



JBoss Enterprise SOA Platform 5

ESB プログラマーガイド

JBoss 開発者向け

エディション 5.2.0

JBoss Enterprise SOA Platform 5 ESB プログラマーガイド

JBoss 開発者向け
エディション 5.2.0

JBoss ESB 開発チーム コミュニティの協力

Red Hat Documentation Group

法律上の通知

Copyright © 2011 Red Hat, Inc..

This document is licensed by Red Hat under the [Creative Commons Attribution-ShareAlike 3.0 Unported License](#). If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

概要

このガイドには、JBoss Enterprise SOA Platform を使用して開発するプログラマー向けの情報が記載されています。

目次

前書き	2
第1章 ESB (ENTERPRISE SERVICE BUS)	3
第2章 JBOSS ENTERPRISE SERVICE BUS	7
第3章 サービスとメッセージ	10
第4章 サービスの構築と使用	27
第5章 その他のコンポーネント	51
第6章 サンプル	53
第7章 高度なトピック	60
第8章 耐障害性と信頼性	79
第9章 サービス設定の定義	86
第10章 データのデコード - MIME デコーダー	115
第11章 WEB サービスサポート	118
第12章 事前定義されたアクション	119
第13章 カスタムアクションの開発	172
第14章 コネクターおよびアダプター	182
付録A 付録 A: JAXB ANNOTATION INTRODUCTION 設定の記述	189
付録B 添付資料 B: サービス指向アーキテクチャーの概要	191
付録C 参考資料	197
推奨資料	197
付録D GNU GENERAL PUBLIC LICENSE 2.0	198
付録E 改訂履歴	205

前書き

1. 謝辞

本書の一部は、『JBossESB 4.7 Programmers Guide』 copyright © 2010, 2011 JBoss Inc (<http://www.jboss.org/jbossesb>) に先に掲載されていました。

第1章 ESB (ENTERPRISE SERVICE BUS)

1.1. ESB (ENTERPRISE SERVICE BUS) とは？

本書は、金融機関、保険ビジネス、公共団体、通信業界などの大企業で **JBoss Enterprise SOA Platform** 向けのプログラムを行うソフトウェア開発者を対象としています。本書では、読者に **JBoss Enterprise SOA Platform** の概念を理解いただけるように、アプリケーション構築の詳細資料を提供し、主要コンポーネント関連の理論を説明しています。

ESB は、サービス指向アーキテクチャーの主要部となっており、ビジネスロジックではなく、インフラストラクチャーロジックを処理しています。ただし、SOA は単なる技術や製品ではなく、実際の技術とは関係のない多くの側面 (アーキテクチャー、方法、組織など) を持つ設計スタイルだと言えます。しかし、いずれは具体的な実装に対して抽象的な SOA をマップする必要があります。そのような時に ESB が必要となります。

1.2. 実際に ESB を使用できる場面とは？

JBoss Enterprise SOA Platform を役立たせることができる実例をいくつか以下に図で示します (これらは相互運用性のない JMS 実装を使用しているもの同士の通信に固有の例となりますが、その原理は汎用であり FTP や HTTP などの他のトランスポートにも適用が可能です)。

1 番目の図ではメッセージングキューがない 2 システム間のシンプルなファイル移動を示しています。

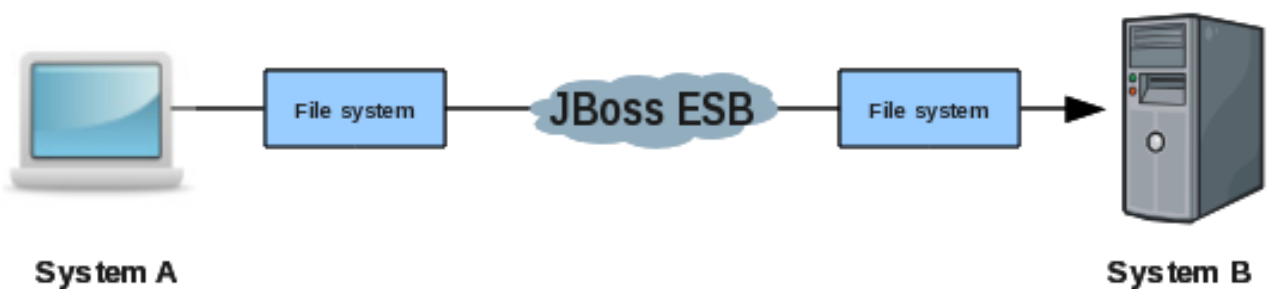


図1.1 シンプルなファイル移動

次の図では、同様のシナリオに変換のアクションを追加しています。

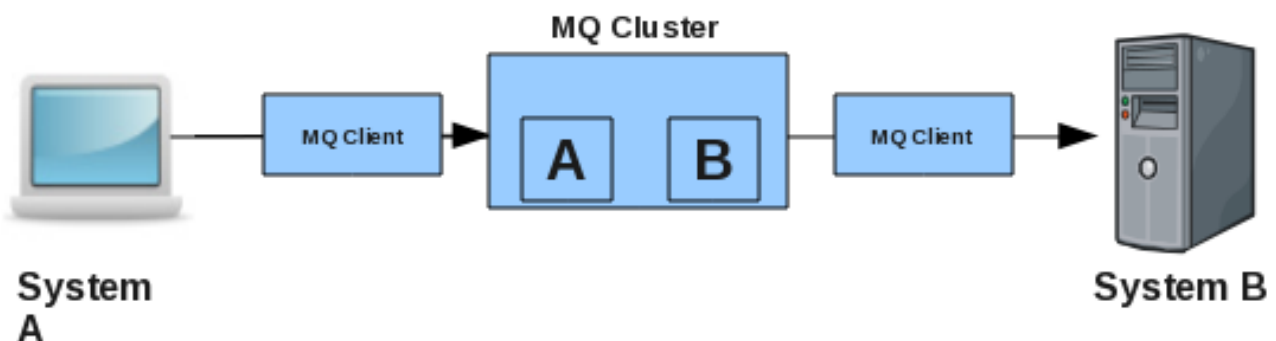


図1.2 変換とシンプルなファイル移動

キューも追加されました。

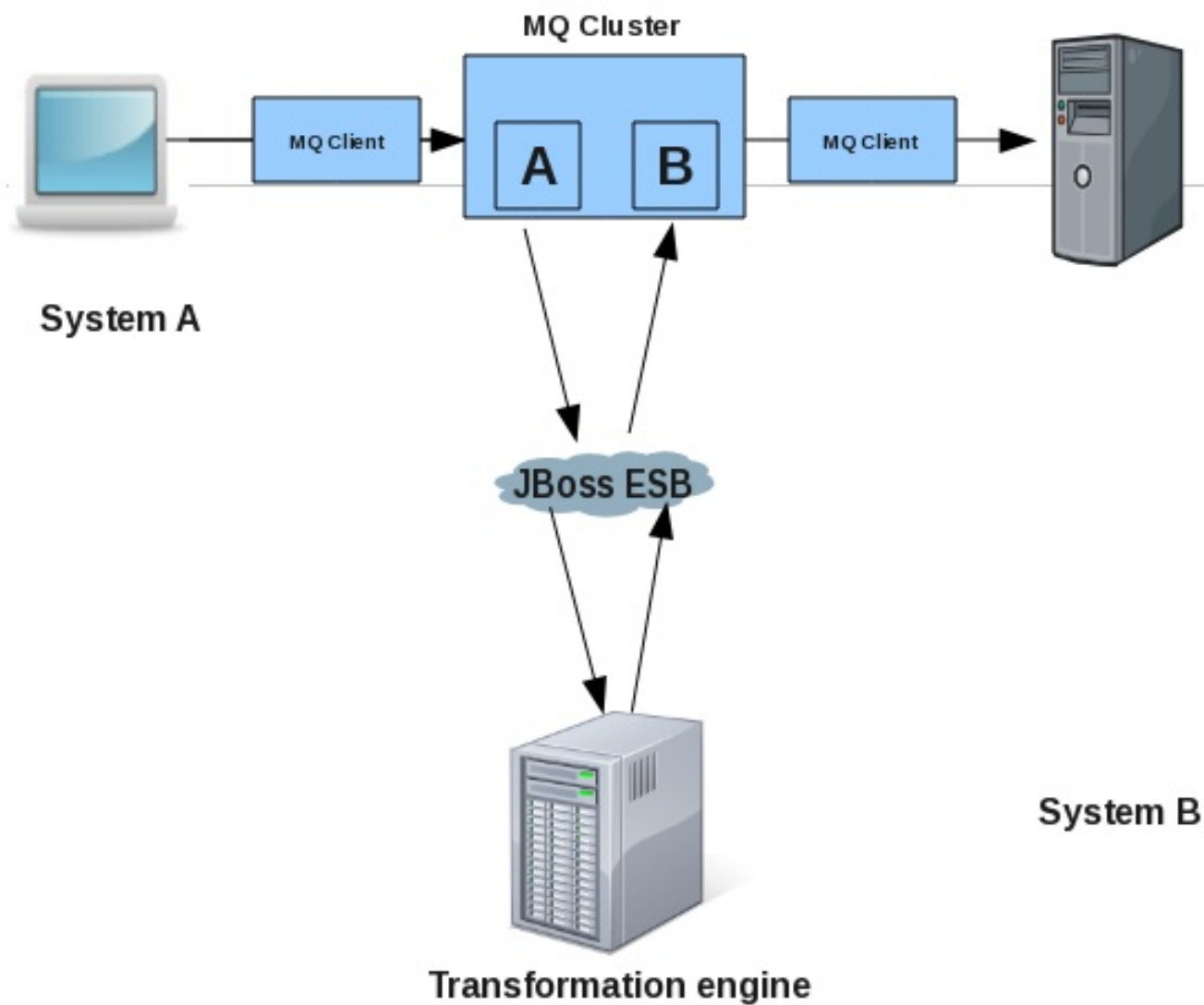


図1.3 変換とキューを使用する簡単なファイル移動

JBossESB は複数パーティのシナリオ以外でも使用できます。たとえば、以下の図ではファイルシステムを使った ESB 経由の基本的なデータ変換を示しています。

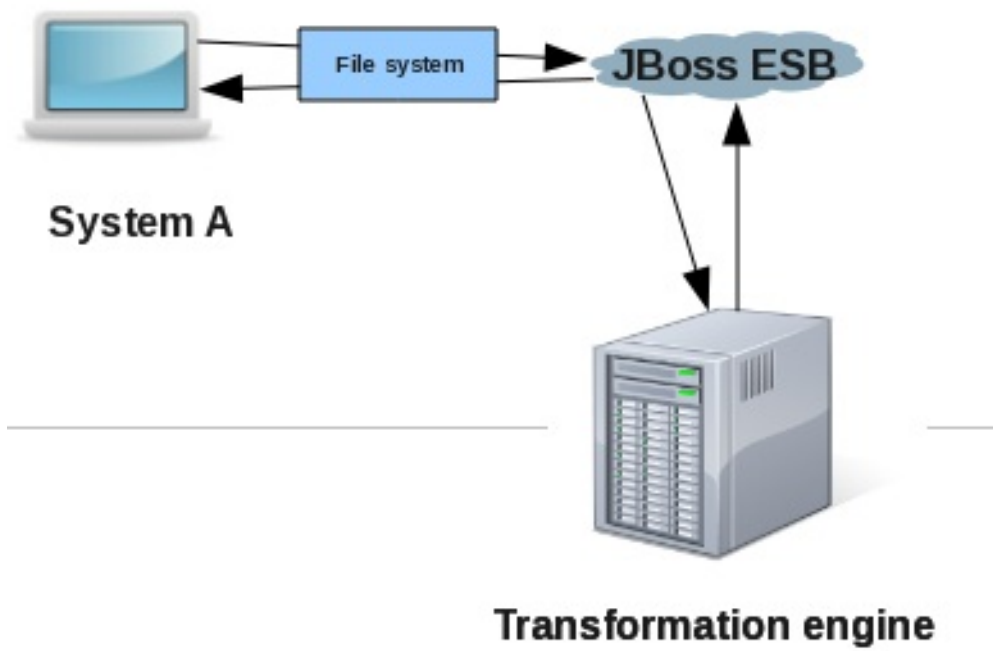


図1.4 基本的なデータ変換

最後のシナリオも変換アクションとキューシステムを使った単一パーティの例になります。

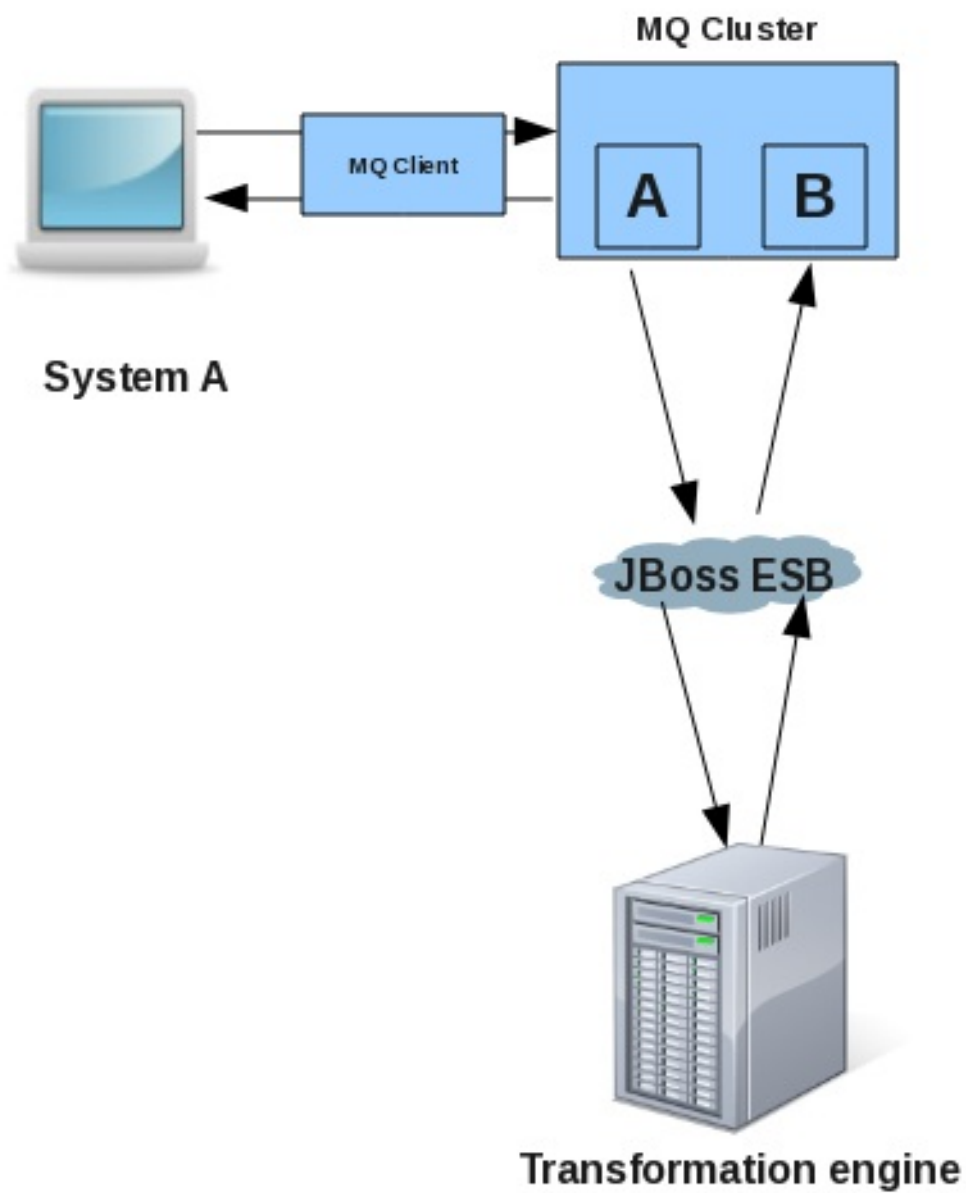


図1.5 変換とキュー

次の章では JBoss ESB のコンセプトについて、また SOA ベースのアプリケーション開発を行う場合にそのコンセプトをどのように使用できるかについて見ていきます。

第2章 JBOSS ENTERPRISE SERVICE BUS

2.1. ROSETTA

JBoss Enterprise SOA Platform の中心は、過去数年間、基幹サイトで商業的にデプロイされている ESB (Enterprise Service Bus) である *Rosetta* です。これらのデプロイメントには、高度に異機種混合な環境が含まれます。このような例として z/OS、DB2、および Oracle データベースが実行されている IBM メインフレーム、Windows および Linux サーバー、広範なサードパーティアプリケーション、企業の IT インフラストラクチャーの外部にあるサードパーティサービスなどがあります。



