下のように定義されます。

- トランザクションは現在のスレッドにのみ関連付けられます。
- トランザクションスコープには、トランザクションプロセッサーに続くすべてのルートノードが含まれます。

トランザクションプロセッサーの前にあるルートノードはトランザクションには含まれません。ただし、ルートがトランザクションエンドポイントで始まる場合は、ルートのすべてのノードがトランザクションに含まれます。「ルート開始時のトランザクションエンドポイント」を参照してください。

以下のルートを見てみましょう。**transacted()** DSL コマンドは、データベースリソースにアクセスする最初の **bean()** 呼び出しの後に誤って発生するため、正しくありません。

```
// Java
import org.apache.camel.builder.RouteBuilder;

public class MyRouteBuilder extends RouteBuilder {
    ...
    public void configure() {
        from("file:src/data?noop=true")
                .bean("accountService", "credit")
                .transacted()  // <-- WARNING: Transaction started in the wrong place!
                .bean("accountService", "debit");
    }
}
```

9.1.6. トランザクションルートにスレッドプールがない

特定のトランザクションが現在のスレッドにのみ関連付けられていることを理解することが重要です。新しいスレッドでの処理は現在のトランザクションに参加しないため、トランザクションルートの途中でスレッドプールを作成しないでください。たとえば、次のルートは問題を引き起こす可能性があります。

```
// Java
import org.apache.camel.builder.RouteBuilder;

public class MyRouteBuilder extends RouteBuilder {
    ...
    public void configure() {
        from("file:src/data?noop=true")
                .transacted()
                .threads(3)  // WARNING: Subthreads are not in transaction scope!
                .bean("accountService", "credit")
                .bean("accountService", "debit");
    }
}
```

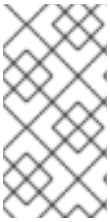
threads() DSL コマンドはトランザクションルートと互換性がないため、前述のルートなどのルートは、データベースが破損する可能性があります。**threads()** 呼び出しが **transacted()** 呼び出しの前であっても、ルートは想定どおりに動作しません。

9.1.7. フラグメントへのルートの分割

ルートをフラグメントに分割し、各ルートのフラグメントを現在のトランザクションに参加させる場合は、**direct:** エンドポイントを使用することができます。たとえば、エクスチェンジを個別のルートフラグメントに送信するには、転送量が大きい (100 を超える) または小さい (100 以下) かに応じて以下のように **choice()** DSL コマンドとダイレクトエンドポイントを使用できます。

```
// Java
import org.apache.camel.builder.RouteBuilder;

public class MyRouteBuilder extends RouteBuilder {
    ...
    public void configure() {
        from("file:src/data?noop=true")
                .transacted()
                .bean("accountService", "credit")
                .choice().when(xpath("/transaction/transfer[amount > 100]"))
                .to("direct:txbig")
                .otherwise()
                .to("direct:txsmall");

        from("direct:txbig")
                .bean("accountService", "debit")
                .bean("accountService", "dumpTable")
                .to("file:target/messages/big");

        from("direct:txsmall")
                .bean("accountService", "debit")
                .bean("accountService", "dumpTable")
                .to("file:target/messages/small");
    }
}
```

direct エンドポイントは同期されているため、**direct:txbig** で始まるフラグメントと **direct:txsmall** で始まるフラグメントの両方は、現在のトランザクションに参加します。つまり、フラグメントは最初のルートフラグメントと同じスレッドで実行されるので、同じトランザクションスコープに含まれています。



注記

ルートのフラグメントに参加するために **seda** エンドポイントを使用することはできません。**seda** コンシューマーエンドポイントは、ルートフラグメント (非同期処理) を実行する新しいスレッドを作成します。したがって、フラグメントは元のトランザクションには参加しません。

9.1.8. リソースエンドポイント

以下の Apache Camel コンポーネントは、ルートの宛先として表示される場合にリソースエンドポイントとして機能します。たとえば、**to()** DSL コマンドに表示される場合です。つまり、これらのエンドポイントはデータベースや永続キューなどのトランザクションリソースにアクセスできます。リソースエンドポイントは、現在のトランザクションを開始したトランザクションプロセッサーと同じトランザクションマネージャーに関連付けられている限り、現在のトランザクションに参加できます。