注記

以下の図では *プロセッサ* クラスは、イベントのトリガー時に処理を担当する *Rosetta* のコア内のアクションクラス群を参照します。

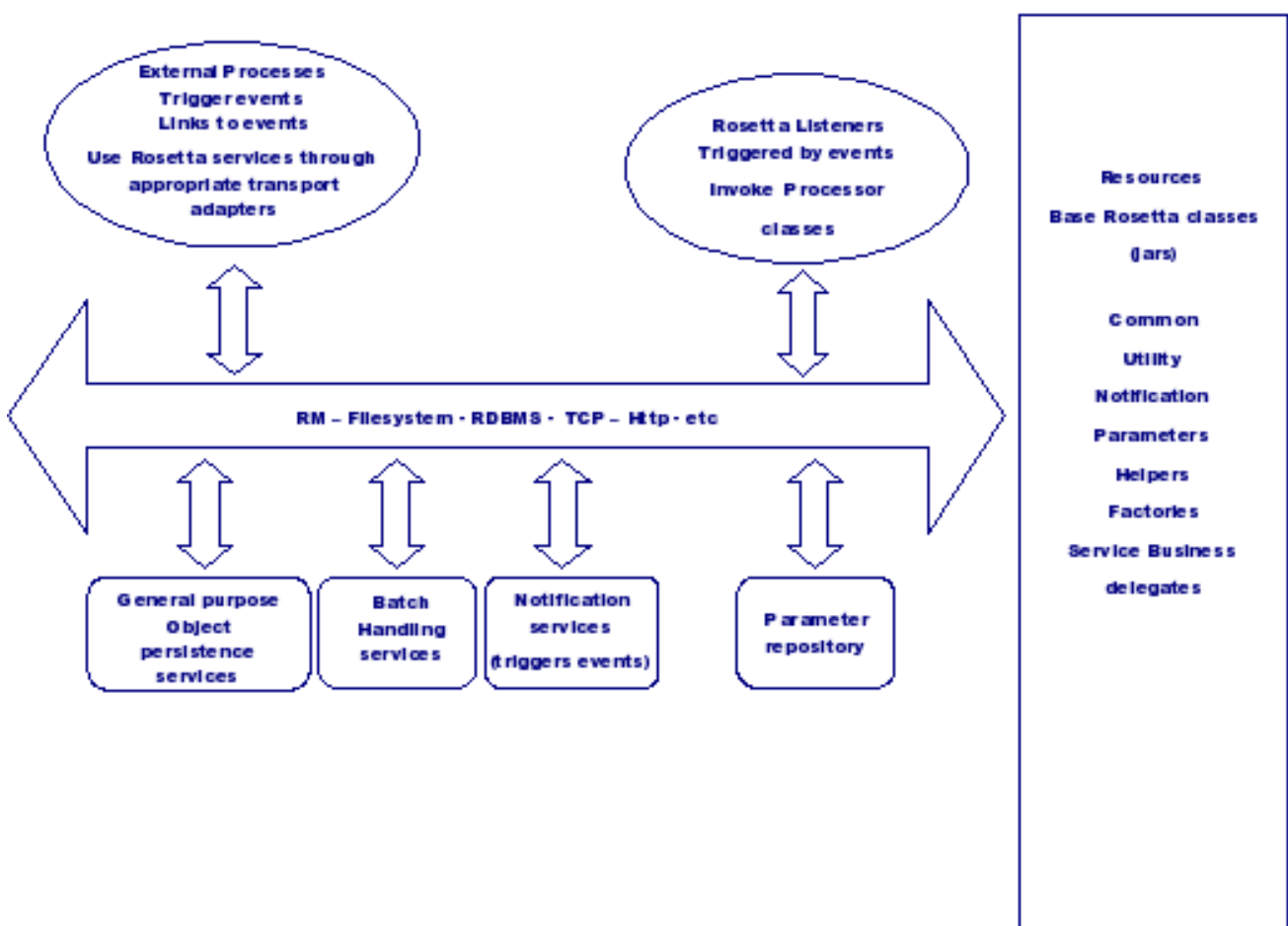
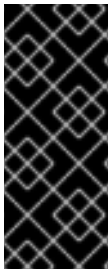


図2.1 Rosetta

様々な理由から、異種環境のアプリケーション、サービス、コンポーネントを相互運用させる場合もあります。例えば、新規デプロイメント内でレガシーシステムを活用するなどです。さらに、このようなエンティティ間のインターアクションは、*同期*または*非同期*で行われます。多くのエンタープライズサービスバス (ESB) では、*Rosetta* が開発されこのようなデプロイメントも容易化されました。さらに、以下の機能を持つツールセットやインフラストラクチャーを提供します。

- さまざまなトランスポートのメカニズムで動作させる設定が簡単に行えます (email や JMS など)。
- 汎用目的のオブジェクトリポジトリを提供します。

- プラグ可能なデータ変換のメカニズムを実現します。
- インタラクションのロギングに対応します。



重要

JBossESB ソース内には `org.jboss.internal.soa.esb` と `org.jboss.soa.esb` の 2 つのツリーがあります。コンテンツは予告なしに変更されるため、`org.jboss.internal.soa.esb` パッケージ内にあるものはすべてその使用を制限してください。`org.jboss.soa.esb` は、Red Hat の廃止予定ポリシーの範囲となります。

2.2. JBOSS ENTERPRISE SERVICE BUS のコア: 概要

Rosetta は 4 つのアーキテクチャコンポーネント上に構築されます。

1. メッセージリスナーとメッセージフィルタリングコード。メッセージリスナーは、受信メッセージをリスンするルーターとしての役割を果たします (JMS キューやトピック、ファイルシステム上にあるメッセージ)。その後、メッセージをフィルタリングして、別のメッセージエンドポイントにルーティングする (送信ルーター経由) 処理パイプラインに渡します。
2. **SmooksAction** プロセッサ経由のデータ変換コンポーネント (この件に関する詳細は、**サービスガイド**の「メッセージ変換」の章を参照)
3. コンテンツベースルーティングサービス (この件に関する詳細は、**サービスガイド**の「コンテンツベースルーティングとは」の章を参照)
4. ESB 内で交換されたメッセージやイベントを保存する際に使用するメッセージリポジトリ (この件に関する詳細は、**サービスガイド**の「レジストリとは」の章を参照)

これらの機能は本ガイドで後述されているビジネスクラス、アダプター、プロセッサのセットを通じて提供されます。クライアントとサービス間の通信は JMS やフラットファイルシステム、email などさまざまな手段で対応されます。

JBoss SOA Platform 5.1 には、JBoss Enterprise Data Services (Teiid ベース) が含まれています。EDS は、ESB サービスが JDBC ドライバーまたは Web サービスの形式のままコンシュームできるように、JDBC ドライバーまたは Web サービスとして公開されます。EDS Virtual Database (VDB) の JDBS 接続ストリングは、一般的な JDBC 接続ストリングとは少し違います。VDB の JDBC 接続ストリングの正しい形式は、`jdbc:teiid:vdb_name@mm://localhost:31000` です。

詳細は **EDS 開発ガイド**を参照してください。

以下は、一般的な **JBoss Enterprise Service Bus** デプロイメントに関する説明です。

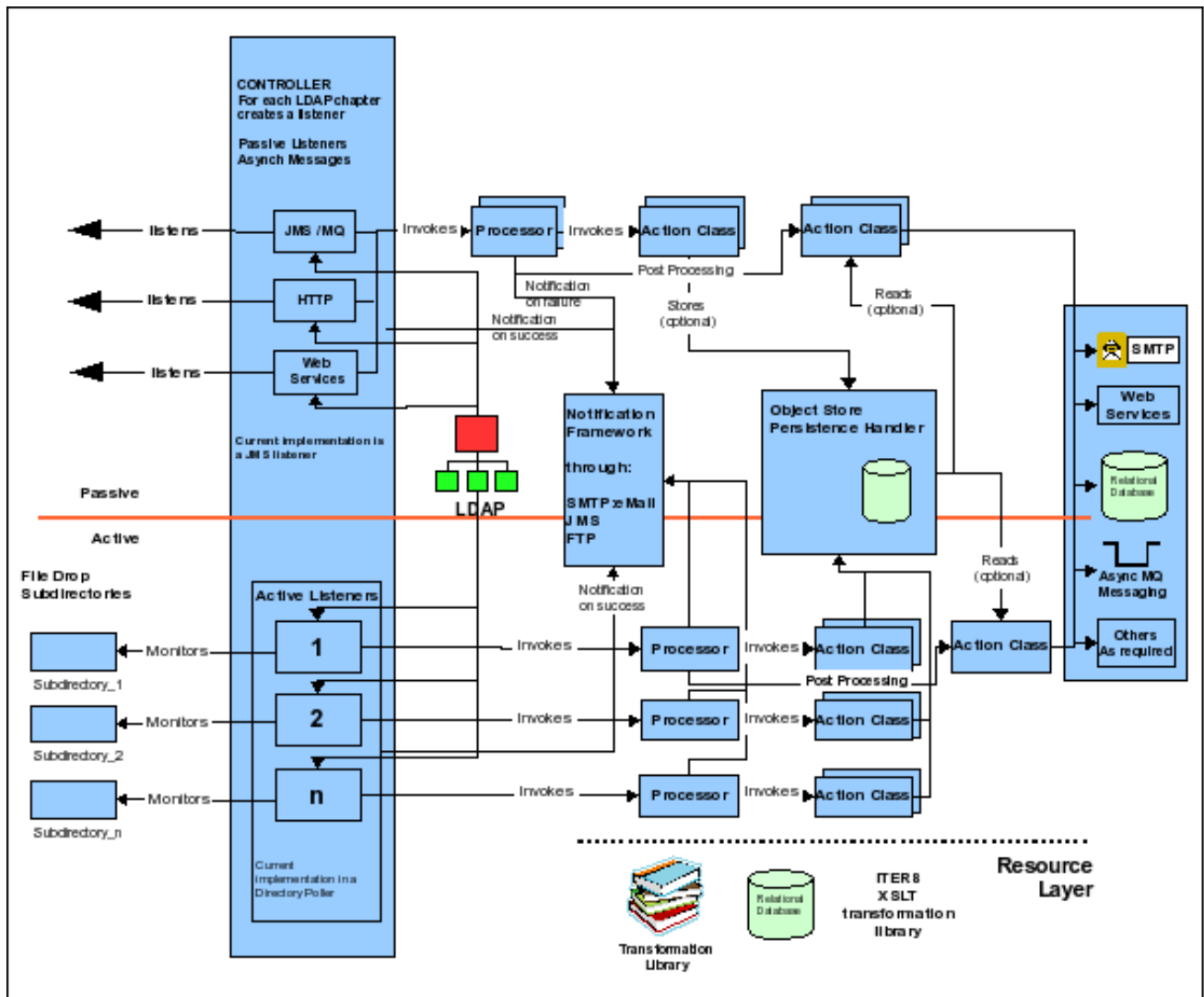


図2.2 一般的なデプロイメント



注記

(Lightweight Directory Access Protocol (LDAP) サーバーなど) 図にあるコンポーネントは設定オプションとなっています。そのため、初期状態では提供されておらず、上図のプロセッサおよびアクションの違いは単に着信イベント（メッセージ）が基盤となる ESB を起動して高度なレベルのサービスを呼び出す場合のコンセプトをわかりやすく説明するためのものです。

概要を理解したあとに、様々な **JBoss Enterprise Service Bus** コンポーネントや、やりとりの方法、SOA ベースアプリケーションの開発に使用する方法など詳細については、本書に後述されていますので、参照してください。

第3章 サービスとメッセージ

ここから、サービス指向アーキテクチャーの原理を保ちつつ、**JBoss Enterprise Service Bus** にあるものはすべてサービスまたはメッセージのいずれかと考えてください。コンセプト別に考えると、SOAP がどのように機能するかより早く理解することができます。

- サービスはそれぞれ、ビジネス論理や統合ポイントをレガシーシステムでカプセル化します。
- メッセージを使い、クライアントとサービスは互いに通信します。

次のセクションより、サービスとメッセージがどのようにサポートされるかを説明します。

3.1. サービス

JBoss Enterprise Service Bus では、サービスは「順番にメッセージを処理するアクションクラスの一覧として定義されます。

この定義が指すアクションクラスの一覧とは、アクションパイプラインのことです。

サービスは、リスナーの一覧を定義することもできます。**Listeners** は、**action pipeline** へメッセージを送るため、サービスのインバウンドルーターとして機能します。

下記は非常に簡単な設定です。ここでは、メッセージの内容をコンソールに出力する単一サービスを定義しています。

```
<?xml version = "1.0" encoding = "UTF-8"?>
<jbossesb
xmlns="http://anonsvn.labs.jboss.com/labs/jbossesb/trunk/product/etc/schem
as/xml/jbossesb-1.0.1.xsd" invmScope="GLOBAL">

<services>
  <service category="Retail" name="ShoeStore" description="Acme Shoe
Store Service">
    <actions>
      <action name="println"
class="org.jboss.soa.esb.actions.SystemPrintln" />
    </actions>
  </service>
</services>

</jbossesb>
```

サービスは **category** 属性と **name** 属性を持っています。JBoss ESB がサービスをデプロイする時、これらの属性を使用してサービスのリスナーをエンドポイントとしてサービスレジストリに登録します。クライアントは **ServiceInvoker** クラスを使用してサービスを呼び出すことができます。

```
ServiceInvoker invoker = new ServiceInvoker("Retail", "ShoeStore");
Message message = MessageFactory.getInstance().getMessage();

message.getBody().add("Hi there!");
invoker.deliverAsync(message);
```

この例では、**ServiceInvoker** は **Services Registry** を使用して **Retail:ShoeStore** サービスが使用できるエンドポイントのアドレスをルックアップします。また、クライアントから使用できるサービスエンドポイントの 1 つにメッセージを送る処理をすべて行います。メッセージのトランスポート

ト処理はクライアントに対して完全に透過的です。

エンドポイントアドレスが **ServiceInvoker** に渡されるタイミングは、サービスに設定されているリスナーの種類により左右されます。例えば、JMS、FTP、HTTP などの種類です (上記の例では、リスナーは定義されていませんが、**invmScope="GLOBAL"** を使うことで、サービスの *InVM listener* が有効になっています)。

InVM トランスポートは、Enterprise Service Bus の機能で、同じ *Java 仮想マシン (JVM)* 上で実行しているサービス同士の通信ができます。



重要

追加のエンドポイントを有効にするには、明示的にリスナーの設定をサービスに追加する必要があります。

2 種類のリスナー設定に対応しています。サポートされるリスナー設定:

- ゲートウェイリスナー

ゲートウェイリスナーの設定は、ゲートウェイエンドポイントを提供します。エンドポイントタイプは、ESB デプロイメント外部からメッセージのエントリポイントを提供します。また、サービスの **action pipeline** へ送る前にメッセージペイロードを *ESB Message* ヘラップし、メッセージペイロードを「正規化」します。

- ESB 対応リスナー

ESB 対応のリスナーは、*ESB 対応のエンドポイント*を提供します。これは、ESB 対応コンポーネント間で ESB メッセージを交換する際に使用するエンドポイントのタイプです(たとえば、ESB 対応リスナーを使い、Bus 上でメッセージの交換ができます)。

この例は、ShoeStore Service に JMS ゲートウェイリスナーが追加される方法を示しています。

```
<?xml version = "1.0" encoding = "UTF-8"?>
<jbossesb xmlns="http://anonsvn.labs.jboss.com/labs/jbossesb/

trunk/product/etc/schemas/xml/jbossesb-1.0.1.xsd">
<providers>
    <jms-provider name="JBossMQ" connection-factory="ConnectionFactory">
        <jms-bus busid="shoeStoreJMSGateway">
            <jms-message-filter dest-type="QUEUE" dest-
name="queue/shoeStoreJMSGateway"/>
        </jms-bus>
    </jms-provider>
</providers>

<services>
    <service category="Retail" name="ShoeStore" description="Acme Shoe
Store Service"

invmScope="GLOBAL">
<listeners>
    <jms-listener name="shoeStoreJMSGateway"
busidref="shoeStoreJMSGateway"
is-gateway="true"/>
</listeners>
```



```

        <actions>
            <action name="println"
class="org.jboss.soa.esb.actions.SystemPrintln" />
        </actions>
    </service>
</services>

</jbossesb>

```

上の例では、バスの <providers> セクションが設定に追加されたのが分かるでしょう。ここで、エンドポイントに対するトランスポートレベルの詳細を設定します。この場合、<jms-provider> セクションが追加されました。これは、Shoe Store JMS キューの <jms-bus> 1 つ定義するのが目的です。このバスは、Shoe Store Service で定義されている <jms-listener> 参照されます。この Shoe Store は InVM や JMS ゲートウェイエンドポイントを使い呼び出し可能となります (**ServiceInvoker** は、常にサービスのローカル InVM エンドポイントがあればそれを優先的に利用します)。

3.2. メッセージ

JBossESB 内のクライアントとサービス間の対話はすべてメッセージの交換によって発生します。疎結合を促進するため、メッセージ交換パターン (MEP) を使用した開発が推奨されます。要求と応答は、必要な場合にインフラストラクチャーやアプリケーションによって相互に関連付けられる、独立したメッセージでなければなりません。このようにして構築されたアプリケーションは耐障害性が高く、デプロイメントやメッセージ配信の要件に対して開発者はより柔軟に対応できるようになります。

サービスの疎結合や堅牢な SOA アプリケーションを確保するため、次のガイドラインが推奨されます。

1. 要求応答アーキテクチャーでなく一方向メッセージを使用します。
2. 交換されたメッセージ内で規定の定義を維持します。後で実装の変更が大変難しくなるため、バックエンド実装の選択を公開するサービスインターフェースを定義しないようにしてください。
3. メッセージペイロードに拡張可能なメッセージ構造を使用します。これにより、変更をバージョン化して後方互換性を維持することができます。
4. 極端に粒度が細かいサービスは開発しないようにしてください。これは、簡単に環境の変化に対応できないような非常に複雑なアプリケーションが必要になることが多いからです (SOA のパラダイムは分散オブジェクトではなく、1 つのサービスである点を忘れないでください)。

リクエストとレスポンスの一方向メッセージ配信パターンを使用している場合は、メッセージ内でのレスポンスの送信先に関する情報を常にエンコードするようにします。このような場合は、必須条件となっています。メッセージボディ (ペイロード) (アプリケーションが処理) にこの情報を置くか、最初のリクエストメッセージの部分に置きます (エンタープライズサービスバスのインフラストラクチャーが処理)。

ESB の中心がメッセージの概念で、この構造は SOAP の構造と似ています。

```

<xs:complexType name="Envelope">
    <xs:attribute ref="Header" use="required"/>
    <xs:attribute ref="Context" use="required"/>
    <xs:attribute ref="Body" use="required"/>
    <xs:attribute ref="Attachment" use="optional"/>
    <xs:attribute ref="Properties" use="optional"/>
    <xs:attribute ref="Fault" use="optional"/>
</xs:complexType>

```


以下に示すように図を用いてその基本的なメッセージの構造を表すことができます (本セクションの残りの部分で、これら各コンポーネントについて詳しく見ていくことにします)。

メッセージの構造は、UML でも表記できます。

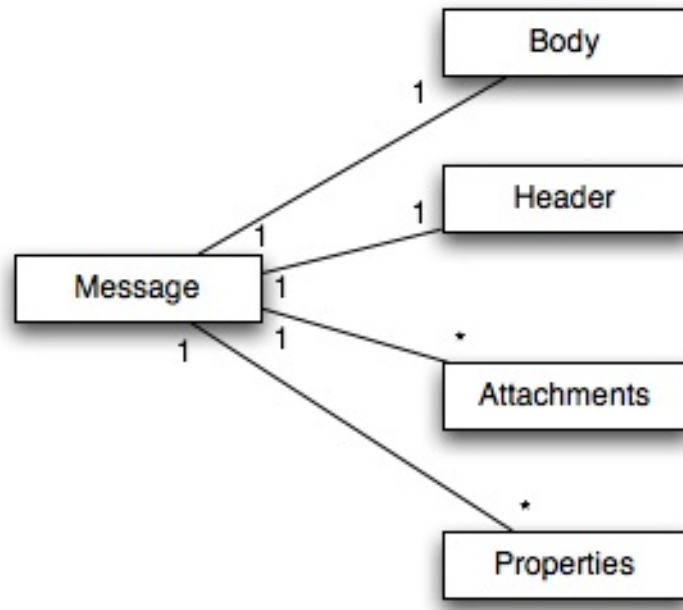


図3.1 メッセージの基本構造

各メッセージは `org.jboss.soa.esb.message.Message` インターフェースの実装です。このパッケージには、メッセージ内の各種フィールドのインターフェースが同梱されます。

```

public interface Message
{
    public Header getHeader ();
    public Context getContext ();
    public Body getBody ();
    public Fault getFault ();
    public Attachment getAttachment ();
    public URI getType ();
    public Properties getProperties ();

    public Message copy () throws Exception;
}