- ActiveMQ
- AMQP

- Hibernate
- iBatis
- JavaSpace
- JBI
- JCR
- JDBC
- JMS
- JPA
- LDAP

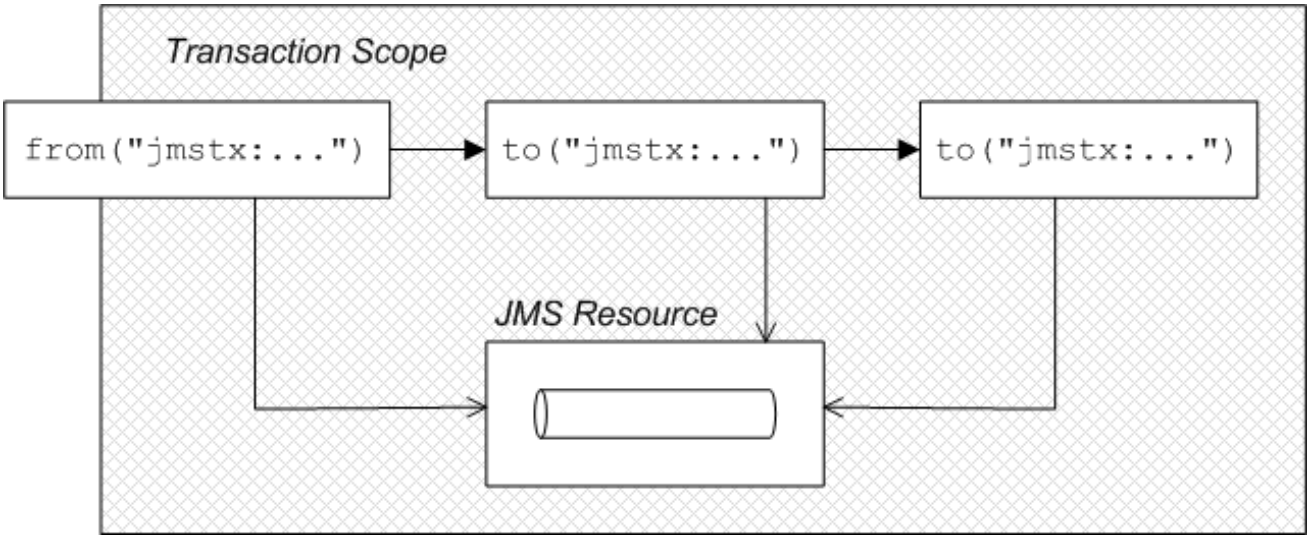
9.1.9. リソースエンドポイントを含むサンプルルート

以下の例は、リソースエンドポイントを含むルートを示しています。これにより、送金の注文が 2 つの異なる JMS キューに送信されます。**credits** キューは、受取人の口座に入金するために注文を処理します。**debits** キューは、送金人の口座から差し引く注文を処理します。対応するデビッドがある場合にのみ、クレジットがあります。したがって、キューへの追加操作を単一のトランザクションに含める必要があります。トランザクションが成功すると、クレジット注文とデビット注文の両方がキューに入れられます。エラーが発生した場合には、どちらの注文もキューに入れられません。

```
from("file:src/data?noop=true")
        .transacted()
        .to("jmstx:queue:credits")
        .to("jmstx:queue:debits");
```

9.2. トランザクションエンドポイントによる境界

ルート開始時のコンシューマーエンドポイントがリソースにアクセスすると、エクスチェンジのポーリング後にトランザクションを開始するため、**transacted()** コマンドを使用する意味がありません。つまり、トランザクションの開始が遅すぎて、コンシューマーエンドポイントをトランザクションスコープに含めることができません。この場合、正しい方法は、トランザクションを開始するエンドポイント自体を設定することです。トランザクションを管理できるエンドポイントは、**トランザクションエンドポイント** と呼ばれます。



トランザクションエンドポイントによる境界設定には、次の 2 つの異なるモデルがあります。

- 一般的なケース: 通常、トランザクションエンドポイントは以下のようにトランザクションを区切ります。

  1. 交換がエンドポイントに到着したとき、またはエンドポイントが交換のポーリングに成功したとき、エンドポイントは関連するトランザクションマネージャーを呼び出してトランザクションを開始します。

  2. エンドポイントは新しいトランザクションを現在のスレッドに割り当てます。

  3. エクスチェンジがルートの最後に到達すると、トランザクションエンドポイントはトランザクションマネージャーを呼び出して現在のトランザクションをコミットします。

- **InOut** エクスチェンジのある JMS エンドポイント: JMS コンシューマーエンドポイントが **InOut** エクスチェンジを受信し、このエクスチェンジが別の JMS エンドポイントにルーティングされる場合は、特別なケースとして扱う必要があります。問題は、要求/応答交換全体を 1 つのトランザクションで囲み込もうとすると、ルートがデッドロックになってしまう可能性があることです。