```



警告

JBossESB では、アタッチメントとプロパティもボディ同様に処理されます。これらの一般的な概念については現在再評価中で、今後のリリースで大幅に変更される可能性があります。

ヘッダーは、エンドポイント参照形式のメッセージルーティングとアドレスの情報で構成されています。また、ヘッダーには、メッセージを一意に特定する際に使用する情報も含まれています。JBoss

Enterprise Service Bus は、W3C 作成の WS-Addressing 規格ベースのアドレススキームを使用しています(`org.jboss.soa.esb.addressing.Call` クラスに関して学習するには、次章を参照してください)。

```
public interface Header
{
    public Call getCall ();
    public void setCall (Call call);
}
```

コンテキストには、トランザクションコンテキストやセキュリティコンテキストなどのセッション関係の情報が格納されます。JBossESB では、ユーザー拡張のコンテキストにも対応しています。



注記

JBoss Enterprise Service Bus のバージョン 5 からはじめて、ユーザー拡張のコンテキストに対応することになります。

ボディは一般的にメッセージの「ペイロード」を格納します。ボディを使用して任意数の異なるデータタイプを送信することができます (ボディ内における単一データ項目の送信や受信に制限はありません)。このようなオブジェクトがメッセージボディヘシリアライズされる方法やメッセージボディからシリアライズされる方法は、オブジェクトタイプによって異なります。

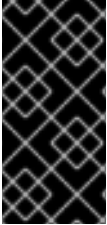


警告

メッセージのボディから/ヘシリアライズしたオブジェクトを送信する際は十分に注意してください。シリアライズ可能なものがすべて受信者側で意味があるわけではありません。例えば、データベース接続などです。

```
public interface Body
{
    public static final String DEFAULT_LOCATION =
        "org.jboss.soa.esb.message.defaultEntry";

    public void add (String name, Object value);
    public Object get (String name);
    public byte[] getContents();
    public void add (Object value);
    public Object get ();
    public Object remove (String name);
    public void replace (Body b);
    public void merge (Body b);
    public String[] getNames ();
}
```

重要

ボディのバイト配列コンポーネントは **JBoss ESB 4.2.1** で廃止されました。継続してバイト配列とボディ内に格納される他のデータを使用したい場合は、固有の名前に **add** を使用してください。クライアントとサービスに対して同じバイト配列の場所が必要な場合、**JBoss ESB** が使用する `ByteBody.BYTES_LOCATION` を使用できます。



警告

複数のサービスやアクションが互いのデータをオーバーライドしないよう、デフォルトの名前付きオブジェクト (**DEFAULT_LOCATION**) は注意して使用してください。

不良はエラー情報を伝達するために使用されます。情報はメッセージボディー内に表示されます。

```
public interface Fault
{
    public URI getCode ();
    public void setCode (URI code);

    public String getReason ();
    public void setReason (String reason);

    public Throwable getCause ();
    public void setCause (Throwable ex);
}
```



警告

JBossESB では、アタッチメントとプロパティもボディ同様に処理されます。これらの一般的な概念については現在再評価中で、今後のリリースで大幅に変更される可能性があります。

メッセージプロパティを使用して、追加のメタデータを定義します。以下にこれらのメッセージプロパティを提示しています。

```
public interface Properties
{
    public Object getProperty(String name);
    public Object getProperty(String name, Object defaultVal);

    public Object setProperty(String name, Object value);
    public Object remove(String name);
}
```



```

public int size();
public String[] getNames();
}

```

注記

java.util.Properties を使用する properties はまだ、**JBoss Enterprise Service Bus** には実装されていません。これは、利用可能なクライアントやサービスのタイプを制限するためです。同じ理由から、Web Services スタックもこれらのプロパティを実装しません。この制約に対応するには、現在の抽出に **java.util.Properties** を埋め込んでください。

メッセージには、メインのペイロードボディには表示されない添付 (イメージ、図、バイナリドキュメント形式、zip ファイルなど) が含まれることがあります。**Attachment** インターフェースは、名前の付いた添付と名前のない添付の両方をサポートします。JBossESB の現在のリリースでは、Java でシリアル化されたオブジェクトのみが添付として扱われます。今後のリリースではこの制限が解除される予定です

```

public interface Attachment
{
    Object get(String name);
    Object put(String name, Object value);

    Object remove(String name);

    String[] getNames();

    Object itemAt (int index) throws IndexOutOfBoundsException;
    Object removeItemAt (int index) throws IndexOutOfBoundsException
    Object replaceItemAt(int index, Object value)
    throws IndexOutOfBoundsException;

    void addItem (Object value);
    void addItemAt (int index, Object value)
        throws IndexOutOfBoundsException;
    public int getUnnamedCount();
    public int getNamedCount();
}

```

注記

添付を使用する理由はさまざまですが、メッセージに論理的な構成を提供するために使用するのが一般的です。エンドポイント間で添付のストリーミングができるようにすると、サイズの大きなメッセージのパフォーマンスを向上することができます。

注記

JBoss Enterprise Service Bus はメッセージや添付のストリーミングに他のエンコーディングメカニズムを指定できません。この機能は今後のリリースに追加され、添付を持つ SOAP の配信メカニズムと結合される予定です。現在、添付はボディ内の名前付きオブジェクトと同じ方法で処理されます。

ペイロードを挿入する場所を決定する際、アタッチメント、プロパティ、名前付きのオブジェクトのいずれを選べばいいのか圧倒されるかもしれません。しかし、選択の基準はいたって簡単です。

- 開発者は、クライアントがサービスとの対話を行うために使用するコントラクトを定義します。このコントラクトの一部で、サービスの機能、非機能部分を指定します。例えば、航空予約サービス (機能) やトランザクション関連のサービス (非機能) です。

また、開発者はサービスが理解できるオペレーション (メッセージ) も定義します。形式 (Java Serialized Message や XML) もメッセージ定義の一部として定義されます (この例ではトランザクションコンテキスト、座席番号、顧客名などです)。コンテンツを定義すると、サービスがペイロードがメッセージのどの部分にあるか分かるよう指定します (アタッチメントや、具体的な名前付きオブジェクト、またはデフォルトの名前付きオブジェクトなどの形式で行うことができます)。これはサービス開発者が決定します。唯一の制限事項は、オブジェクトとアタッチメントはグローバル意名でなければならない点です (そうでなければ、同じメッセージボディが複数の「Hop」で転送される場合、サービスやアクションは別のものに向けられた部分的なペイロードを誤って拾ってしまう場合があります)。

- サービスのコントラクト定義 (UDDI レジストリか、帯域外通信で) 取得でき、これでメッセージのどの部分にペイロードを置くかを定義します。それ以外の場所に置かれた情報は、ほぼ無視されるため、サービスが正しく実行されなくなります。

3.3. データの取得、設定

Enterprise Service Bus コンポーネントはすべて

(**actions**、**listeners**、**gateways**、**routers**、**notifiers**) はデフォルトでメッセージのデフォルトのペイロードの場所を使用して、メッセージ上にあるデータを "get"、"set" するよう設定されています。

すべての ESB コンポーネントは **MessagePayloadProxy** を使用してメッセージの get や set を管理します。このクラスは上記のようにデフォルトのケースに対応しますが、すべてのコンポーネントで一様に無効にすることもできます。以下のコンポーネントプロパティを使用して、メッセージペイロードの get や set の場所を一様に無効にすることができます。

- `get-payload-location`: メッセージペイロードを取得する場所。
- `set-payload-location`: メッセージペイロードを設定する場所。

注記

JBossESB 4.2.1GA 以前は、デフォルトのメッセージペイロードの交換パターンはありませんでした。今後のリリースでの後方互換については、以下の手順に従ってください。

1. **jbossesb.sar** に移動します。
2. **jbossesb-properties.xml** ファイルを開きます。
3. `core` セクションで、`use.legacy.message.payload.exchange.patterns` プロパティを **true** に設定します。

3.4. ボディの拡張

メッセージボディのコンテンツを操作するプロセスを簡素化するインターフェースが多数存在します。定義済みのメッセージ構造とメソッドを提供することで簡素化します。

これらのインターフェースは基本のボディインターフェースの拡張で、既存のクライアントやサービスと共に使用できます。メッセージの基礎となるデータ構造は変更されないため、メッセージのコンシューマーは新しいタイプを認識する必要はありません。

拡張タイプ

`org.jboss.soa.esb.message.body.content.TextBody`

ボディの内容が任意のストリングの場合これを使用します。`getText` メソッドや `setText` メソッドを使用して操作することができます。

`org.jboss.soa.esb.message.body.content.ObjectBody`

ボディの内容がシリアル化されたオブジェクトの場合はこれを使います。`getObject` メソッドや `setObject` メソッドを使用して操作することができます。

`org.jboss.soa.esb.message.body.content.MapBody`

ボディの内容がマップ (ストリング、シリアル化されている) の場合これを使用します。`setMap` メソッドやその他のメソッドを使用して操作することができます。

`org.jboss.soa.esb.message.body.content.BytesBody`

ボディの内容が任意の Java データタイプを含むバイトストリームの場合これを使用します。データタイプのメソッドを使用して操作することができます。`BytesMessage` が作成されると、操作の必要に応じて読み取り専用モードまたは書き込み専用モード内に置かれます。`readMode()` メソッドや `writeMode()` メソッドを使用してモードを変更することができますが、モードが変更される度にバッファーポイントがリセットされます。`flush()` メソッドを呼び出して、すべての更新がボディに適応されたことを確認する必要があります。

`XMLMessageFactory` クラスや `SerializedMessageFactory` クラスを使用すると、これらのインターフェースを基にしたボディの実装を持つメッセージを作成することができます。メッセージに対して作業を行う場合、`MessageFactory` クラスや `MessageFactory` に関連したクラスを使用するより、`XMLMessageFactory` クラスや `SerializedMessageFactory` クラスを使用した方が便利です。

`createTextBody` など、ボディの各タイプに関連する `create` メソッドが存在します。このメソッドによって、特定タイプのメッセージを作成し初期化することができます。メッセージの作成後、ローボディまたはインターフェースメソッドを使用して直接メッセージを操作することができます。ボディの構造は送信後も維持されるため、メッセージを作成したインターフェースのメソッドを使用すればメッセージの受信側が操作することもできます。



注記

ベースのボディインターフェースへの拡張は任意でオリジナルのボディに追加することができます。このように、既存のクライアントとサービスと連携して使用することができます。メッセージ内の基盤のデータ構造が変更されないため、必要であればメッセージコンシューマーは、これらの新しいタイプは認識しないままにすることができます。重要なのは、この拡張はデフォルトの場所にデータを格納しない点です。データは、拡張インスタンス上にある該当のゲッターを使用してリトリブする必要があります。

3.5. メッセージヘッダー

メッセージヘッダーの内容は `org.jboss.soa.esb.addressing.Call` クラスのインスタンスに格納されます。


```

public class Call
{
    public Call ();
    public Call (EPR epr);
    public Call (Call copy);
    public void setTo (EPR epr);
    public EPR getTo () throws URISyntaxException;

    public void setFrom (EPR from);
    public EPR getFrom () throws URISyntaxException;

    public void setReplyTo (EPR replyTo);
    public EPR getReplyTo () throws URISyntaxException;

    public void setFaultTo (EPR uri);
    public EPR getFaultTo () throws URISyntaxException;

    public void setRelatesTo (URI uri);
    public URI getRelatesTo () throws URISyntaxException;
    public void copy();
    public void setAction (URI uri);
    public URI getAction () throws URISyntaxException;
    public final boolean empty();
    public void setMessageID (URI uri);
    public URI getMessageID () throws URISyntaxException;
    public String toString();
    public String stringForm();
    public boolean valid();
    public void copy (Call from);
}

```

org.jboss.soa.esb.addressing.Call は一方向パターンと要求返答対話パターンの両方をサポートします。

表3.1 org.jboss.soa.esb.addressing.Call プロパティ

プロパティ	タイプ	必須	説明
To	EPR	Yes	メッセージ受信側のアドレス
From	EPR	No	メッセージ発信元のエンドポイント
Reply To	EPR	No	このメッセージに返信する受信側を特定するエンドポイント参照
FaultTo	EPR	No	不良の警告の受信側を識別するエンドポイント参照
Action	URI	Yes	これは、メッセージが暗示するセマンティクスを一意かつ不透明に識別します。

プロパティ	タイプ	必須	説明
MessageID	URI	場合による	時空間でこのメッセージを一意に識別する URI。別のアプリケーションに使う予定のメッセージは [MessageID] を共有することはできません。メッセージは、通信エラーなどが原因で再送信され、同じ [MessageID] プロパティを使用する可能性があります。このプロパティの値は、不透明な URI で、等価を超えた解釈については定義されません。返信が来るものについては、このプロパティを必ず設置する必要があります。

ヘッダーと各種エンドポイント参照の関係は以下のように表記できます。

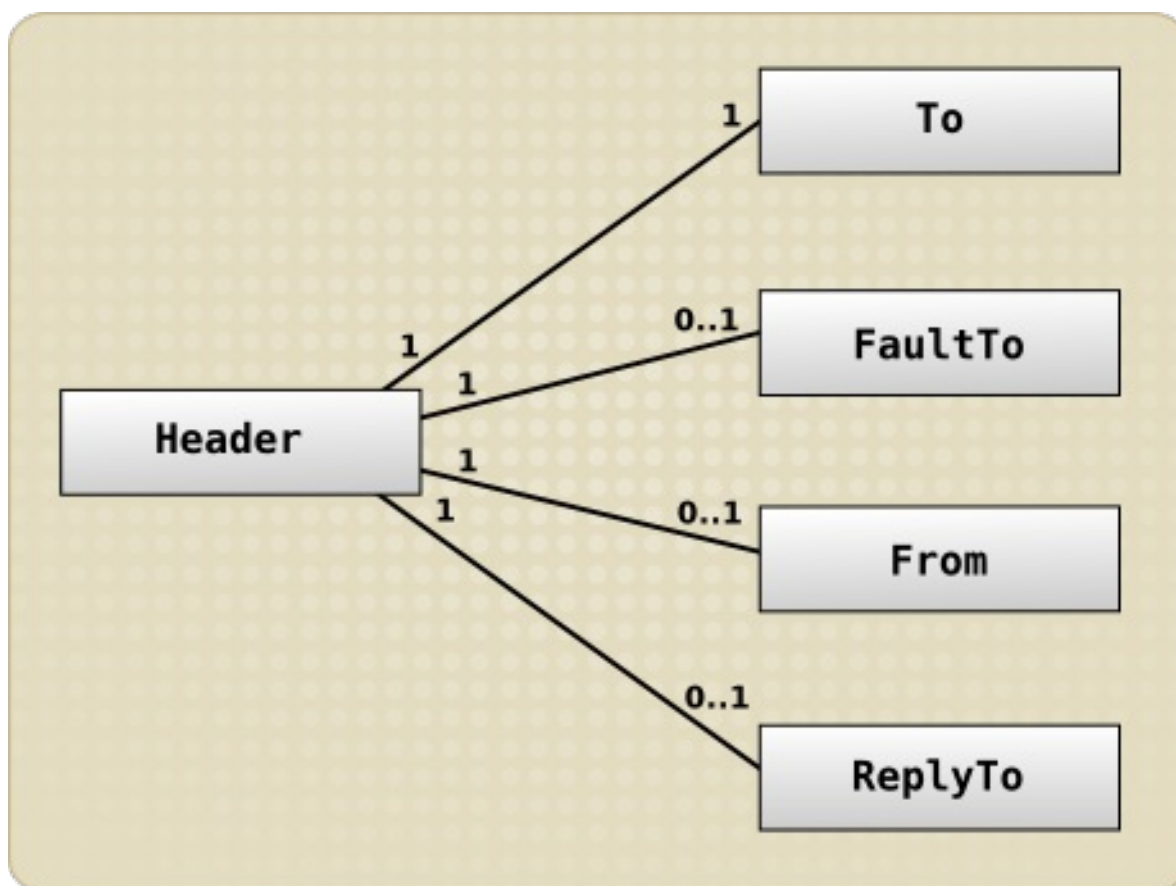


図3.2 UML で表記されたヘッダーとエンドポイント参照の関係

サービスを開発し使用する際、ヘッダーの役割について考慮しなければなりません。たとえば、要求と応答に基づいた同期の対話パターンが必要な場合、ReplyTo フィールドの設定が求められます。設定がない場合はデフォルトのエンドポイント参照が使用されます。要求／応答の場合でも、指定があれば応答は元の送信側に戻る必要はありません。一方方向メッセージ (応答なし) を送信する場合、ReplyTo フィールドを設定しても無視されるため、設定しないようにしてください。



注記

詳細については、LogicalEPR の章を参照してください。



警告

EPR の内部形式は API 実装固有のものであるため、これに依存すべきではありません。この形式が同じ形でとどまるという保証はありません。



注記

メッセージヘッダーは、メッセージと連携して構成されます。エンドポイント間で配信されると、このメッセージヘッダーは不変となります。インターフェースにより、受信側が個別の値を変更できるようになりますが、**JBoss Enterprise Service Bus** はこれらの変更については無視します (今後のリリースでは、明確化を目的に、アプリケーションプログラミングインターフェースでこのような変更が行えないようになるはずです)。このルールについては、**WS-Addressing** 規格にまとめられています。

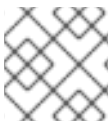
3.6. LOGICALEPR

ReplyTo や FaultTo EPR は、物理 EPR (JMS-EPR など) ではなく、常に論理 EPR を使用するようにしてください。**LogicalEPR** とは、ESB サービス／エンドポイントの名前とカテゴリを指定する EPR のことです。LogicalEPR には物理アドレス指定の情報は含まれません。

LogicalEPR は EPR ユーザー (通常 ESB ですが、そうとは限りません) を想定しないため、LogicalEPR の選択が奨励されます。**LogicalEPR** のクライアントは EPR で提供されるサービス名やカテゴリの詳細を使用して、**呼び出し**が行われる時に (適切なアドレス指定情報を取得する場合など) サービスやエンドポイントに対する物理エンドポイントの詳細をルックアップできます。クライアントは適切な物理エンドポイントタイプを選択することもできます。

3.7. デフォルトの FAULTTO

FaultTo は、不良関連のメッセージの送信先を識別するエンドポイント参照です。



注記

LogicalEPR の詳細を参照してください。

3.8. デフォルトの REPLYTO

JBoss Enterprise Service Bus では、対話パターンは一方方向メッセージ交換をベースとするように推奨していました。メッセージは、自動的にレスポンスを受信しない場合があります (送信者がレスポンスを受け取るかについてはアプリケーションにより左右されます)。このように、返信先アドレス (エンドポイント参照) は、ヘッダーのルーティング情報のうちオプションの情報で、必要に応じてアプリケーションはこの値を設定するようにしてください。しかし、レスポンスが必要で、エンドポイント参照 (*ReplyTo* EPR) が設定されていない場合、**JBoss Enterprise Service Bus** はトランスポートの各タイプにデフォルト値を提供できます (**ReplyTo** のデフォルトを使用するには、システム管理者は **JBoss Enterprise Service Bus** をデフォルト使用するように設定する必要があります)。

表3.2 トランスポートによるデフォルトの ReplyTo

トランスポート	ReplyTo
JMS	元のリクエストの配信に使用した名前と同じものを持つキュー。接尾辞は _reply です。
JDBC	元の要求の配信に使用した名前を持つ同じデータベース内にあるテーブル。接尾辞は _reply_table (応答テーブルには、要求テーブルと同じ列定義が必要となります)。
ファイル	ローカルファイルとリモートファイル共に管理的な変更は必要ありません。元の送信側のみが応答を受け取るようにするため、応答は固有のサフィックスで要求と同じディレクトリに書き込まれます。

3.9. メッセージペイロード

メッセージペイロードはボディ、添付、プロパティの組み合わせになります。



警告

JBossESB では、アタッチメントとプロパティもボディ同様に処理されます。これらの一般的な概念については現在再評価中で、今後のリリースで大幅に変更される可能性があります。

ペイロードの UML 表記を以下に示します。

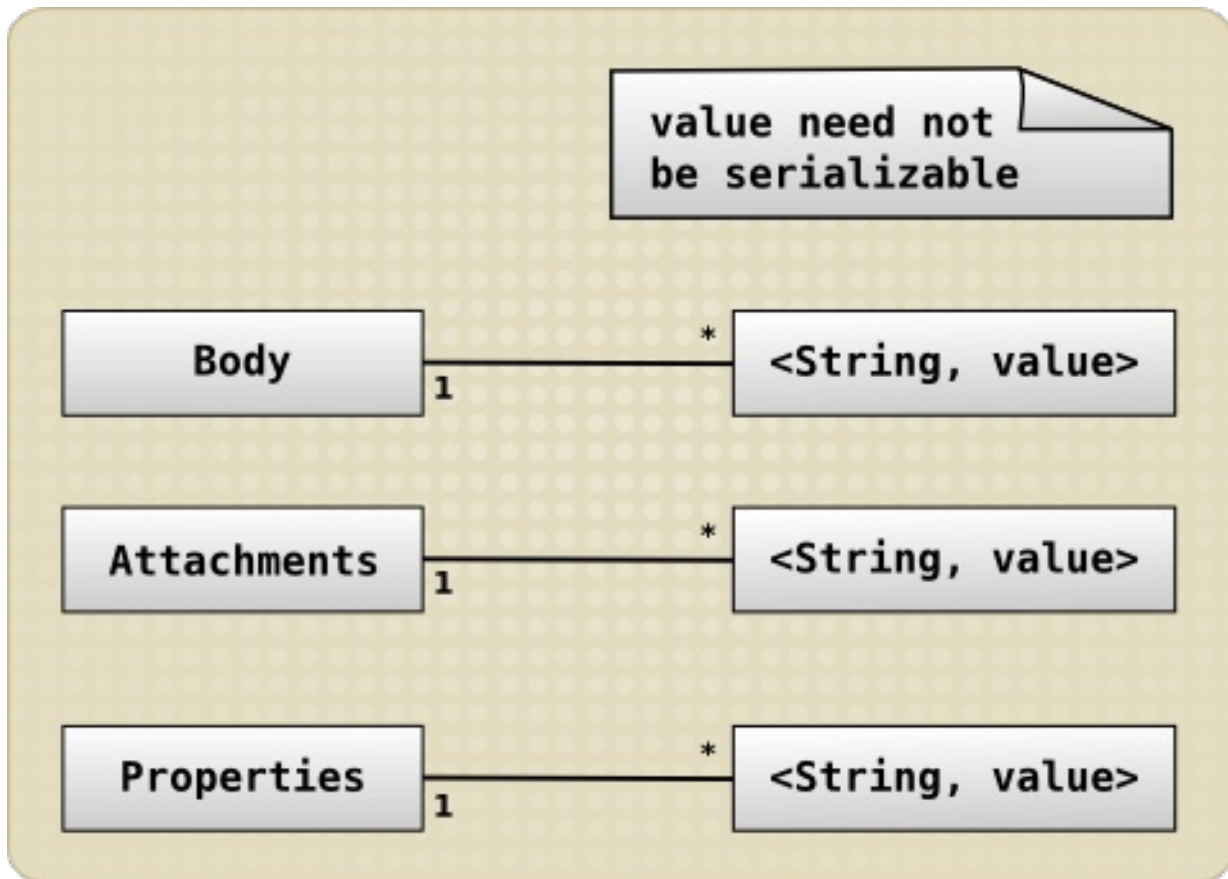


図3.3 メッセージペイロードの UML 表記

より複雑なコンテンツを適用するには、名前付きオブジェクトに対応している **add** メソッドを使用します。粒度の細かくデータにアクセスできるように、名前付きのオブジェクトペアを使用します。どのタイプのオブジェクトでもメッセージボディに追加できます (Java シリアライズ可能でないオブジェクトを追加するには、**JBoss Enterprise Service Bus** にメッセージのマーシャル化とアンマーシャル化機能を追加する必要があります。このプロセスの詳細については、「メッセージフォーマット」のセクションを参照してください)。



注記

"setting" や "getting" に名前が提示されていない場合は、DEFAULT_LOCATION 設定に定義されているものを使用します。



注記

メッセージ内でシリアライズされた Java オブジェクトを使用する際は、サービス実装に制約を加えるため、注意してください。

メッセージボディに「名前付きオブジェクト」を使用するのが最も簡単でしょう。ボディ全体をデコードしなくてもメッセージペイロード内で各データ項目を追加、削除、検査することができます。また、ペイロード内の名前付きオブジェクトとバイト配列を組み合わせることも可能です。



注記

現在、Java シリアライズオブジェクトしか添付することができません。この制約は、今後のリリースではなくなる予定です。

**注記**

現在のリリースでは、Java シリアライズオブジェクトしか添付することができません。この制約は、今後のリリースではなくなる予定です。

3.10. MESSAGEFACTORY

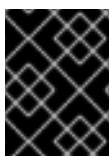
各 ESB コンポーネントは ESB メッセージを Java オブジェクトの集合として処理しますが、多くの場合で ESB メッセージをシリアライズする必要があります。データストアへの保存や異なる JBossESB プロセス間でのメッセージ送信、デバッグなどの場合に ESB メッセージをシリアライズする必要があります。

正規化形式の要件は、各 ESB デプロイメント独自の特性に左右されるため、**JBoss Enterprise Service Bus** はメッセージに対して単一の正規化形式を指定することはありません。`org.jboss.soa.esb.message.Message` インターフェースの実装はすべて `org.jboss.soa.esb.message.format.MessageFactory` クラスから取得されます。

```
public abstract class MessageFactory
{
    public abstract Message getMessage ();
    public abstract Message getMessage (URI type);
    public abstract void reset();
    public static MessageFactory getInstance ();
}
```

メッセージシリアライゼーションの実装は URI によって一意に識別されます。新しいインスタンスを作成する時に実装を指定するか、設定されているデフォルトを使用します。

- `MessageType.JBOSS_XML`: これは、ワイヤー上にあるメッセージの XML 表現を使用します。メッセージのスキーマは、`message.xsd` ファイルで定義します (`schemas` ディレクトリ内にあります)。URI は `urn:jboss/esb/message/type/JBOSS_XML` です。
- `MessageType.JAVA_SERIALIZED`: この実装では、メッセージの全コンポーネントをシリアライズできます。また、このメッセージタイプの受信側には、デシリアライズするための十分な情報 (Java クラス) が必要なのは明らかなです。URI は `urn:jboss/esb/message/type/JAVA_SERIALIZED` です。

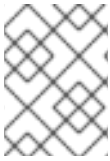
**重要**

アプリケーションが簡単に特定のサービス実装に結合されてしまうため、メッセージ形式の `JBOSS_SERIALIZED` バージョンの使用には細心の注意を払ってください。

他のメッセージ実装をランタイム時に提供するには、`org.jboss.soa.esb.message.format.MessagePlugin` を使用します。

```
public interface MessagePlugin
{
    public static final String MESSAGE_PLUGIN =
        "org.jboss.soa.esb.message.format.plugin";
    public Object createBodyType(Message msg, String type);
    public Message getMessage ();
    public URI getType ();
}
```


各プラグインは `getType()` メソッドを使用して、提供するメッセージ実装のタイプを一意に識別しなければなりません。プラグインの実装は、`org.jboss.soa.esb.message.format.plugin` 拡張の付いたプロパティ名を使って `jbossesb-properties.xml` ファイル内でシステムに対し識別されなければなりません。



注記

デフォルトのメッセージタイプは、**JBoss_XML** です。これを変更するには、`org.jboss.soa.esb.message.default.uri` プロパティを任意の URI 名に設定します。

3.11. メッセージのフォーマット

シリアル化メッセージフォーマットには、JBoss_XML と JBoss_SERIALIZED の 2 種類あります。詳細は本項を参照してください。

MessageType.JAVA_SERIALIZED

この実装では、メッセージの全コンポーネントがシリアル化可能でなければなりません。このタイプのメッセージの受信側は、メッセージをデシリアル化できなければなりません。よって、メッセージ内に格納された Java クラスをインスタンス化できなければならないことになります。

この実装では、すべての内容が *Java* でシリアル化できなければなりません。メッセージにシリアル化可能でないオブジェクトを追加しようとすると、**IllegalArgumentException** がスローされます。

URI は `urn:jboss/esb/message/type/JAVA_SERIALIZED` です。

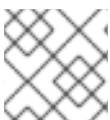
MessageType.JBoss_XML

この実装は、ワイヤー上にあるメッセージの XML 表現を使用します。メッセージのスキーマは、`message.xsd` ファイルで定義されます (`schemas` ディレクトリにあります)。任意オブジェクトをメッセージに追加できます。つまり、シリアル化する必要はありません。そのため、メッセージのシリアル化が必要な場合、このようなオブジェクトを XML にマーシャリングまたは、アンマーシャリングできるようにする必要があります。`org.jboss.soa.esb.message.format.xml.marshal.MarshalUnmarshalPlugin` から設定できます。

```
public interface MarshalUnmarshalPlugin
{
    public static final String MARSHAL_UNMARSHAL_PLUGIN =
        "org.jboss.soa.esb.message.format.xml.plugin";
    public boolean canPack(final Object value);
    public boolean marshal (Element doc, Object param)
        throws MarshalException;

    public Object unmarshal (Element doc) throws UnmarshalException;

    public URI type ();
}
```



注記

デフォルトで、Java シリアル化オブジェクトはサポートされています。

プラグインのマーシャリングは、**jbossesb-properties.xml** 設定ファイルよりシステムに登録しなければなりません。プラグインは MARSHAL_UNMARSHAL_PLUGIN で始まる属性名を持たなければなりません。

XML でオブジェクトをパッキングする時、**JBoss Enterprise Service Bus** はそのオブジェクトタイプを処理できるプラグインが見つかるまで、登録されたプラグインのリスト内を検索します (適切なプラグインが見つからない場合は、不良メッセージを返します)。メッセージの受信側でのアンパッキングを容易にするため、オブジェクトをパッキングしたプラグインの名前も添付されます。

第4章 サービスの構築と使用

4.1. LISTENER、NOTIFIER/ROUTER、および ACTION

4.1.1. Listener

リスナーは ESB 認識のメッセージ受信用エンドポイントをカプセル化します。メッセージの受信で、**listener** は **replyTo** エンドポイントにその結果をルーティングする前にメッセージを処理するメッセージプロセッサの「パイプライン」にメッセージをフィードします。**pipeline** 内で行われるアクション処理はいくつかのステップで構成されることがあります。ここでは、特定のプロセッサにメッセージが変換され、何らかのビジネスロジックが次のプロセッサで適用されてから、その結果が **pipeline** 内の次のステップまたは別のエンドポイントにルーティングされます。



注記

リスナーに対して様々なパラメーターを設定できます。例としては、有効なワークスレッドの数などです。これらのオプションの種類については、"Configuration" のセクションを参照してください。

4.1.2. Router

ルーターを使用して、メッセージやペイロードをエンドポイントに転送します。**router** によっては、unwrap プロパティに対応するものもあります。このプロパティを **true** に設定すると、自動的にメッセージペイロードを抽出して、次の ESB 未対応のエンドポイントに送信します。このオプションを **false** に設定すると、ペイロードを抽出せずに、ESB メッセージをすべて渡します (後者の場合、受け取り側のエンドポイントは ESB 対応であるため、メッセージの処理が可能です)。

action pipeline は、設定内で待機しているアクションがあったとしても、**router** 操作が実行されると、メッセージを処理しなくなります。このような種類の分割が必要な場合は、**StaticWiretap** アクションを使用します。

StaticWiretap や **StaticRouter** などのツールは、他のサービスへのルーティングにのみ使用することができます。メッセージコンテンツをベースにしたアクションで、動的にルーティングができるプログラムもあります。コンテンツベースルーターの詳細については、サービスガイドの「コンテンツベースルーティングとは」の章を参照してください。



注記

様々な種類のルーターの使用については、「事前定義済みアクション」の章の「ルーター」の項目を参照してください。

4.1.3. Notifier

成功またはエラーの情報を ESB 認識しないエンドポイントに伝播させることができる方法が **notifiers** になります。ESB 認識のエンドポイントとの通信には **notifiers** を使用しないでください。ESB 認識のエンドポイントと ESB を認識しないエンドポイントに同じチャンネルでのリスンを行わせることはできません。ESB 認識のエンドポイントと通信する場合はアクション内で **couriers** か **ServiceInvoker** を使用することを考慮してください。

ESB 認識のトランスポートがすべて **notifiers** に対してサポートされているわけではありません (また、その逆も同様)。notifiers は意図的に非常にシンプルになっています。byte[] または String のいずれか 1 つのみをトランスポートすることができます (ペイロードで **toString()** を呼び出すことによ

り取得)。

上記に概説したように、**listener** の役割とはメッセージ配信のエンドポイントとして動作し **action pipeline** にメッセージを配信することになります。各 **listener** の設定は次に記す情報を提供する必要があります。

- **registry** (**service-category**、**service-name**、**service-description**、および **EPR-description** タグ名)。オプションの **remove-old-service** タグ名を **true** に設定すると、Enterprise Service Bus はこの新しいインスタンスを追加する前に **registry** から既存のすべてのサービスエントリを削除します。ただし、すべての エンドポイント参照 (EPR) を含むサービス全体が削除されるため注意してください。
- **listener** クラスのインスタンス化 (**listenerClass** タグ名を参照)
- **listener** が処理するエンドポイント参照 (EPR)。これはトランスポート固有となります。次の例は Java Message Service のエンドポイント参照 (JMS EPR) に該当します (**connection-factory**、**destination-type**、**destination-name**、**jndi-type**、**jndi-URL**、**message-selector** のタグ名を参照)。
- **action pipeline**。1 つ以上の `<action>` 要素を必要とし、それぞれが少なくとも **class** のタグ名を含んでいる必要があり、これが **chain** 内のそのリンクに対して、インスタンス作成する **action** クラスを確定します。

```
<?xml version = "1.0" encoding = "UTF-8"?>
<jbossesb
xmlns="http://anonsvn.labs.jboss.com/labs/jbossesb/trunk/product/etc/schemas/xml/jbossesb-1.0.1.xsd" parameterReloadSecs="5">

<providers>
  <jms-provider name="JBossMQ"
    connection-factory="ConnectionFactory"
    jndi-URL="jnp://127.0.0.1:1099"
    jndi-context-factory="org.jnp.interfaces.NamingContextFactory"
    jndi-pkg-prefix="org.jboss.naming:org.jnp.interfaces">
    <jms-bus busid="quickstartGwChannel">
      <jms-message-filter dest-type="QUEUE"
        dest-name="queue/quickstart_helloworld_Request_gw"/>
    </jms-bus>
    <jms-bus busid="quickstartEsbChannel">
      <jms-message-filter dest-type="QUEUE"
        dest-name="queue/quickstart_helloworld_Request_esb"/>
    </jms-bus>
  </jms-provider>
</providers>

<services>
  <service category="FirstServiceESB"
    name="SimpleListener" description="Hello World">
    <listeners>
      <jms-listener name="JMS-Gateway"
        busidref="quickstartGwChannel" maxThreads="1"
        is-gateway="true"/>
      <jms-listener name="helloworld"
        busidref="quickstartEsbChannel" maxThreads="1"/>
    </listeners>
  </service>
</services>
</jbossesb>
```



```

<actions>
  <action name="action1" class="org.jboss.soa.esb.samples.
quickstart.helloworld.MyJMSListenerAction"
    process="displayMessage" />
  <action name="notificationAction"
    class="org.jboss.soa.esb.actions.Notifier">
    <property name="okMethod" value="notifyOK" />
    <property name="notification-details">
      <NotificationList type="ok">
        <target class="NotifyConsole"/>
      </NotificationList>
      <NotificationList type="err">
        <target class="NotifyConsole"/>
      </NotificationList>
    </property>
  </action>
</actions>
</service>
</services>
</jbossesb>

```

この設定例では **javax.jms.ObjectMessage** 内でシリアル化される受信 ESB メッセージを待機している **listener** オブジェクト (jms-listener 属性) のインスタンスを作成し、それぞれの受信メッセージを 2 つのステップから成る (<action> エレメント) **action pipeline** に配信します。

1. action1: **MyJMSListenerAction** (普通のサンプルが続く)
2. notificationAction: **org.jboss.soa.esb.actions.SystemPrintln** クラス