9.2.1. JMS エンドポイントを使用したルートの例

「トランザクションエンドポイントによる境界」 は、ルート (**from()** コマンド内で) の開始時にトランザクションエンドポイントがあることによってルートがトランザクションになる例を示しています。すべてのルートノードがトランザクションスコープに含まれます。この例では、ルートのすべてのエンドポイントが JMS リソースにアクセスします。

9.2.2. Java DSL でのルート定義

以下の Java DSL の例は、トランザクションエンドポイントでルートを起動してトランザクションルートを定義する方法を示しています。

```
from("jmstx:queue:giro")
        .to("jmstx:queue:credits")
        .to("jmstx:queue:debits");
```

上記の例では、トランザクションスコープにエンドポイント **jmstx:queue:giro**、**jmstx:queue:credits** および **jmstx:queue:debits** が含まれます。トランザクションに成功すると、エクスチェンジは **giro** キューから永続的に削除され、**credits** キューおよび **debits** キューにプッシュされます。トランザクションが失敗すると、エクスチェンジは **credits** および **debits** キューに配置されず、エクスチェンジは

giro キューに戻されます。デフォルトでは、JMS はメッセージの再配信を自動的に試行します。以下のように、JMS コンポーネント Bean **jmstx** はトランザクションを使用するように明示的に設定する必要があります。

```
<blueprint ...>
  <bean id="jmstx" class="org.apache.camel.component.jms.JmsComponent">
    <property name="configuration" ref="jmsConfig" />
  </bean>

  <bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration">
    <property name="connectionFactory" ref="jmsConnectionFactory" />
    <property name="transactionManager" ref="jmsTransactionManager" />
    <property name="transacted" value="true" />
  </bean>
  ...
</blueprint>
```

前述の例では、トランザクションマネージャーインスタンス **jmsTransactionManager** は JMS コンポーネントに関連付けられ、**transacted** プロパティーは **true** に設定され、**InOnly** エクスチェンジのトランザクションデマケーションを有効にします。

9.2.3. Blueprint XML でのルート定義

上記のルートは、以下のように Blueprint XML で表現できます。

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

  <camelContext xmlns="http://camel.apache.org/schema/blueprint">
    <route>
      <from uri="jmstx:queue:giro" />
      <to uri="jmstx:queue:credits" />
      <to uri="jmstx:queue:debits" />
    </route>
  </camelContext>

</blueprint>
```

9.2.4. DSL transacted() コマンドは必要ありません。

トランザクションエンドポイントで始まるルートに **transacted()** DSL コマンドは必要ありません。ただし、デフォルトのトランザクションポリシーが **PROPAGATION_REQUIRED** (「トランザクション伝播ポリシー」 を参照) であることを仮定すると、以下の例のように **transacted()** コマンドを含めることは通常問題ありません。

```
from("jmstx:queue:giro")
      .transacted()
      .to("jmstx:queue:credits")
      .to("jmstx:queue:debits");
```

ただし、このルートは、デフォルト以外の伝播ポリシーを持つ単一の **TransactedPolicy** Bean が Blueprint XML で作成されている場合など、このルートが予期せぬ方法で動作する可能性があります。「デフォルトの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ポリシー」を参照してください。そのため、通常はトランザクションエンドポイントで始まるルートに **transacted()** DSL コマンドを含めない方がよいでしょう。

9.2.5. ルート開始時のトランザクションエンドポイント

以下の Apache Camel コンポーネントは、ルートの開始点で表示される場合にトランザクションエンドポイントとして機能します。たとえば、**from()** DSL コマンドに表示される場合です。つまり、これらのエンドポイントはトランザクションクライアントとして動作するよう設定でき、トランザクションリソースにもアクセスできます。

- ActiveMQ
- AMQP
- JavaSpace
- JMS
- JPA

9.3. 宣言型トランザクションによる境界

ブループリント XML を使用する場合、ブループリント XML ファイルでトランザクションポリシーを宣言して、トランザクションの境界を定めることもできます。たとえば、**Required** ポリシーなどで適切なトランザクションポリシーを Bean または Bean メソッドに適用すると、特定の Bean または Bean メソッドが呼び出されるたびにトランザクションが開始されるようにすることができます。bean メソッドの最後に、トランザクションがコミットされます。このアプローチは、トランザクションが Enterprise Java Bean で処理される方法と似ています。

OSGi 宣言的トランザクションでは、Blueprint ファイルの以下のスコープでトランザクションポリシーを定義できます。

- 「bean レベルの宣言」
- 「トップレベルの宣言」

「**tx:transaction** 属性の説明」 も参照してください。

9.3.1. bean レベルの宣言

Bean レベルでトランザクションポリシーを宣言するには、以下のように **tx:transaction** 要素を **bean** 要素の子として挿入します。

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
      xmlns:tx="http://aries.apache.org/xmlns/transactions/v1.1.0">

  <bean id="accountFoo" class="org.jboss.fuse.example.Account">
    <tx:transaction method="*" value="Required" />
    <property name="accountName" value="Foo" />
  </bean>

  <bean id="accountBar" class="org.jboss.fuse.example.Account">
    <tx:transaction method="*" value="Required" />
    <property name="accountName" value="Bar" />
  </bean>

</blueprint>
```