この **action** クラスは XML アクション設定をデバッグする際に便利であることがわかります。

```

public class MyJMSListenerAction
{
    ConfigTree _config;
    public MyJMSListenerAction(ConfigTree config) { _config = config; }

    public Message process (Message message) throws Exception
    {
        System.out.println(message.getBody().get());
        return message;
    }
}

```

action クラスを仕様して、ニーズに合わせてフレームワークをカスタマイズします。**ActionProcessingPipeline** クラスは少なくとも次を提供するために他のアクションクラスを必要とします。

- **ConfigTree** タイプの引数を 1 つとるパブリックコンストラクター
- message 引数を取り、メッセージを返す 1 つまたは複数のパブリックメソッド

オプションのコールバックメソッドを使用して **action pipeline** の各ステップの結果を通知します (以下のアイテム 5 と 6 を参照)。

org.jboss.soa.esb.listeners.message.ActionProcessingPipeline クラスは `<action>` エレメントを使って設定されるすべてのステップに対して次のステップを実行します。

1. `class` 属性で指定されるクラスのオブジェクトをインスタンス化します。これには、**ConfigTree** タイプの引数を 1 つとるコンストラクターを必要とします。
2. `process` 属性のコンテンツを分析します。

コンテンツはインスタンス作成されるクラスのパブリックメソッド名をコンマで区切った一覧になり (ステップ 1)、それぞれ **Message** タイプの単一引数をとらなければならない、結果、**pipeline** 内で次のステップに渡される `message` オブジェクトを返します。

`process` 属性がない場合、**pipeline** は **process** と呼ばれる単一の処理メソッドを使用するとみなされます。

連続的に複数の `<action>` エレメントを使用するのに比べ、単一の `<action>` エレメントで複数のメソッド名一覧を使用する方が利点があります。**action** クラスが一度インスタンス作成されると、複数のメソッドはそのクラスの同じインスタンスで呼び出されるからです。これによりオーバーヘッドを低減し、インスタンスオブジェクト内にステータス情報を保持することができます。

ユーザー提供の (新しい) **action** クラスに対して便利な方法になりますが、他の代替方法 (`<action>` エレメントの一覧) も変わらず他の既存 **action** クラスを再利用できる方法となります。

3. 前のステップで返されるメッセージを使って一覧内の各メソッドを連続して呼び出します。

いずれかのステップで返される値が **null** の場合、**pipeline** は処理を直ちに停止します。

各 `<action>` 要素で成功するコールバックメソッドは以下のとおりです。

- `process` 属性のメソッド一覧が正常に実行された場合、**pipeline** は `okMethod` 属性のコンテンツを分析します。
- メソッドが指定されていない場合、処理が次の `<action>` エレメントに継続されます。
- メソッドが `okMethod` 属性内にある場合、ステップ 3 の最後のメソッドにより返されたメソッドを使用して呼び出します。パイプラインが正常に完了すると、`okMethod` 通知が最後ものから最初のものへ全ハンドラーに対して呼び出されます。

各 `<アクション>` エレメント内の失敗に対するコールバックメソッドは以下のとおりです。

- 例外が発生すると **exceptionMethod** 通知が現在の (失敗している) ハンドラーから最初のハンドラーまですべてのハンドラー上で呼び出されます (現在のところ、**exceptionMethod** の指定がないと、得られる出力はログ記録のエラーのみになります)。
- **ActionProcessingFaultException** がプロセスメソッドから送出されると次のセクションで定義されるルールに従ってエラーメッセージが返されます。エラーメッセージの内容は例外の **getFaultMessage** から返されるもの、またはオリジナルの例外内の情報を含むデフォルトの **Fault** のいずれかになります。

固有のニーズにあわせて ESB の動作を調整するためユーザーによって供給される **action** クラスは追加でランタイムの設定が必要な場合があります (例: 上記の XML 内の **notifier** クラスは

<NotificationList> 子エレメントを必要とする)。各 <action> エレメントは上述の属性を利用してこれ以外の属性やオプションの子エレメントをすべて無視します。しかし、これらは必要となる **ConfigTree** 引数内の **action** クラスコンストラクターに渡されます。各アクションクラスはその該当する <action> エレメントでインスタンス作成されるので、「兄弟となる」 **action** エレメントを表示しません (表示してはいけません)。



注記

<action> ターゲット内の **NotificationList** エレメントを囲むために使用されるプロパティの名前は検証されません。

4.1.4. 注釈付きのアクションクラス

JBoss Enterprise Service Bus には、クリーンな **action** 実装をより簡単に作成するアクションアノテーションがあります。これにより、インターフェース、抽象クラスの実装や **ConfigTree** タイプの処理に関する複雑性が表面化しなくなります (**jboss-esb.xml** ファイル内の設定情報)。

以下がそのアノテーションです。

- @Process
- @ConfigProperty
- @Initialize
- @Destroy
- @BodyParam
- @PropertyParams
- @AttachmentParam
- @OnSuccess
- @OnException

4.1.4.1. @Process

最も簡単な実装は、@Process のアノテーション付きの単一メソッドと基本的な *plain old Java object* (POJO) でアクションを作成します。

```
public class MyLogAction {

    @Process
    public void log(Message message) {
        // log the message...
    }
}
```

@Process アノテーションは、有効な ESB **action** としてクラスを特定します。クラス内に複数のメソッドがある場合、メッセージインスタンスを処理するのに使用するメソッドを特定します (または、メッセージの一部。これについては、@BodyParam、@PropertyParams and @AttachmentParam アノテーションの説明時に詳しくみていきます)。

この **action** インスタンスを **pipeline** に設定するには、ロー／ベースレベルの **action** 実装と同じプロセスを使用します (**AbstractActionPipelineProcessor** を継承する、または **ActionLifecycle** を実装するもの、あるいはその他のサブタイプや抽象実装):

```
<service .....>
  <actions>
    <action name="logger" class="com.acme.actions.MyLogAction" />
  </actions>
</service>
```

@Process のアノテーションがついた複数のメソッドに **action** 実装が関連付けられている場合、process 属性を使用して、メッセージインスタンスの処理に使用するものはどれか指定します。

```
<service .....>
  <actions>
    <action name="logger" class="com.acme.actions.MyLogAction"
              process="log" />
  </actions>
</service>
```

4.1.4.2. @Process メソッドの戻り値

以下を返すように @Process メソッドを実装することができます。

- void: これは、上記の Logger アクション実装にあるように、戻り値がないという意味です。
- message: これは ESB メッセージインスタンスです。 **action pipeline** 上でアクティブな／現在のインスタンスになります。
- その他のタイプ。メソッドが ESB メッセージインスタンスを返さない場合、返されたオブジェクトインスタンスは、 **action pipeline** にある現在の ESB メッセージインスタンスに設定されます。メッセージがどこに設定されるかは、set-payload-location <action> 設定プロパティにより変わります。これは、通常の **MessagePayloadProxy** ルールに従いデフォルト設定します。

4.1.4.3. @Process メソッドパラメーター

@Process メソッドを使用して、様々な方法でパラメーターを指定します。

1. メソッドパラメーターとして ESB メッセージインスタンスを指定
2. 任意のパラメータータイプを 1 つ以上指定。Enterprise Service Bus フレームワークは、アクティブ／現在のパイプラインメッセージインスタンス内でそのタイプのデータを検索します。まず、メッセージボディ、次にプロパティ、最後に添付を検索し、そのパラメーターの値としてこのデータを渡します (該当するものが見つからない場合は **null**)。

最初のオプションに関する例は、Logger アクションの上記の例に記述されています。2 つ目のオプションの例を以下に示しています。

```
public class OrderPersister {

    @Process
    public OrderAck storeOrder(OrderHeader orderHeader,
                               OrderItems orderItems) {
```



```

        // process the order parameters and return an ack...
    }
}

```

この例では、`@Process` メソッドは、**OrderHeader** や **OrderItem** オブジェクトインスタンスの作成や、現在のメッセージへの添付について **pipeline** の 1 つ前のアクションに左右されます (より現実的な実装では、XML または EDI ペイロードを注文インスタンスにデコーディングする一般的なアクション実装を持ちます。**OrderPersister** は、単独のパラメーターとして注文インスタンスを取ります)。以下に例を示します。

```

public class OrderDecoder {

    @Process
    public Order decodeOrder(String orderXML) {
        // decode the order XML to an Order instance...
    }

}

public class OrderPersister {

    @Process
    public OrderAck storeOrder(Order order) {
        // persist the order and return an ack...
    }

}

```

サービス設定で 2 つのアクションをつなぎます。

```

<actions>
    <action name="decode" class="com.acme.orders.OrderDecoder" />
    <action name="persist" class="com.acme.orders.OrderPersister" />
</actions>

```

アノテーションが少ないため、オプション 2 のほうが読み込みが簡単ですが、実行時に適切なパラメーターをメッセージ内から検索する処理が**決定的でない**ため、リスクがあります。このため、Red Hat は `@BodyParam`、`@PropertyParam`、`@AttachmentParam` アノテーションをサポートします。

これらの `@Process` メソッドパラメーターのアノテーションを使用して、明示的にメッセージ内のどこから `@Process` メソッドの個別のパラメーター値がリトリブされているか定義します。名前の通り、これらのアノテーションはそれぞれ、パラメーターごとに名前付きのロケーションを指定することができます。

```

public class OrderPersister {

    @Process
    public OrderAck storeOrder(
        @BodyParam("order-header") OrderHeader orderHeader,
        @BodyParam("order-items") OrderItems orderItems) {

        // process the order parameters and return an ack...

    }

}

```


指定のメッセージロケーションに値が含まれていない場合、このパラメーターについて null が渡されます (@Process メソッドインスタンスはこの処理方法を決定)。反対に指定のロケーションに誤ったタイプの値が含まれている場合、**MessageDeliverException** がスローされます。

4.1.4.4. @ConfigProperty

アクションは、カスタム設定がある程度必要になるものがほとんどです。ESB アクション設定では、プロパティは <action> 要素の <property> サブ要素として提供されます。

```
<action name="logger" class="com.acme.actions.MyLogAction">
  <property name="logFile" value="logs/my-log.log" />
  <property name="logLevel" value="DEBUG" />
</action>
```

これらのプロパティを活用するため、低／基本レベルのアクション実装を使用する必要があります (**AbstractActionPipelineProcessor** の継承または、**ActionLifecycle** の実装)。これは、**ConfigTree** クラスと連携します (コンストラクター経由でアクションに提供します)。アクションを実装するには、以下の手順に従います。

1. **ConfigTree** インスタンスを提供するアクションクラスへコンストラクターを定義します。
2. **ConfigTree** インスタンスから該当のアクション設定プロパティをすべて取得します。
3. 必須のアクションプロパティを確認して、<action> 設定に指定されていない場所で例外をあげます。
4. プロパティの全値を文字列 (**ConfigTree** で提供) から、アクション実装で使用される適切なタイプにデコーディングします。例えば、**java.lang.String** から **java.io.File** に、**java.lang.String** から **Boolean** に、**java.lang.String** から **long** などにデコーディングします。
5. 設定値が対象のプロパティタイプにデコーディングできない場所では、例外が出されます。
6. 考えられる各種設定すべてに単体テストを実装して、さきほど表示したタスクが正しく完了するようにします。

上記のタスクは一般的に実行するのは難しくありませんが、作業が増え、エラーが発生しやすく、アクションごとに設定の間違いを処理する方法も異なってきます。結果として、比較すると全体的に明確さが欠けますが、多くのコードを追加する必要がある場合もあります。

アノテーション付きのアクションは、**@ConfigProperty** を使用してこのような問題に対処します。2 種の必須設定プロパティ (logFile and logLevel) を持つ、MyLogActions 実装を拡張します。

```
public class MyLogAction {

    @ConfigProperty
    private File logFile;

    @ConfigProperty
    private LogLevel logLevel;

    public static enum LogLevel {
        DEBUG,
        INFO,
        WARN
    }
}
```

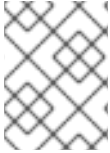


```

    }

    @Process
    public void log(Message message) {
        // log the message at the configured log level...
    }
}

```



注記

(フィールドではなく) "setter" メソッド上で `@ConfigProperty` アノテーションを定義することも可能です。

必要事項は以上です。Enterprise Service Bus がアクションをデプロイすると、デコーディングされた値にある実装とマップの両方を検証します (enum のサポートと上記の `LogLevel` enum)。`@ConfigProperty` アノテーションを処理するアクションフィールドを検索します。開発者は、**ConfigTree** クラスを処理したり、余分のコードを開発したりする必要がありません。

デフォルトでは、`@ConfigProperty` アノテーションを処理するクラスフィールドはすべて必須です。必須ではないフィールドは、以下の 2 種のいずれかの方法で処理されます。

1. フィールドの `@ConfigProperty` アノテーションで `use = Use.OPTIONAL` を指定
2. フィールドの `@ConfigProperty` アノテーションで `defaultVal` を指定 (任意)

ログアクションのプロパティをオプションのみにするには、以下のようなアクションを実装します。

```

public class MyLogAction {

    @ConfigProperty(defaultVal = "logs/my-log.log")
    private File logFile;

    @ConfigProperty(use = Use.OPTIONAL)
    private LogLevel logLevel;

    public static enum LogLevel {
        DEBUG,
        INFO,
        WARN
    }

    @Process
    public void log(Message message) {
        // log the message...
    }
}

```

`@ConfigProperty` アノテーションは、2 つの追加フィールドをサポートします。

1. `name`: これを使用して、アクションインスタンスで指定の名前のフィールドを生成する際に使用するアクション設定プロパティ名を明示的に指定します。
2. `choice`: このフィールドを使用して、許容の設定値を制限します。これは、列挙型を使用することで可能です (`LogLevel`)

この名前フィールドは、古いアクション (低／基本レベルの実装タイプを使用) を新しいアノテーションベースの実装に以降する際などに使用できます。プロパティの古い設定名 (後方互換との関係で変更不可) は有効な Java フィールド名へのマッピングは行いません。ログアクションを例にとります。以下がログアクションの古いアクションだと仮定します。

```
<action ...>
  <property name="log-file" value="logs/my-log.log" />
  <property name="log-level" value="DEBUG" />
</action>
```

このプロパティ名は、有効な Java フィールド名にマッピングを行わないため、`@ConfigProperty` アノテーションで名前を指定します。

```
public class MyLogAction {

    @ConfigProperty(name = "log-file")
    private File logFile;

    @ConfigProperty(name = "log-level")
    private LogLevel logLevel;

    public static enum LogLevel {
        DEBUG,
        INFO,
        WARN
    }

    @Process
    public void log(Message message) {
        // log the message...
    }
}
```

4.1.4.5. プロパティの値のデコード

Bean 設定のプロパティ値は、文字列の値からデコーディングされます。適切な POJO bean プロパティタイプと一致させるには、以下のシンプルなルールを使用します。

1. プロパティタイプに単一引数の文字列コンストラクターがある場合、これを使用します。
2. Primitive な場合、オブジェクトタイプの単一引数文字列コンストラクターを使用します。例えば、int の場合、整数のオブジェクトを使用します。
3. 列挙型の場合 `Enum.valueOf` を使用して設定の文字列を列挙型の値に変換します。

4.1.4.6. @Initialize と @Destroy

ときに、アクション実装はデプロイメント時に初期化タスクを実行する必要があります。また、アンデプロイ時にはクリーンアップも行う必要があります。このような理由から、`@Initialize` および `@Destroy` メソッドアノテーションがあります。

これについて例を挙げます。デプロイメント時に、ロギングアクションはチェックをいくつか行います (例えばファイルやディレクトリが存在するかなど)。アンデプロイされると、アクションはクリーンアップタスクを実行する必要があります (ファイルを閉じるなど)。以下がこれらのタスクを実行するためのコードです。


```

public class MyLogAction {

    @ConfigProperty
    private File logFile;

    @ConfigProperty
    private LogLevel logLevel;

    public static enum LogLevel {
        DEBUG,
        INFO,
        WARN
    }

    @Initialize
    public void initializeLogger() {
        // Check if file already exists... check if parent folder
        // exists etc...
        // Open the file for writing...
    }

    @Destroy
    public void cleanupLogger() {
        // Close the file...
    }

    @Process
    public void log(Message message) {
        // log the message...
    }
}

```

注記

@ConfigProperty アノテーションはすべて、ESB デプロイヤーが @Initialize メソッドを呼び出す前に処理されます。そのため、@Initialize メソッドは、カスタマイズの初期化を行う前に準備が整ったこれらのフィールドに依存します。

注記

これらのアノテーションを両方使用して、メソッドを指定する必要はありません。必要がある場合のみ指定します。つまり、メソッドが初期化のみ必要な場合、@Initialize アノテーションのみを使用します (@Destroy アノテーションで注釈をつけた matching メソッドを提供する必要はありません)。

注記

単一のメソッドを指定して、@Initialize と @Destroy 両方でアノテーションをつけることができます。

**注記**

@Initialize メソッドに **ConfigTree** パラメーターをオプションで指定可能です。これを行うと **ConfigTree** インスタンスを基本とするアクションにアクセスできるようになります。

4.1.4.7. @OnSuccess と @OnException

アクションが設定されている **pipeline** の中で、実行が成功した際、または失敗した際にこれらのメソッドを実行すべきか指定するため、これらのアノテーションを使用します。

```
public class OrderPersister {

    @Process
    public OrderAck storeOrder(Order order) {
        // persist the order and return an ack...
    }

    @OnSuccess
    public void logOrderPersisted(Message message) {
        // log it...
    }

    @OnException
    public void manualRollback(Message message,
                               Throwable theError) {
        // manually rollback...
    }
}
```

これらのアノテーションの場合、パラメーターはパラメーターに渡すかはオプションです。上記のパラメーターの一部、すべてを提供、何も提供しないといった選択ができます。Enterprise Service Bus フレームワークは、それぞれの場合で該当のパラメーターを解決します。

4.1.5. アクションとメッセージ

Action は Message が到着することで起動されます。特定の Action 実装は Message 内のどこにデータが存在しているか認識している必要があります。サービスは Action の任意の番号を使って実装される場合があるため、単一入力の Message が複数の Action の代表として情報を含んでいる可能性があります。このような場合、Message Body 内にそのデータ用の固有の場所を 1 つまたは複数選択してサービスを使用する側に対してこれを伝えるのは Action 開発者の義務となります。

**注記**

さらに、Action 同士はチェーン化される場合があるため、チェーンの始めの方の Action にオリジナル入力の Message を修正させるまたは完全に置き換えさせることが可能です。