上記の例では、必要なトランザクションポリシーは **accountFoo** Bean と **accountBar** Bean のすべてのメソッドに適用されます。ここで、メソッド属性はすべての Bean メソッドと一致するワイルドカード * を指定します。

9.3.2. トップレベルの宣言

トップレベルでトランザクションポリシーを宣言するには、以下のように **tx:transaction** 要素を **blueprint** 要素の子として挿入します。

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
      xmlns:tx="http://aries.apache.org/xmlns/transactions/v1.1.0">

  <tx:transaction bean="account*" value="Required" />

  <bean id="accountFoo" class="org.jboss.fuse.example.Account">
    <property name="accountName" value="Foo" />
  </bean>

  <bean id="accountBar" class="org.jboss.fuse.example.Account">
    <property name="accountName" value="Bar" />
  </bean>

</blueprint>
```

上記の例では、**Required** トランザクションポリシーは、**ID** がパターン **account*** に一致するすべての Bean のメソッドに適用されます。

9.3.3. tx:transaction 属性の説明

tx:transaction 要素は以下の属性をサポートします。

bean

(トップレベルのみ) トランザクションポリシーが適用される Bean ID(コンマまたはスペース区切り) の一覧を指定します。以下に例を示します。

```
<blueprint ...>
  <tx:transaction bean="accountFoo,accountBar" value="..." />
</blueprint>
```

ワイルドカード文字 * も使用できます。これは、各リストエントリーで最大 1 度表示される可能性があります。以下に例を示します。

```
<blueprint ...>
  <tx:transaction bean="account*,jms*" value="..." />
</blueprint>
```

Bean 属性を省略すると、デフォルトで * に設定されます (Blueprint ファイルのすべての非合成 Bean と一致)。

メソッド

(トップレベルおよび Bean レベル) トランザクションポリシーが適用されるメソッド名 (コンマまたはスペース区切り) の一覧を指定します。以下に例を示します。

```
<bean id="accountFoo" class="org.jboss.fuse.example.Account">
   <tx:transaction method="debit,credit,transfer" value="Required" />
   <property name="accountName" value="Foo" />
</bean>
```

ワイルドカード文字 * も使用できます。これは、各リストエントリーで最大 1 度表示される可能性があります。

method 属性を省略すると、デフォルトで * に設定されます (該当する Bean のすべてのメソッドと一致)。

value

(トップレベルおよび Bean レベル) トランザクションポリシーを指定します。ポリシー値のセマンティクスは、以下のように EJB 3.0 仕様で定義されているポリシーと同じです。

- **Required** - 現在のトランザクションをサポートします。存在しない場合は新しいトランザクションを作成します。
- **Mandatory** - 現在のトランザクションをサポートします。現在のトランザクションが存在しない場合に例外を出力します。
- **RequiresNew** – 新しいトランザクションを作成し、現在のトランザクションが存在する場合は一時停止します。
- **Supports** - 現在のトランザクションをサポートします。存在しない場合は非トランザクションを実行します。
- **NotSupported** - 現在のトランザクションをサポートしません。むしろ常に非トランザクションで実行します。
- **Never** - 現在のトランザクションはサポートしません。現在のトランザクションが存在する場合は例外を出力します。

9.4. トランザクション伝播ポリシー

トランザクションクライアントが新しいトランザクションを作成する方法に影響を与える場合は、**JmsTransactionManager** を使用してそのトランザクションポリシーを指定できます。特に、Spring トランザクションポリシーを使用すると、トランザクションの伝播動作を指定できます。たとえば、トランザクションクライアントが新しいトランザクションを作成しようとしていて、トランザクションが現在のスレッドにすでに関連付けられていることを検知した場合に、そのまま新しいトランザクションを作成して、古いトランザクションを一時停止するべきですか ?また、既存のトランザクションに引き継がせる必要がありますか。これらの種類の動作は、トランザクションポリシーで伝播動作を指定することによって規制されます。

トランザクションポリシーは、Blueprint XML で Bean としてインスタンス化されます。その後、その Bean **ID** を **transacted()** DSL コマンドに引数として指定すると、トランザクションポリシーを参照できます。たとえば、動作 **PROPAGATION_REQUIRES_NEW** の対象となるトランザクションを開始する場合は、次のルートを使用できます。