**重要**

セキュリティ上、サービスチェーン内で未知の Action を使用する場合は注意が必要です。情報の暗号化を推奨します。

重要

複数のアクションが入力メッセージのデータを共有、変更する場合、チェーンを介して情報が流れるため、オリジナルの情報を保持するよう Red Hat は推奨しています。チェーンの先のほうに進んでも最初の情報にアクセスできるようにするためです (これができない状況もあります)。

入力されたデータを変更するアクションは、**org.jboss.soa.esb.actions.post** のボディの場所に追加することもできます。つまり、チェーン内に N 個のアクションがある場合、N 番目のアクションは、通常検索する箇所でオリジナルデータを検索します。また、N-1 番目のアクションがデータを変更した場合、N は他の指定の場所で検索を行います。アクションチェーンをスムーズに実行するには、N 番目のアクションが **org.jboss.soa.esb.actions.pre** をチェックして、N-2 番目のアクションがデータを変更していないか確認します。



警告

デフォルトのボディロケーションを使用するアクションをチェーン化する場合はコンフリクトが起こらないように注意が必要です。

4.1.6. レスポンスの処理

JBoss Enterprise Service Bus は、**action pipeline** レスポンスの処理を目的とする 2 種類の処理メカニズムに対応しています。このメカニズムは、*明示的*および*暗黙的*処理メカニズムとよばれ、後者はアクションのレスポンスをベースとしています。

暗黙的メカニズムを使用した場合、レスポンスは次のように処理されます。

- **pipeline** 内のアクションが null メッセージを返す場合、レスポンスは送信されません。
- **pipeline** 内の最後のアクションがエラー以外の応答を返した場合、要求メッセージの ReplyTo EPR に返信が送信されます。これが設定されていない場合は、要求メッセージの From EPR に送信されます。応答をルーティングできない事態が発生すると、エラーメッセージがシステムによりログ記録されます。

明示的なメカニズムを使用した場合、レスポンスは次のように処理されます。

- **pipeline** が **OneWay** で指定されている場合、レスポンスは送信されません。
- **pipeline** が **RequestResponse** で指定されている場合、要求メッセージの ReplyTo EPR に返信が送信されます。これが設定されていない場合は、要求メッセージの From EPR に送信されます。EPR に指定がない場合、システムによるエラーメッセージはログ記録されません。

重要

すべてのアクションパイプラインに明示的な処理のメカニズムを使用させることを推奨します。mep 属性を **jboss-esb.xml** ファイル内の actions エレメントに追加するだけでこれを行えます。この属性の値は **OneWay** または **RequestResponse** のいずれかにします。

4.1.7. アクション処理中のエラーの扱い方

アクションチェーンを処理しているあいだにエラーが発生する可能性があります。このようなエラーは **pipeline** から例外として送出され、これによりパイプラインの処理を終了させるはずですが、前述のように、失敗メッセージが **ActionProcessingFaultException** として返されます。送信者 (または仲介者) に返されるエラー情報が重要である場合、FaultTo EPR を設定してください。これを設定しないと **JBoss Enterprise Service Bus** は ReplyTo EPR に基づいてエラーメッセージの配信を試行し、ReplyTo EPR も設定されていない場合は From EPR に配信を試行します。いずれの EPR も設定されていない場合、エラー情報はローカルにロギングされます。

アクション実装は、様々な種類のエラーメッセージを返すことができます。しかし、**JBoss Enterprise Service Bus** は、以下の "system" エラーメッセージに対応しています。例外が送出され、アプリケーション固有の失敗メッセージがある場合、統一のリソース識別子により特定される場合があります。

urn:action/error/actionprocessingerror

これは、**chain** 内のアクションが **ActionProcessingFaultException** を送出したが返す失敗メッセージを含んでいなかったという意味になります。例外の詳細は **Fault** の **reason** String 内に含まれます。

urn:action/error/unexpectederror

つまり、処理中に予期しない例外が出現していることになります。例外に関する詳細は **Fault** の **reason** String 内にあります。

urn:action/error/disabled

アクション処理が無効になります。

Action チェーン内で例外が送出される場合、**FaultMessageException** 内のクライアントに伝播して戻されます。これは **Courier** か **ServiceInvoker** のクラスから再度送出されます。**Fault** メッセージが受信されるときにも必ず送出されるこの例外は、**Fault** のコードおよび理由の他に伝播された例外を含みます。

4.2. メタデータとフィルター

Enterprise Service Bus を通ってメッセージが循環されるため、メタデータを添付したい場合があります。また、トランザクションやセキュリティ情報を追加するなど、動的にメッセージを補強する必要があるかもしれません。これらの機能はいずれもゲートウェイと ESB ノード群両方に対して **filter mechanism** を使って Enterprise Service Bus でサポートされます。

org.jboss.soa.esb.filter.InputOutputFilter クラスは 2 種類のメソッドを持ちます。

- **public Message onOutput (Message msg, Map<String, Object> params)** は **CourierException** をスローします。これははトランスポートへのメッセージの流れとして呼び出されます。実装はメッセージを修正して新しいバージョンを返すことができます。追加情報は追加のパラメーターの形式で呼び出し側により提供することが可能です。
- **public Message onInput (Message msg, Map<String, Object> params)** は、**CourierException** をスローします。これは、トランスポートからのメッセージの流れとして呼び出されます。実装はメッセージを修正して新しいバージョンを返すことができます。追加情報は追加のパラメーターの形式で呼び出し側により提供することが可能です。

フィルターは **jbossesb-properties.xml** ファイルの **filters** セクションで定義します (**jbossesb.sar** アーカイブにあり)。これには、**org.jboss.soa.esb.filter.<number>** プロパティを使いま

す。<number> の部分は何の値でも構いません。複数のフィルターを呼び出す順序を示すのに使用します (低い値から高い値)。



注記

この環境にデプロイする各 ESB インスタンスに、**jbossesb-properties.xml** への変更を置く必要があります。こうすることで、すべてのインスタンスが同じメタデータを処理できるようにします。

JBoss Enterprise Service Bus には以下が含まれています。

- **org.jboss.internal.soa.esb.message.filter.MetadataFilter**
- **org.jboss.internal.soa.message.filter.GatewayFilter**
- **org.jboss.internal.soa.esb.message.filter.EntryExitTimeFilter**

これらは、指定のプロパティ名や返された文字列の値を持つプロパティとして、以下のメタデータをメッセージに追加します。

ゲートウェイ関連のメッセージプロパティ

org.jboss.soa.esb.message.transport.type

ファイル、FTP、JMS、SQL、Hibernate

org.jboss.soa.esb.message.source

これは、メッセージが読み込まれたファイルの名前

org.jboss.soa.esb.message.time.dob

メッセージが ESB に入った時間 (送信された時間またはゲートウェイに到着した時間)

org.jboss.soa.esb.message.time.dod

メッセージが ESB から出た時間 (受信した時間)

org.jboss.soa.esb.gateway.original.queue.name

メッセージが JMS ゲートウェイノード経由で受信された場合、このエレメントは受信された元のキューの名前を含む

org.jboss.soa.esb.gateway.original.url

メッセージが SQL ゲートウェイノード経由で受信された場合、このエレメントはオリジナルのデータベース URL を含む



注記

すべての Enterprise Service Bus ノードで **GatewayFilter** のデプロイは安全ですが、**gateway node** にデプロイされる場合は、メッセージへの情報追加しか行いません。

適切なフィルターを作成して登録することで、メッセージにより多くのメタデータを追加することができます。フィルターは追加パラメーター内の次のような名前が付くエントリの存在があるかないかでゲートウェイノード内で稼働しているかどうかを判断することができます。

ゲートウェイ生成のメッセージパラメーター

`org.jboss.soa.esb.gateway.file`

メッセージ提供元のファイル。このゲートウェイがファイルベースである場合にのみ表される。

`org.jboss.soa.esb.gateway.config`

ゲートウェイインスタンスの初期化に使用された **ConfigTree**。



注記

Red Hat は、ファイルベースの Java Message Service および SQL ゲートウェイを使用する場合のみ、**GatewayFilter** をサポートします。

4.3. サービスとは

JBoss Enterprise Service Bus では、サービスは連続する形でメッセージを処理するアクションクラスの一覧であると定義されています。この定義内では、Enterprise Service Bus は、サービスの構成要素には制限を貸さないとされています。本ガイドの前半で説明したように、理想的な SOA インフラストラクチャーとは、メッセージが非常に重要となり実装固有の詳細が抽象インターフェースの背後に隠れるようなクライアントとサービス間に積極的な疎結合の対話式を採用しているインフラストラクチャーです。これにより、実装はクライアントやユーザー側での変更を必要としない変更が可能になります。メッセージの定義に対する変更で必然的に伴うのはクライアントへの更新のみになります。

こうした理由から、Enterprise Service Bus はサービスの定義と構成にメッセージ駆動型パターンを使用します。クライアントは Message をサービスに送り、基本的なサービスインターフェースは本質的に受け取った Message で動作する単一プロセスのメソッドになります。内部的にサービスは 1 つまたは複数の Action から構成され、着信 Message を処理するためチェーン化することができます。Action が行うことは実装依存になります。つまり、データベーステーブルのエントリを更新したり EJB を呼び出すなどです。

1. カスタマイズのサービスを開発する場合、最初にコンシューマー側に公開する概念的なインターフェース (や規定) を決定する必要があります。この規定は Message の観点から定義されなければなりません (つまり、ペイロードをどのようにするか、どのタイプの応答 Message が生成されるかなどです)。
2. 定義が完了したら、規定情報を **registry** 内で公開しなければなりません (現在、JBossESB には自動的にこれを行う手段はありません)。
3. これでクライアントは公開された規定に従ってサービスを使用することができるようになります。サービスが Message をどのように処理し必要な作業を行うかについては実装での選択になります。単一の Action 内でも複数の Action 内でも可能です。管理性と再利用性のどちらをとるかなど、多少の代償はつきものです。



注記

今後のリリースでは、サービスの開発を容易にするツールサポートを向上させる予定です。

4.3.1. ServiceInvoker

JBoss Enterprise Service Bus 4.2 では、開発にかかる作業の簡略化を支援するため **ServiceInvoker** が採用されました。**ServiceInvoker** は多くの低レベル詳細を表示させずステートレスサービスのフェールオーバーメカニズムと不透明に動作します。このように、Red Hat ではは ESB サービスのクライアント側インターフェースとして、**ServiceInvoker** の使用を推奨します。

```
public class ServiceInvoker
{
    public ServiceInvoker(Service service) throws
MessageDeliverException;
    public ServiceInvoker(String serviceCategory, String serviceName)
throws MessageDeliverException;
    public ServiceInvoker(Service service, List<PortReference.Extension>
extensions);
    public Message deliverSync(Message message, long timeoutMillis)
throws MessageDeliverException, RegistryException, FaultMessageException;
    public void deliverAsync(Message message) throws
MessageDeliverException;
    public Service getService();
    public String getServiceCategory();
}
```

ServiceInvoker のインスタンスはクライアントが対話を必要とするそれぞれのサービスに対して作成することができます。作成すると、そのインスタンスはプライマリの EPR またフェールオーバーの際には代替となる EPR を確定するため必要に応じて **registry** にコンタクトします。

作成すると、クライアントはサービスに対して同期的 (**deliverSync**) にまたは非同期的 (**deliverAsync**) にメッセージを送信する方法を確定できるようになります。同期的な方法の場合、タイムアウトを指定する必要があり、これはクライアントが応答を待機する時間になります。この期間内に応答が受信されないと **ResponseTimeoutException** が送出されます。

registry へコンタクトがとれない、またはサービスのルックアップに成功した場合、**RegistryException** が、**deliverSync** から送出されます。タイムアウトの値は以下の 3 つのいずれかをさします。

1. サービスの問題
2. 単にオーバーロードして時間内に応答できない
3. 許容のタイムアウト時間よりもリクエストした作業に時間がかかっている



注記

問題が一時的な場合、次の試行で消える可能性があります。



注記

サービスと通信中の問題については、**FaultMessageException** が送出されます。

本ガイドの前半で説明したように、Message を送信する際に **To**、**ReplyTo**、**FaultTo**などの値を **Message** ヘッダー内に指定することが可能です。**ServiceInvoker** を使用する場合、構成される際に **registry** に既にコンタクトを行っているため **To** フィールドは不要になります。実

際、**ServiceInvoker** を通じて Message を送信する場合、To フィールドは同期的配信モードでも非同期的配信モードでも無視されます。



注記

JBoss Enterprise Service Bus の今後のリリースでは、別の配信先として提供された To フィールドを使用することができ、**registry** により返されたエンドポイント参照により、アクティブなサービスへの解決が失敗します。



注記

JBoss Enterprise Service Bus のフェールオーバーのプロパティについては、本書の「高度なトピック」の章で説明しています。複数のコピーが **registry** にある場合 **ServiceInvoker** が個別サービスインスタンスの問題を不透明にするのですが、高度なトピックでは、その方法についても説明しています。しかし、問題発生直後にアプリケーションに通知するための自動フェールオーバーを避けたい場合もあります。この方法は、**property** にて **org.jboss.soa.esb.exceptionOnDeliverFailure** プロパティを **true** に設定することでグローバルレベルの設定を行います。あるいは、メッセージ別にこれを設定します。固有のメッセージで同プロパティを **true** に設定することも可能です (いずれの場合もデフォルト値は **false** です)。

4.3.2. トランザクション

InVM や Java Message Service など、一部の媒体で、**トランザクション配信セマンティクス**に対応しています。このような媒体に対する配信セマンティクスは **JMS トランザクションセッション**をベースにしています。つまり、トランザクションの範囲内にあるキューにメッセージが置かれるという意味です。しかし、内包されるトランザクションがコミットされるまで実際は配信されません。その後、別のトランザクションの範囲内で受け取り側がプルします。

JBossESB 4.5 以降はこのようなブロックされる状況を検出するよう試み、**IncompatibleTransactionScopeException** をスローします。JBoss ESB は、随時これをキャッチして状況に応じて機能します。

4.3.3. サービスと **ServiceInvoker**

クライアントとサービスの環境では、クライアントとサービスという用語を使って役割を表し、単一のエンティティが同時にクライアントでありサービスであり得ます。このように、**ServiceInvoker** とは純粋にクライアントの領域になりませんのでご注意ください。サービス内、特に Action 内での使用が可能です。たとえば、ビルトインの **content-based router** を使用せずに、特定のビジネスロジックの評価に基づき異なるサービスに着信メッセージを再ルーティングさせるため Action を使用したい場合、あるいは Action によって後日の管理目的で **Dead Letter Queue** に障害メッセージの特定タイプをルーティングするよう決定させることもできます。

このようにして **ServiceInvoker** を使用すると、(「高度なトピック」の章で説明されているように) サービスにとって不透明なフェールオーバーのメカニズムという利点が得られます。つまり、複雑な開発作業をすることなく、より堅固な方法で他のサービスや障害などへの一方向要求のルーティングが可能になるということになります。

4.3.4. InVM トランスポート

InVM トランスポートは同じ JVM で実行されているサービス間の通信を提供します (InVM とは *In Virtual Machine* のことです)。つまり、**ServiceInvoker** のインスタンスは、ネットワークやメッセージシリアル化のオーバーヘッドなしに同じ JVM 内からサービスを起動できます。



注記

サービス間通信を容易化するために使用する内部のデータ構造を最適化することで、InVM 機能がパフォーマンスの利点を受け取ることができます。メッセージの保存にしようとするキューは永続的ではありません。つまり、メッセージは問題があれば失われる可能性があります。さらに、キューが空になる前にサービスが停止した場合、これらのメッセージは配信されません。他の制限は本章全体で記載しています。

JBoss Enterprise Service Bus では、複数のトランスポートにて同時にサービスが呼び出されるため、高性能および信頼性を実現できる方法でサービスを設計します。これには、各メッセージタイプに大して適切なトランスポートを選択してください。

```
<service category="ServiceCat" name="ServiceName" description="Test
Service">
  <actions mep="RequestResponse">
    <action name="action"
class="org.jboss.soa.esb.listeners.SetPayloadAction">
      <property name="payload" value="Tom Fennelly" />
    </action>
  </actions>
</service>
```

初期の Enterprise Service Bus は、このトランスポートをサポートしておらず、各サービスにメッセージ対応リスナー 1 つ以上持たせるように設定する必要がありました。これは必要なくなり、サービスは `<listeners>` 設定なしに設定でき、仮想マシン内から呼び出しも可能です。

```
<service category="ServiceCat" name="ServiceName" description="Test
Service">
  <actions mep="RequestResponse">
    <action name="action"
class="org.jboss.soa.esb.listeners.SetPayloadAction">
      <property name="payload" value="Tom Fennelly" />
    </action>
  </actions>
</service>
```

こうすることで、サービスの設定も比較的簡単になります。

4.3.5. InVM のスコープ

InVM サービス呼び出しのスコープは `<service>` 要素の `invmScope` 属性を使用して制御します。Red Hat では現在、以下の 2 つのスコープに対応しています。

1. **NONE**: Service は InVM トランスポートで起動できません。デフォルト値。
2. **GLOBAL**: Service は同じ Classloader スコープ内から InVM トランスポートで起動可能です。



注記

今後のリリースには **LOCAL** スコープが追加され、デプロイされた同じ **.esb** アーカイブ内での起動が制限されます。

適切な名前のついた **invmScope** 属性を使用することで、各サービスの個別 InVM スコープを指定します (サービス設定の <service> 要素)。

```
<service category="ServiceCat" name="ServiceName" invmScope="GLOBAL"
  description="Test Service">
  <actions mep="RequestResponse">
    <action name="action"
      class="org.jboss.soa.esb.listeners.SetPayloadAction">
      <property name="payload" value="Tom Fennelly" />
    </action>
  </actions>
</service>
```

Enterprise Service Bus デプロイメント用のデフォルトの InVM Scope は **jbossesb-properties.xml** ファイルで `core:jbos.esb.invm.scope.default` プロパティを使用して指定されます。指定されている設定値は **NONE** ですが、このプロパティが未定義の場合、デフォルトのスコープは実際には **GLOBAL** になります。

4.3.6. 処理された InVM

InVM listener は、トランザクションスコープまたは非トランザクションスコープの中で実行可能です。トランザクション対応の他のトランスポートとして同じように実行します。明示的または暗黙的に設定を変更することでこの動作を制御します。

サービス要素の `invmTransacted` 属性を使い、トランザクションスコープの明示的設定を操作します。この設定は常に、暗黙的設定よりも優先されます。



注記

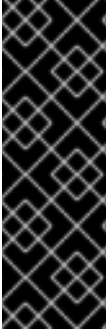
同じサービス上に設定されたトランザクショントランスポートが他にもある場合、**InVM listener** は暗黙的にトランザクションされます。現在、これらの追加トランスポートは JMS、Scheduled または SQL タイプとなります。

4.3.7. トランザクションセマンティック



警告

Enterprise Service Bus の *InVM* トランスポートはトランザクション形式であり、**message queue** は揮発性メモリにのみ保持されます。つまり、システム障害時またはシャットダウン時にこのトランスポートのメッセージキューは失われます。



重要

InVM トランスポートでは、ACID (Atomicity, Consistency, Isolation, Durability) セマンティクスすべてを実現できない場合があります。特に、データベースなど他のトランザクションリソースとあわせて使用している場合などです。これは、InVM キューの変動性が原因となっています。しかし、多くの場合、この欠点よりも、InVM のパフォーマンスの利点のほうがかなり大きくなります。全 ACID セマンティクスが必要な場合、Red Hat は、Java Message Service などの別のトランザクショントランスポートを使用するよう推奨しています。

InVM がトランザクション内で使用される場合、トランザクションが実際にコミットされるまで、メッセージは受取側のキューにはでてきません。しかし、送信側が後にキューに入れるようにメッセージが受信された旨の確認を直後に受け取ります。受取側は、キューからトランザクションのスコープ内にあるメッセージをプルしようとする、トランザクションが結果的にロールバックされる場合メッセージは自動的にキューに置かれます。メッセージの送信側や受信側がトランザクションの結果を知りたい場合、直接モニタリングするか、トランザクションの同期登録をする必要があります。

トランザクションマネージャーがメッセージをキューに戻す場合、同じ場所に戻らない場合があります。これは、パフォーマンスを最適化する目的でこのように設計されています。アプリケーションで特定の順番にメッセージを並べる必要がある場合、別のトランスポートか、単一の wrapper メッセージ内にグループ関連のメッセージを使用するようにしてください。

4.3.8. スレッド化

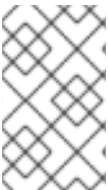
InVM トランスポートに関連するリスナースレッドの数を変更するには、以下のコードを使用します。

```
<service category="HelloWorld" name="Service2"
  description="Service 2" invmScope="GLOBAL">
  <property name="maxThreads" value="100" />
  <listeners>...
  <actions>...
```

4.3.9. ロックステップ配信

InVM transport は低いオーバーヘッドでメモリ内メッセージキューにメッセージを配信します。これは非常に高速であり、メッセージを消費するサービスに対して配信が行われるのが速すぎるとメッセージキューが溢れてしまうことがあります。これらの状況を緩和するために InVM トランスポートはロックステップ配信メカニズムを提供します。

子のメカニズムでは、サービスがメッセージを取得する前にメッセージがサービスに配信されないようにします。これは、受信側サービスがメッセージを取得するか、タイムアウト時間に到達するまでメッセージ配信をブロックすることによって行われます。



注記

これは同期配信方法ではありません。応答を待ったり、サービスがメッセージを処理するのを待ったりせず、メッセージがサービスによってキューから削除されるまでブロックします。

ロックステップ配信はデフォルトで無効になっていますが、<service> の <property> 設定を使用してサービスに対して設定できます。

- inVMLockStep: LockStep 配信を有効にするかどうかを制御するブール値

- `inVMLockStepTimeout`: メッセージ取得の待機時にメッセージ配信をブロックする最大時間 (ミリ秒単位)

```
<service category="ServiceCat" name="Service2"
  description="Test Service">
  <property name="inVMLockStep" value="true" />
  <property name="inVMLockStepTimeout" value="4000" />

  <actions mep="RequestResponse">
    <action name="action" class="org.jboss.soa.esb.mock.MockAction" />
  </actions>
</service>
```



注記

トランザクションのスコープ内で **InVM transport** を使用する場合、ロックステップ配信は無効になります。内包のトランザクションをコミットする際に、メッセージが偶発的にキューに挿入されるためです。このコミットメントは、ロックステップの待機時間の前後であれば発生する可能性があります。

4.3.10. 負荷分散

ServiceInvoker を使用する場合は、InVM トランスポート (利用可能な場合) よりもサービスを起動することが常に優先されます。他の負荷分散のストラテジは、ターゲットサービスに対して InVM エンドポイントが存在しない場合のみ適用されます。

ServiceInvoker を使用する場合は、InVM トランスポート (利用可能な場合) よりもサービスを起動することが常に優先されます。他の負荷分散ストラテジは、ターゲットサービスに対して InVM エンドポイントが存在しない場合のみ適用されます。

4.3.11. 「値で渡す」と「参照で渡す」

デフォルトでは、**InVM transport** は参照としてメッセージを渡します。場合によっては、データの整合性に問題が出てくる場合があります。また、クラスキャストの問題、つまり、**ClassLoader** 境界を越えてメッセージが交換されてしまう場合もあります。

これらの問題を回避するには、*値別*にメッセージが渡されるように設定します。このサービスの `inVMPassByValue` プロパティを **true** に設定することで、メッセージが値別に渡されるように設定します。

```
<service category="ServiceCat" name="Service2" description="Test Service">
  <property name="inVMPassByValue" value="true" />

  <actions mep="RequestResponse">
    <action name="action" class="org.jboss.soa.esb.mock.MockAction" />
  </actions>
</service>
```

4.4. サービス規定定義

契約定義は、受信要求、送信応答、および対応するサービスによってサポートされる障害詳細メッセージを表す XML スキーマ定義を含めることによってサービスに対して指定できます。要求および応答

メッセージを表すスキーマはメッセージのメインボディセクションのコンテンツの形式を定義するために使用され、そのコンテンツの検証を強制的に実行できます。

スキーマはサービスの <actions> エレメントに以下の属性を指定することによって宣言されます。

表4.1 サービス規定属性

名前	説明	タイプ
inXsd	要求メッセージのスキーマを含むリソース (単一エレメントを表す)	xsd:string
outXsd	応答メッセージのスキーマを含むリソース (単一エレメントを表す)	xsd:string
faultXsd	スキーマのコンマで区切られた一覧 (それぞれが 1 つまたは複数のエラーエレメントを表す)	xsd:string
requestLocation	ボディ内の要求コンテンツの場所 (デフォルトの場所でない場合)	xsd:string

4.5. メッセージ検証

要求および応答メッセージの内容は自動的に検証され、関連づけられたスキーマが <actions> エレメントに対して宣言されていることを確認します。この検証は、<actions> エレメントの validate 属性を **true** の値で指定することによって有効にできます。



注記

検証はデフォルトで無効にされます。

4.6. ESB サービスを WEB サービスエンドポイント経由で公開

コントラクトスキーマを宣言すると、Web サービスエンドポイントを介して ESB サービスが自動的に公開されます (**Contract Web** アプリケーションを介して探すことができます)。この機能は `webService` 属性を指定することによって変更できます。この値は以下のとおりです。

表4.2 Web サービス属性

名前	説明
false	Web サービスエンドポイントは公開されない
true	Web サービスエンドポイントは公開される (デフォルト)

デフォルトでは、Web サービスのエンドポイントは **WS-Addressing** をサポートしません。この機能を有効にするには、addressing 属性を設定します。

表4.3 WS-Addressing の値

値	説明
false	WS-Addressing はサポートされていません (デフォルト)
true	WS-Addressing サポートが必要です。

アドレッシングのサポートを有効にすると、WS-Addressing Message Id、Relates To URIs、Relationship タイプがプロパティとして受信メッセージに追加されます。

表4.4 WS-Addressing プロパティ

プロパティ	説明
org.jboss.soa.esb.gateway.ebws.messageID	WS-Addressing のメッセージ識別子
org.jboss.soa.esb.gateway.ebws.relatesTo	WS-Addressing Relates To URIs を含む文字列配列
org.jboss.soa.esb.gateway.ebws.relationshipType	Relates To URIs に該当する WS-Addressing Relationship タイプを含む文字列配列

次の例では、(Web サービスのエンドポイントを使って公開することなしに) メッセージのリクエスト/レスポンスを検証したいサービスの宣言方法について例示しています。

```
<service category="ServiceCat" name="ServiceName" description="Test
Service">
  <actions mep="RequestResponse" inXsd="/request.xsd"
outXsd="/response.xsd"
    webservice="false" validate="true">
    <!-- .... >
  </actions>
</service>
```

以下の例は要求および応答メッセージを検証し、Web サービスエンドポイントを介してサービスを公開するサービスの宣言を示しています。またサービスは要求が指定されたボディの場所の REQUEST に提供され、応答が指定されたボディの場所の RESPONSE に返されます。

```
<service category="ServiceCat" name="ServiceName" description="Test
Service">
  <actions mep="RequestResponse" inXsd="/request.xsd"
outXsd="/response.xsd"
    validate="true" requestLocation="REQUEST"
responseLocation="RESPONSE">
    <!-- .... -->
  </actions>
</service>
```


第5章 その他のコンポーネント

本章では、**JBoss Enterprise Service Bus** 内の他のインフラストラクチャーコンポーネントやサービスについて説明します。これらのサービスのいくつかはそのサービス独自のドキュメントがありますので、この場合はそちらも参照してください。本章の目的は開発者が利用できるその他のコンポーネントに関する概要を示すことです。

5.1. メッセージストア

JBoss Enterprise Service Bus のメッセージストアのメカニズムは監査追跡の目的を念頭に置いて設計されています。他の ESB サービスと同様にプラグイン可能なサービスであり、特殊なニーズがある場合など開発者が独自の永続メカニズムにプラグインすることができます。JBossESB で提供される実装はデータベース永続メカニズムになります。ファイル永続メカニズムなどが必要な場合は、単にこれを行う独自のサービスを記述し設定変更でデフォルトの動作をオーバーライドします。



注記

現在、メッセージストアはベース実装のみです。Red Hat では、メッセージストアの機能が高度な監査と管理要求に応えられるようにするためコミュニティおよびパートナーの方々と共同で作業を進めていく予定です。したがってこれは出発点となります。

5.2. データ変換

クライアントとサービスは同じボキャブラリーを使用して交信することがよくありますが、そうではない場合にはあるデータ形式から別の形式に即時対応による変換が必要になります。単一のデータ形式がすべてのビジネスオブジェクト、特に大規模な開発や長期にわたる開発において適しているとみなすのは非現実的です。したがって、あるデータ形式から別の形式への変換メカニズムを提供することが必要になってくるわけです。

Transformation Service と呼ばれています。**JBoss Enterprise Service Bus** のこのバージョンは **Smooks** をベースとしたすぐに使用できる Transformation Service が同梱されます。**Smooks** は変換実装および管理フレームワークになります。これにより、XSLT、Java などの形式で変換ロジックを実装できます。また、この変換ロジックの集約管理ができる管理フレームワークを実現します。



注記

Smooks に関する詳細は、サービスガイドの「メッセージ変換」の章と **Smooks ガイド** を参照してください。

5.3. コンテンツベースルーティング

JBoss Enterprise Service Bus は、送信元にメッセージを動的にルーティングする必要がある場合があります。以下のような場合に必要です。

- 元の送信先が利用できない
- サービスが移動された
- アプリケーションがコンテンツベース、時間、その他の属性でメッセージの送信先を管理する必要がある

コンテンツベースのルーティングの仕組みを使用して、任意の複雑なルールをもとにメッセージをルーティングします (これらのルールはXPath、Regex Content-Based Routing (CBR)、あるいはJBoss Rules 表記に定義可能です)。または、コンテンツベースのルーティング機能を提供します。

5.4. レジストリ

SOA のコンテキスト内で、**registry** はそのサービスに関する情報を格納するための集約点をアプリケーションやビジネスに提供します。そのクライアントに対して標準市場と同レベルの情報や同じ範囲におけるサービスの提供が求められます。理想的には **registry** は自動ディスカバリや e-commerce トランザクションの実行を容易にし、ビジネストランザクションの動的環境を実現する必要もあります。したがって、**registry** は単に「e-business ディレクトリ」というだけではなく、SOA インフラストラクチャの固有となるコンポーネントになります。

いろいろな意味で、**registry** は **JBoss Enterprise Service Bus** の心臓部となります。サービスはエンドポイント参照 (EPR) を自身で公開し、また使用されなくなるとエンドポイント参照を削除します。コンシューマーは **registry** を調べて現在のタスクに適したサービスの EPR を確定することができます。

第6章 サンプル

6.1. メッセージの使用方法

コンセプト的には、メッセージは SOA 開発の手段において非常に重要なコンポーネントで、クライアントとサービスの間で送信されるアプリケーション固有のデータを含んでいます。メッセージの内容は、サービスとそのクライアント群間の規定に関する重要な側面を表しています。本セクションでは、メッセージに関するベストプラクティスや使用法について説明します。

まず、フライト予約サービスを使用する次のサンプルを見てみましょう。このサービスは以下の操作をサポートします。

reserveSeat

これはフライト番号と座席番号をとり、成功または失敗の表示を返します。

querySeat

これはフライト番号と座席番号をとり、その座席が現在予約されているかどうかの表示を返します。

upgradeSeat

これはフライト番号と 2 つの座席番号をとります (現在予約されている座席と移動先の座席)。

このサービスを開発する場合、ビジネスロジックを実装するために *Enterprise Java Beans (EJB3)* や *Hibernate* などの技術を使用する可能性が高くなります。このサンプルでは、どのようにビジネスロジックが実装されるのかについては触れずサービスの方に集中することとします。

このサービスの役割は、ロジックをそのバスに接続することです。これを行うためには、クライアントに対して定義する規定など、サービスがバスに対してどのように公開されるかを判断する必要があります。現在の **JBoss Enterprise Service Bus** バージョンでは、この規定はクライアントとサービスが交換できるメッセージの形式をとっています。ESB 内ではこの規定に関する公式な仕様はありません。現在、仕様は開発者が定義し、ESB から帯域外でクライアントと通信しなければなりません。これについては今後のリリースで修正される予定です。



注記

現在、ESB にはこのコントラクトに対する正式な仕様はありません。言い換えると、開発者が Enterprise Service Bus に関係なく定義し、クライアントとやり取りを行うものです。この点については、今後のリリースで修正されます。

6.1.1. メッセージの構造

サービスの観点では、メッセージ内のすべてのコンポーネント中でボディが最も重要となります。これは、ビジネスロジック固有の情報を伝達するためにボディが使用されるからです。対話するには、クライアントとサービスの両方がお互いを理解しなければなりません。これには、トランスポートに同意する形式 (JMS や HTTP など) やダイアレクトに同意する形式 (メッセージデータの表示や対応する形式) をとります。

クライアントがメッセージをフライト予約サービスに直接送信するといったシンプルなケースの場合、メッセージに関するオペレーションをサービスが判断する方法を確定する必要があります。この場合、`org.example.flight.opcode` と呼ばれる場所にて `opcode` (オペレーションコード) が文字列 (`reserve`、`query`、`upgrade`) としてボディ内に表示されることを開発者が決定します。その他の文字列値 (または値の指定がない場合) は不正なメッセージとしてみなされます。



重要

メッセージ内のすべての値に固有の名前を与えて他のクライアントやサービスとのクラッシュを避けるのが重要となります。

メッセージボディはクライアントとサービス間でデータが交換される主要な手段となります。すべての任意データタイプを格納することができる十分な柔軟性を持ち合わせています (各オペレーションに関連するビジネスロジックを実行するために必要となる他のパラメーターも適切にエンコードされます)。

- 座席番号には **org.example.flight.seatnumber** を使用します (整数)。
- フライト番号には **org.example.flight.flightnumber** を使用します (文字列)。
- アップグレードする座席番号には **org.example.flight.upgradenumber** を使用します (整数)。

表6.1 オペレーションのパラメータ

オペレーション	opcode	seatnumber	flightnumber	upgradenumber
reserveSeat	String: reserve	整数	文字列	N/A
querySeat	String: query	整数	文字列	N/A
upgradeSeat	String: upgrade	整数	文字列	整数

前述の通り、これらのオペレーションはすべてクライアントに情報を返します。こうした情報は同様にメッセージ内でカプセル化されます。ここで説明するプロセスと同じやり方で、こうした応答メッセージの形式が判断されます。説明が難しくなるため、ここでは応答メッセージについては考慮しません。

JBossESB のアクションの観点では、サービスは 1 つ以上のアクションを使用して構築することができます。たとえば、メインのビジネスロジックに関与するアクションに受信メッセージを渡す前に、アクションは受信メッセージを事前処理し、その内容を何らかの方法で変換することができます。各アクションは分離して記述されている可能性があります (同じ構成内の別のグループまたは完全に異なる構成など)。各アクションは対応する Message データの独自のビューを持つことが重要となります。そうでない場合、チェーンされたアクションによって上書きされたり、アクション同士が干渉し合う可能性があります。

6.1.2. サービス

この時点でサービスを構成できるくらいまで学習されました。シンプルにするため、ビジネスロジックは次の疑似オブジェクト内でカプセル化されていると仮定します。

```
class AirlineReservationSystem
{
    public void reserveSeat (...);
    public void querySeat (...);
    public void upgradeSeat (...);
}
```




注記

ビジネスロジックは POJO、EJB、Spring など開発が可能です。**JBoss Enterprise Service Bus** ではこれら多くの手段について特に設定をすることなくそのままの状態によるサポートを提供します (関連のドキュメントとサンプルをお読みください)。

サービスアクションの処理は以下のようになります。

```
public Message process (Message message) throws Exception
{
    String opcode = message.getBody().get("org.example.flight.opcode");

    if (opcode.equals("reserve"))
        reserveSeat(message);

    else if (opcode.equals("query"))
        querySeat(message);

    else if (opcode.equals("upgrade"))
        upgradeSeat(message);

    else
        throw new InvalidOpcode();

    return null;
}
```



注記

WS-Addressing と同様、メッセージ内に組み込まれた opcode ではなく、メッセージヘッダーの action フィールドを使用することができます。この欠点は、複数の JBossESB アクションがチェーン化され、各アクションに異なる opcode が必要な場合は機能しないことです。

6.1.3. ペイロードのデコーディング

ご覧の通り、process メソッドはスタート地点にすぎません。次に、受信のメッセージペイロードをデコードするメソッドを提供しなければなりません。以下のように行います。

```
public void reserveSeat (Message message) throws Exception
{
    int seatNumber = message.getBody().get("org.example.flight.seatnumber");
    String flight =
        message.getBody().get("org.example.flight.flightnumber");

    boolean success =
        airlineReservationSystem.reserveSeat(seatNumber, flight);

    // now create a response Message
    Message responseMessage = ...

    responseMessage.getHeader().getCall().setTo(
        message.getHeader().getCall().getReplyTo()
    );
}
```



```

    responseMessage.getHeader().getCall().setRelatesTo(
        message.getHeader().getCall().getMessageID()
    );

    // now deliver the response Message
}