```
from("file:src/data?noop=true")
        .transacted("PROPAGATION_REQUIRES_NEW")
        .bean("accountService","credit")
        .bean("accountService","debit")
        .to("file:target/messages");
```

PROPAGATION_REQUIRES_NEW 引数は、**PROPAGATION_REQUIRES_NEW** 動作で設定されるトランザクションポリシー Bean の Bean **ID** を指定します。「Blueprint XML でのポリシー Bean の定義」を参照してください。

9.4.1. Spring トランザクションポリシー

Apache Camel では、**org.apache.camel.spring.spi.SpringTransactionPolicy** クラスを使用して Spring トランザクションポリシーを定義できます。これは基本的にネイティブ Spring クラスのラッパーです。**SpringTransactionPolicy** クラスは以下の 2 つのデータをカプセル化します。

- **PlatformTransactionManager** タイプのトランザクションマネージャーへの参照
- 伝播動作

たとえば、以下のように **PROPAGATION_MANDATORY** 動作で Spring トランザクションポリシー Bean をインスタンス化できます。

```
<blueprint ...>
  <bean id="PROPAGATION_MANDATORY
"class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="transactionManager" ref="txManager" />
    <property name="propagationBehaviorName" value="PROPAGATION_MANDATORY" />
  </bean>
  ...
</blueprint>
```

9.4.2. 伝播動作の説明

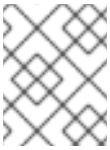
Spring では、以下の伝播動作がサポートされます。これらの値は当初、JavaeEE でサポートされる伝搬動作でモデル化されました。

PROPAGATION_MANDATORY

現在のトランザクションをサポートします。現在のトランザクションが存在しない場合は例外が発生します。

PROPAGATION_NESTED

現在のトランザクションが存在する場合はネストされたトランザクション内で実行し、それ以外の場合は **PROPAGATION_REQUIRED** のように動作します。



注記

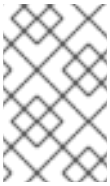
ネストされたトランザクションは、すべてのトランザクションマネージャーでサポートされているわけではありません。

PROPAGATION_NEVER

現在のトランザクションはサポートしません。現在のトランザクションが存在する場合は例外が発生します。

PROPAGATION_NOT_SUPPORTED

現在のトランザクションはサポートしません。常に非トランザクションを実行します。



注記

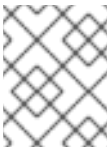
このポリシーでは、現在のトランザクションを一時停止する必要があります。これは、すべてのトランザクションマネージャーでサポートされているわけではありません。

PROPAGATION_REQUIRED

(デフォルト) 現在のトランザクションをサポートします。存在しない場合は、新しいトランザクションを作成します。

PROPAGATION_REQUIRES_NEW

新しいトランザクションを作成し、現在のトランザクションが存在する場合はそれを一時停止します。



注記

トランザクションの一時停止は、すべてのトランザクションマネージャーでサポートされているわけではありません。

PROPAGATION_SUPPORTS

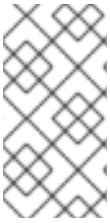
現在のトランザクションをサポートします。トランザクションが存在しない場合は、非トランザクションを実行します。

9.4.3. Blueprint XML でのポリシー Bean の定義

以下の例は、サポートされるすべての伝播動作に対してトランザクションポリシー Bean を定義する方法を示しています。便宜上、各 Bean ID は、伝播動作値で指定された値と同じですが、実際には、BeanID に任意の値を使用できます。

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

   <bean id="PROPAGATION_MANDATORY "
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
      <property name="transactionManager" ref="txManager" />
      <property name="propagationBehaviorName" value="PROPAGATION_MANDATORY" />
   </bean>

   <bean id="PROPAGATION_NESTED"
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
      <property name="transactionManager" ref="txManager" />
      <property name="propagationBehaviorName" value="PROPAGATION_NESTED" />
   </bean>

   <bean id="PROPAGATION_NEVER"
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
      <property name="transactionManager" ref="txManager" />
      <property name="propagationBehaviorName" value="PROPAGATION_NEVER" />
   </bean>

   <bean id="PROPAGATION_NOT_SUPPORTED"
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
      <property name="transactionManager" ref="txManager" />
```

```
        <property name="propagationBehaviorName" value="PROPAGATION_NOT_SUPPORTED" />
    </bean>

    <!-- This is the default behavior. -->
    <bean id="PROPAGATION_REQUIRED"
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
        <property name="transactionManager" ref="txManager" />
    </bean>

    <bean id="PROPAGATION_REQUIRES_NEW"
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
        <property name="transactionManager" ref="txManager" />
        <property name="propagationBehaviorName" value="PROPAGATION_REQUIRES_NEW" />
    </bean>

    <bean id="PROPAGATION_SUPPORTS"
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
        <property name="transactionManager" ref="txManager" />
        <property name="propagationBehaviorName" value="PROPAGATION_SUPPORTS" />
    </bean>

</blueprint>
```



注記

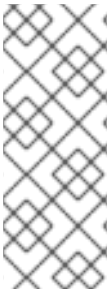
これらの bean 定義のいずれかを独自の Blueprint XML 設定に貼り付ける場合は、必ずトランザクションマネージャーへの参照をカスタマイズするようにしてください。つまり、**txManager** への参照をトランザクションマネージャー Bean の実際の **ID** に置き換えます。

9.4.4. Java DSL の PROPAGATION_NEVER ポリシーを使用したルートの例

トランザクションポリシーがトランザクションに何らかの影響を与えることを示す簡単な方法は、次のルートに示すように、既存のトランザクションの途中に **PROPAGATION_NEVER** ポリシーを挿入することです。

```
from("file:src/data?noop=true")
        .transacted()
        .bean("accountService","credit")
        .transacted("PROPAGATION_NEVER")
        .bean("accountService","debit");
```

このように使用すると、**PROPAGATION_NEVER** ポリシーはすべてのトランザクションを必然的に中止し、トランザクションのロールバックにつながります。アプリケーションへの影響を簡単に確認できるはずです。



注記

transacted() に渡される文字列の値は Bean **ID** で、伝播の動作名ではないことに注意してください。この例では、Bean **ID** が伝播動作名と同じになるように選択されますが、必ずしもそうである必要はありません。たとえば、アプリケーションで複数のトランザクションマネージャーを使用している場合、特定の伝播動作を持つ複数のポリシー Bean が作成される可能性があります。この場合、伝播の動作の後に Bean の名前を付けることはできませんでした。

9.4.5. Blueprint XML で PROPAGATION_NEVER ポリシーを使用したルートの例

上記のルートは、以下のように Blueprint XML で定義できます。

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <camelContext xmlns="http://camel.apache.org/schema/blueprint">
    <route>
      <from uri="file:src/data?noop=true" />
      <transacted />
      <bean ref="accountService" method="credit" />
      <transacted ref="PROPAGATION_NEVER" />
      <bean ref="accountService" method="debit" />
    </route>
  </camelContext>

</blueprint>
```

9.5. エラー処理およびロールバック

トランザクションルートでは標準の Apache Camel エラー処理技術を使用できますが、例外とトランザクション境界との間の相互作用を理解することが重要です。特に、出力された例外が原因で通常、トランザクションのロールバックが発生することを考慮する必要があります。以下のトピックを参照してください。

- 「トランザクションをロールバックする方法」
- 「デッドレターキューを定義する方法」
- 「トランザクションに関する例外のキャッチ」

9.5.1. トランザクションをロールバックする方法

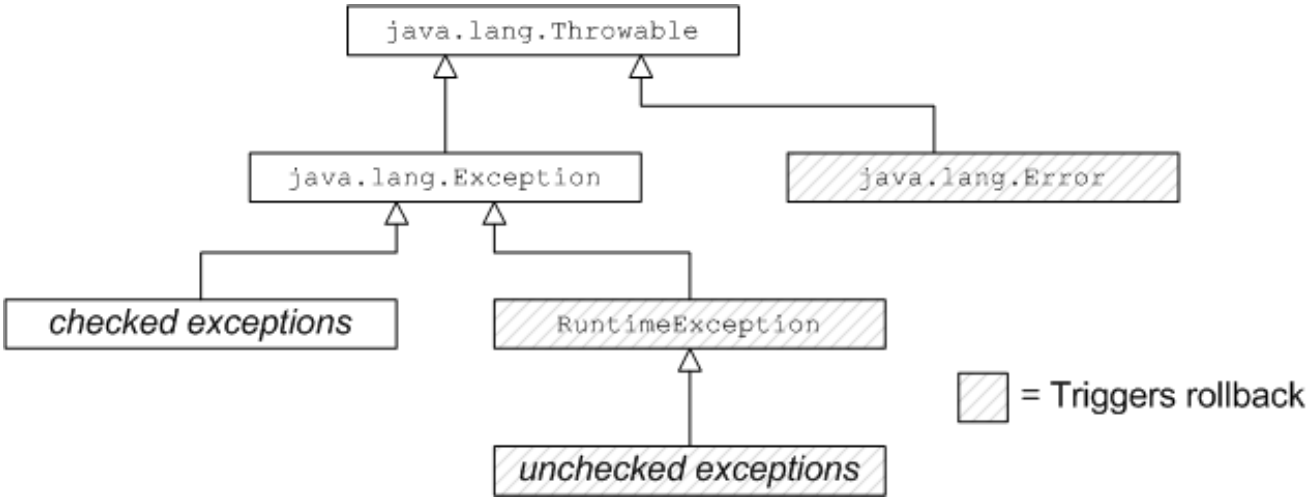
以下の方法のいずれかを使用してトランザクションをロールバックできます。

- 「ランタイム例外を使用したロールバックのトリガー」
- 「**rollback()** DSL コマンドの使用」
- 「**markRollbackOnly()** DSL コマンドの使用」

9.5.1.1. ランタイム例外を使用したロールバックのトリガー

Spring トランザクションをロールバックする最も一般的な方法は、ランタイム (チェックされていない) 例外を出力することです。つまり、例外は **java.lang.RuntimeException** のインスタンスまたはサブクラスです。**java.lang.Error** タイプの Java エラーも、トランザクションロールバックをトリガーします。一方、チェックされた例外では、ロールバックはトリガーされません。

次の図は、Java エラーと例外がトランザクションにどのように影響するかをまとめたもので、ロールバックをトリガーするクラスは灰色で網掛けされています。



注記

Spring フレームワークは、どの例外をロールバックするか、トリガーすべきでない XML アノテーションのシステムも提供します。詳細は、Spring Reference Guide の Rolling back を参照してください。

警告

ランタイム例外がトランザクション内で処理される場合、つまり、例外がトランザクションの境界を設定するコードに浸透する前に、トランザクションはロールバックされません。詳細は、「デッドレターキューを定義する方法」 を参照してください。

9.5.1.2. rollback() DSL コマンドの使用

トランザクションルートの途中でロールバックをトリガーする場合は、**rollback()** DSL コマンドを呼び出してこれを実行できます。これにより、**org.apache.camel.RollbackExchangeException** 例外が発生します。つまり、**rollback()** コマンドは、ランタイム例外を出力する標準的なアプローチを使用してロールバックをトリガーします。

たとえば、アカウントサービスアプリケーションで送金のサイズに絶対的な制限を設ける必要があると判断したとします。以下の例でコードを使用して、金額が 100 を超えるとロールバックをトリガーできます。

```
from("file:src/data?noop=true")
    .transacted()
    .bean("accountService","credit")
    .choice().when(xpath("/transaction/transfer[amount > 100]"))
        .rollback()
    .otherwise()
        .to("direct:txsmall");

from("direct:txsmall")
```

```
    .bean("accountService","debit")
    .bean("accountService","dumpTable")
    .to("file:target/messages/small");
```

注記

前述のルートでロールバックをトリガーすると、無限ループにトラップされます。この理由は、**rollback()** によって出力された **RollbackExchangeException** 例外がルートの開始時点で **file** エンドポイントに伝播するためです。File コンポーネントには、例外が出力された交換を再送する信頼性機能が組み込まれています。もちろん、再送すると、交換によって別のロールバックがトリガーされ、無限ループが発生してしまいます。以下の例は、この無限ループを回避する方法を示しています。

9.5.1.3. markRollbackOnly() DSL コマンドの使用

markRollbackOnly() DSL コマンドを使用すると、例外を出力せずに現在のトランザクションを強制的にロールバックできます。これは、「**rollback()** DSL コマンドの使用」 の例など、例外が発生したことで悪影響がある場合などに役立ちます。

以下の例は、**rollback()** コマンドを **markRollbackOnly()** コマンドに置き換えることで、「**rollback()** DSL コマンドの使用」 の例を変更する方法を示しています。このバージョンのルートは無限ループの問題を解決します。この場合、送金金額が 100 を超えると、現在のトランザクションはロールバックされますが、例外は出力されません。file エンドポイントは例外を受信しないため、エクスチェンジを再試行せず、失敗したトランザクションは暗黙的に破棄されます。

次のコードは、**markRollbackOnly()** コマンドで例外をロールバックします。

```
from("file:src/data?noop=true")
    .transacted()
    .bean("accountService","credit")
    .choice().when(xpath("/transaction/transfer[amount > 100]"))
        .markRollbackOnly()
    .otherwise()
        .to("direct:txsmall");
...
```

ただし、前述のルート実装は理想的ではありません。ルートはトランザクションを (データベースの一貫性を保ちながら) クリーンにロールバックし、無限ループに入らないようにしますが、失敗したトランザクションの記録は保持しません。実際のアプリケーションでは、通常、失敗したトランザクションを追跡する必要があります。たとえば、トランザクションが成功しなかった理由を説明するために、関連の顧客に通知を送ることができます。失敗したトランザクションを追跡する便利な方法は、ルートにデッドレターキューを追加することです。

9.5.2. デッドレターキューを定義する方法

失敗したトランザクションを追跡するには、**onException()** 句を定義すると、関連するエクスチェンジオブジェクトをデッドレターキューに迂回できます。ただし、トランザクションのコンテキストで使用する場合は、例外処理とトランザクション処理の間に対話がある可能性があるため、**onException()** 句を定義する方法に注意する必要があります。次の例は、再出力された例外を抑制する必要があると仮定して、**onException()** 句を適切に定義する方法を示しています。

```
// Java
import org.apache.camel.builder.RouteBuilder;
```

```
public class MyRouteBuilder extends RouteBuilder {
    ...
    public void configure() {
        onException(IllegalArgumentException.class)
            .maximumRedeliveries(1)
            .handled(true)
            .to("file:target/messages?fileName=deadLetters.xml&fileExist=Append")
            .markRollbackOnly();  // NB: Must come *after* the dead letter endpoint.

        from("file:src/data?noop=true")
            .transacted()
            .bean("accountService","credit")
            .bean("accountService","debit")
            .bean("accountService","dumpTable")
            .to("file:target/messages");
    }
}
```

上記の例では、**onException()** は **IllegalArgumentException** 例外をキャッチするよう設定され、問題のあるエクスチェンジをデッドレターファイル **deadLetters.xml** に送信します。もちろん、この定義を変更して、アプリケーションで発生するあらゆる種類の例外をキャッチすることができます。例外の再出力動作とトランザクションのロールバック動作は、**onException()** 句の次の特別な設定によって制御されます。

- **handled(true)** - 再出力された例外を抑制します。この特定の例では、再出力された例外は、File エンドポイントに伝播するときに無限ループをトリガーするため、望ましくありません。「**markRollbackOnly()** DSL コマンドの使用」 を参照してください。ただし、場合によっては、例外を再出力できる場合があります (たとえば、ルートの開始時のエンドポイントが再試行機能を実装していない場合)。

- **markRollbackOnly()** - 例外を出力せずに、現在のトランザクションをロールバック用にマークします。この DSL コマンドは、エクスチェンジをデッドレターキューにルーティングする **to()** コマンドの後に挿入する必要があることに注意してください。そうでない場合は、**markRollbackOnly()** が処理のチェーンを割り込みするため、エクスチェンジはデッドレターキューに到達しません。

9.5.3. トランザクションに関する例外のキャッチ

onException() を使用する代わりに、トランザクションルートで例外を処理する簡単な方法は、ルートに **doTry()** および **doCatch()** 句を使用することです。たとえば、以下のコードは、無限ループに陥ることなく、トランザクションルートで **IllegalArgumentException** をキャッチおよび処理する方法を示しています。

```
// Java
import org.apache.camel.builder.RouteBuilder;

public class MyRouteBuilder extends RouteBuilder {
    ...
    public void configure() {
        from("file:src/data?noop=true")
            .doTry()
                .to("direct:split")
            .doCatch(IllegalArgumentException.class)
                .to("file:target/messages?fileName=deadLetters.xml&fileExist=Append")
            .end();
```

```
        from("direct:split")
            .transacted()
            .bean("accountService","credit")
            .bean("accountService","debit")
            .bean("accountService","dumpTable")
            .to("file:target/messages");
    }
}
```

この例では、ルートは 2 つのセグメントに分割されます。最初のセグメント (**file:src/data** エンドポイントから) は受信エクスチェンジを受け取り、**doTry()** および **doCatch()** を使用して例外処理を実行します。2 つ目のセグメント (**direct:split** エンドポイントから) は、すべてのトランザクション処理を実行します。このトランザクションセグメント内で例外が発生すると、最初に **transacted()** コマンドに伝播され、それにより現在のトランザクションがロールバックされ、最初のルートセグメントの **doCatch()** 句によってキャッチされます。**doCatch()** 句は例外を再出力しないため、ファイルエンドポイントは再試行せず、無限ループが回避されます。