```

このメソッドは、ボディ内の情報がどのように抽出され、ビジネスロジックでメソッドを呼び出すのに使用されるかを表しています。**reserveSeat** のケースでは、クライアントにより応答が予期されます。この応答メッセージは、ビジネスロジックにより返される情報や、元の受信したメッセージより取得した配信情報を使って構成されます。この例の場合、受信メッセージの ReplyTo フィールドより取得する To アドレスが応答に必要となります。また、応答を元の要求に関連させる必要がありますが、これは応答の RelatesTo フィールドと要求の MessageID を使用して関連させます。

サービスでサポートされるこの他すべてのオペレーションは同様にコード化されます。

6.1.4. クライアント

サービスによりサポートされるメッセージ定義があれば、クライアントコードを作成できます。サービスをサポートするために使用するビジネスロジックは、サービスによって直接公開されることはありません (SOA の重要な原則の 1 つであるカプセル化に反するため)。クライアントコードは、基本的にサービスコードの反対になります。

```

ServiceInvoker flightService = new ServiceInvoker(...);
Message request = // create new Message of desired type

request.getBody().add("org.example.flight.seatnumber", 1);
request.getBody().add(" org.example.flight.flightnumber", "BA1234");

request.getHeader().getCall().setMessageID(1234);
request.getHeader().getCall().setReplyTo(myEPR);

Message response = null;

do
{
    response = flightService.deliverSync(request, 1000);

    if (response.getHeader().getCall().getRelatesTo() == 1234)
    {
        // it's out response!

        break;
    }
    else
        response = null; // and keep looping
} while maximumRetriesNotExceeded;

```




注記

上記の多くは、従来のクライアント／サーバスタブジェネレーターと同様であるように見えるかもしれませんが、これらのシステムでは、(opcodes やパラメーターなどの) 低レベルの詳細は高レベルのスタブ抽象化の背後に隠されることになります。JBossESBの今後のリリースでは、開発アプローチを緩和するためこうした抽象化をサポートしていく予定です。ボディやヘッダーなどローメッセージコンポーネントの作業は、ほとんどの開発者から見えなくなります。

6.1.5. リモートサービス呼び出しの設定

カスタマイズなしに ESB のアクションから **ServiceInvoker** を使用できることがわかります。別途設定は必要ありません。しかし、リモートの Java 仮想マシンから使用するには (スタンドアローンの Java アプリケーション、サーブレット、Enterprise Java Bean などの場合)、まず以下の **JAR** ファイルが利用できる状態であるかを確認する必要があります。

jbossesb-rosetta.jar	trove.jar
jbossesb-config-model-[version].jar	juddi-client-[version].aop.jar
jbossts-common.jar	juddi-core-[version].aop.jar
log4j.jar	commons-configuration-[version].jar
stax-ex.jar	commons-lang-[version].jar
stax-api-[version].jar	jboss-messaging-client.jar
jbossall-client.jar	jboss-remoting.jar
scout-[version].aop.jar	commons-codec-[version].jar
xbean-[version].jar	wstx.jar
commons-logging.jar	xercesImpl.jar
javassist.jar	



注記

これらのファイルは、**\$SOA_HOME/jboss-as/client/**, **\$SOA_HOME/jboss-as/common/lib/** と **\$SOA_HOME/jboss-as/server/\$SOA_CONF/deployers/esb.deployer/lib/** にあります。

次に、以下のファイルがクラスパスにあるか確認します(**quickstarts** ディレクトリにあります)。

- **jbossesb-properties.xml**
- **META-INF/uddi.xml**

6.1.6. サンプルクライアント

以下の Java プログラムを使用して、リモートクライアントの設定が正しく機能しているかどうかを確認します (まず、**helloworld** quick-start がデプロイされており、Enterprise Service Bus サーバーが実行中であることを確認します)。

```
package org.jboss.esb.client;

import org.jboss.soa.esb.client.ServiceInvoker;
import org.jboss.soa.esb.listeners.message.MessageDeliverException;
import org.jboss.soa.esb.message.Message;
import org.jboss.soa.esb.message.format.MessageFactory;

public class EsbClient
{
    public static void main(String[] args)
    {
        System.setProperty("javax.xml.registry.ConnectionFactoryClass",
            "org.apache.ws.scout.registry.ConnectionFactoryImpl");

        try
        {
            Message message = MessageFactory.getInstance().getMessage();
            message.getBody().add("Sample payload");
            ServiceInvoker invoker = new
ServiceInvoker("FirstServiceESB", "SimpleListener");
            invoker.deliverAsync(message);
        }
        catch (final MessageDeliverException e)
        {
            e.printStackTrace();
        }
    }
}
```

6.1.7. コツとヒント

以下のヒントやコツは、クライアントとサービスの開発に役立つはずです。

- アクションを開発する際、アクション固有のペイロード情報は必ずメッセージボディ内の独自の場所で維持するようにしてください。
- メッセージ内でバックエンドサービス実装の詳細を公開しないようにしてください。公開してしまうと、クライアントに影響を与えずに実装を変更することが難しくなります。実装にとらわれないメッセージ定義 (内容や形式など) は、疎結合を維持できるようにします。
- ステートレスサービスの場合は、フェールオーバーを不透明に処理するため **ServiceInvoker** を使用します。
- 要求／応答のアプリケーションを構築する場合は、メッセージヘッダー内で相互関係の情報 (MessageID と RelatesTo) を使用します。
- メインのサービス opcode に Header Action フィールドを使用するように考慮してください。
- 応答用の配信アドレスのない非同期の対話を使用する場合、後で監視できるようすべてのエラーを **message store** に送信することを考慮してみてください。

- **JBoss Enterprise Service Bus** がサービス規定の定義や公開に対してより自動的なサポートを提供できるようになるまで、開発者とユーザーが使用できる定義のレポジトリを個別に維持するようにしてください。

第7章 高度なトピック

JBoss Enterprise Service Bus 関連の高度なコンセプトについては本章を参照してください。

7.1. フェールオーバーと負荷分散のサポート

ミッションクリティカルなシステムでは、冗長性を念頭に置いた設計が重要となります。**JBoss Enterprise Service Bus** にはビルトインのフェールオーバー、負荷分散そして堅固なアーキテクチャーの構築に役立つ遅延メッセージ再配信機能が含まれます。SOA を使用する場合、サービスはビルディングユニットとなることを意味します。JBoss Enterprise Service Bus では多くのノードにわたりまったく同一となるサービスのリプリケーションが可能になります。各ノードは JBossESB のインスタンスを実行している仮想のマシンまたは物理マシンになり得ます。こうした JBossESB 全インスタンスの集合はバスと呼ばれます。このバス内のサービスはメッセージの交換に異なる配信チャンネルを使用します。ESB 用語では、こうしたチャンネルは JMS、FTP、HTTP などになります。このような異なる「プロトコル」はシステムにより外部的に ESB の JMS プロバイダーや FTP サーバーなどに提供されます。サービスは 1 つのプロトコルまたは複数のプロトコルをリスンするよう設定可能です。リスンするよう設定される各プロトコルに対して、ESB は **registry** 内にエンドポイント参照 (EPR) を作成します。

7.1.1. サービス、エンドポイント参照 (EPR)、リスナー、アクション

jboss-esb.xml ファイル内では、各サービス要素は 1 つ以上のリスナーやアクションで構成されています。以下の設定 (一部) は、**JBossESBHelloWorld** の例では、ある程度これをベースとしています。サービスが初期化されると、カテゴリ、名前、説明を **UDDI registry** に登録します。そして、各リスナー要素に対して **ServiceBinding** を UDDI に登録します。エンドポイント参照にこれらを保存します。

```
...
<service category="FirstServiceESB" name="SimpleListener"
description="Hello World">
  <listeners>
    <jms-listener name="helloWorld" busidref="quickstartEsbChannel"
maxThreads="1"/>
  </listeners>
  <actions>
    <action name="action1"
class="org.jboss.soa.esb.actions.SystemPrintln"/>
  </actions>
</service>
...
```

カテゴリとサービス名が渡されると、別のサービスが **Hello World registry** を検索して、メッセージを送信できます。**JMSEPR** を受け取り、それを使ってメッセージを送信できます。こうした負荷の大きい作業はすべて **ServiceInvoker** クラスで行われます。HelloWorld サービスが **quickstartEsbChannel** 経由でメッセージを受け取ると、**pipeline** の最初のアクションである **SystemPrintln** に所属するプロセスメソッドに渡します。

7.1.2. 複製されたサービス

この例では、サービスは **Node1** で実行しています。**helloworld.esb** ファイルを取り、**Node2** にもデプロイした (下例を参照) 場合、**Node2** 側で **FirstServiceESB - SimpleListener** サービスがすでに登録されていることがわかります。そのため、2 つ目の **ServiceBinding** をこのサービスに追加して、複製します。**Node1** が失敗すると、**Node2** が継続して機能します。

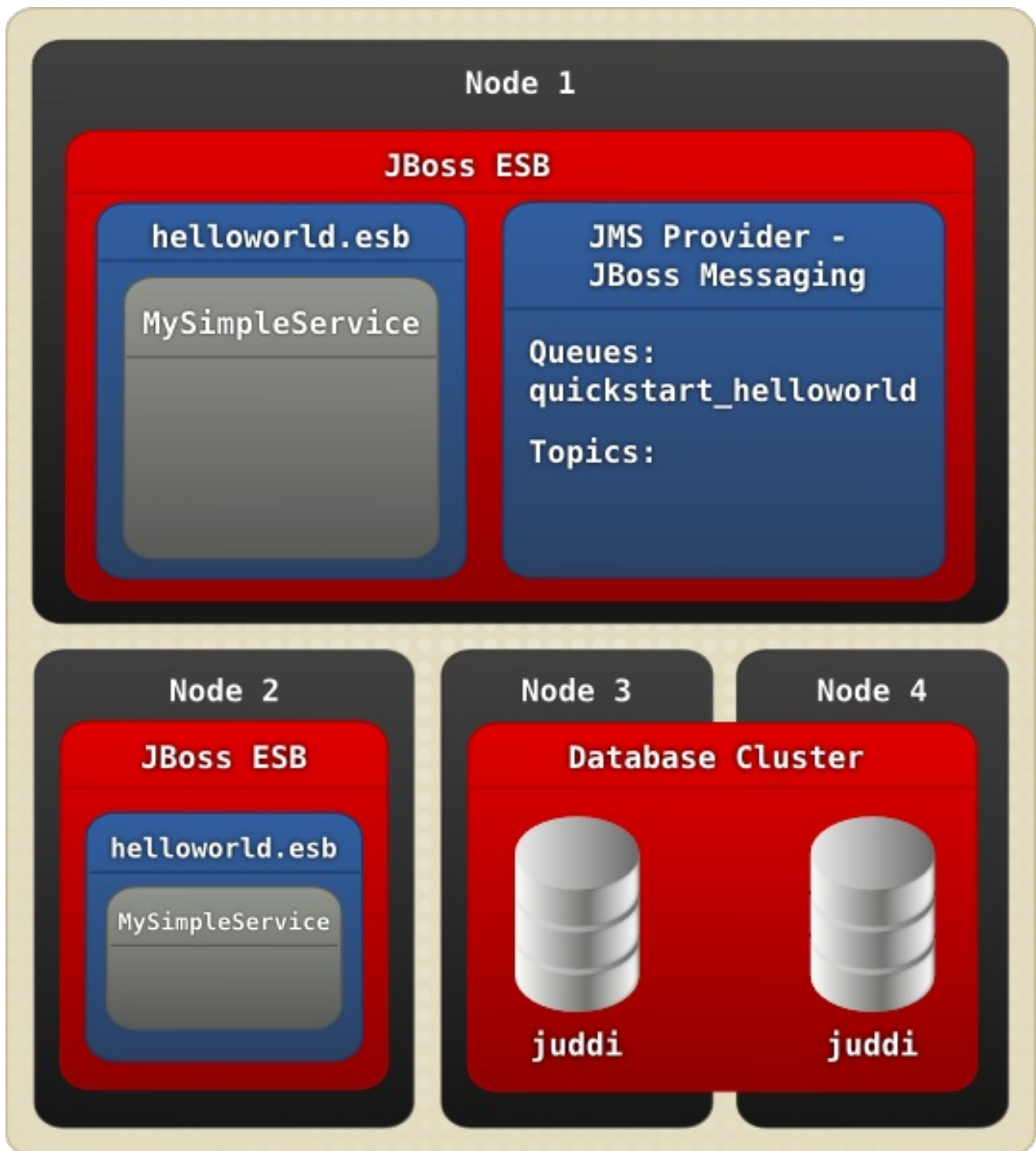


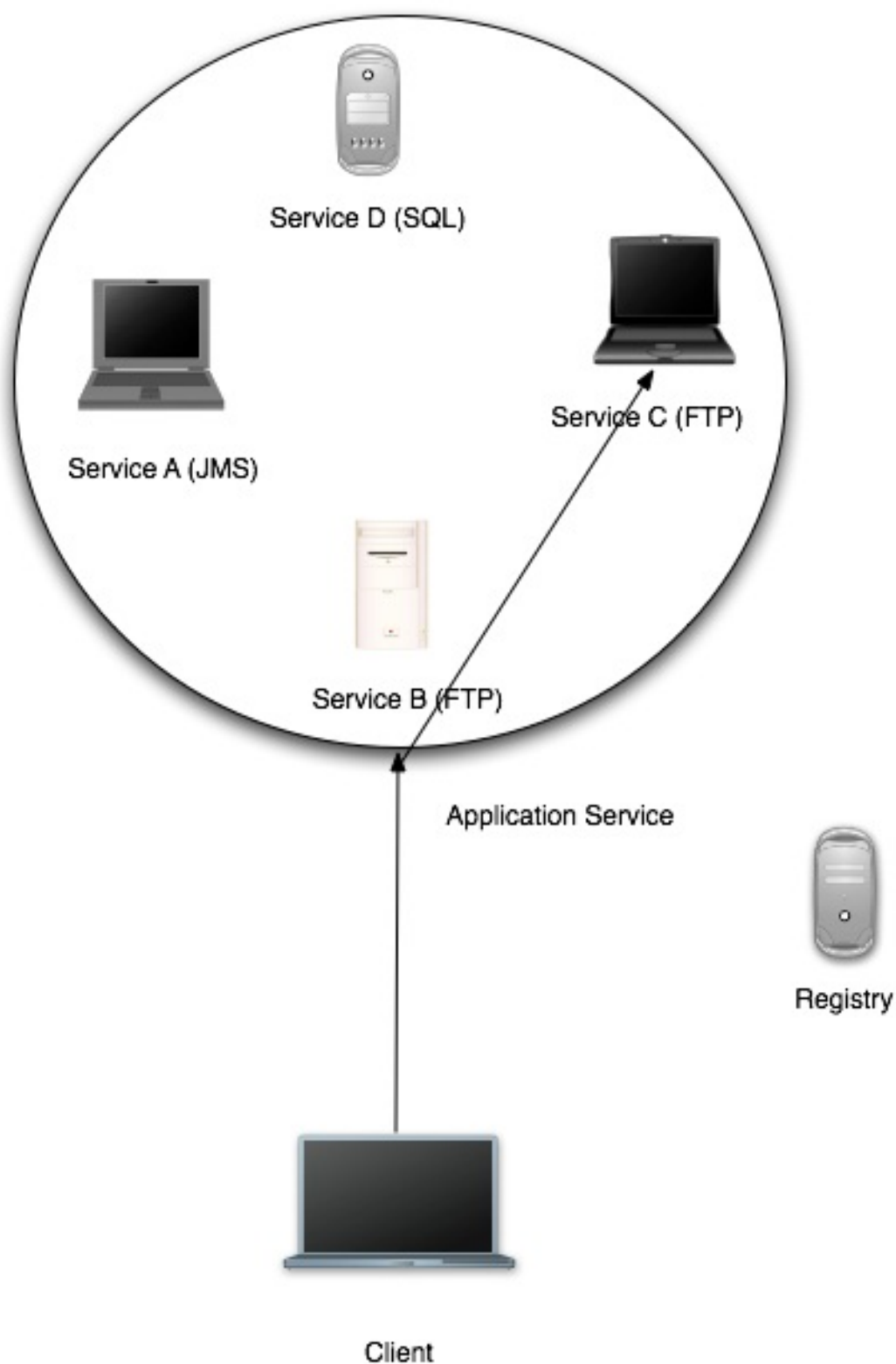
図7.1 異なるノードにある 2 つのサービスインスタンス

両方のサービスインスタンスが同じキューをリッスンするため負荷分散機能を得ることになります。ただし、まだこのセットアップでは単一障害点があることになります。このため、次のセクションで説明するプロトコルクラスタリングがひとつの選択肢となります。

このタイプのレプリケーションを使用してサービスの利用度を向上したり負荷分散機能を提供することができます。詳しく説明するため、論理サービス (アプリケーションサービス) を持つ以下の図を見てください。実際には 4 つの個別なサービスから構成され、それぞれ同じ機能を提供し同じサービス規定に準じています。異なるのは同じトランスポートプロトコルを共有する必要がないということだけです。しかし、アプリケーションサービスのユーザーに関する限り、ユーザーにはサービス名とカテゴリで識別される単一のサービスしか見えません。**ServiceInvoker** はアプリケーションサービスが実際には 4 つのサービスから構成されているという事実をクライアントには見えないようにしています。これにより個別サービスの障害が隠され、複製されるサービスグループの少なくとも 1 インスタンスが利用できる状態にある限りクライアントは先に進むことができます。

**重要**

このタイプのレプリケーションはステートレスのサービスにのみ使用してください。



プロバイダーはサービスコンシューマー側のサービスを自主的に複製する場合があります。しかし、サービスが自動的にレジストリ内で定義された代替サービスにフェールオーバーされるのを好まない場合があります。自動的なフェールオーバーを防ぐにはメッセージプロパティ `org.jboss.soa.esb.exceptionOnDeliverFailure` を **true** に設定します。このプロパティを設定すると、

メッセージを再送信する代わりに **ServiceInvoker** によって **MessageDeliverException** がスローされます。すべてのメッセージにこれを指定するには、JBoss Enterprise Service Bus **property** ファイルの Core セクションでこのプロパティを設定します。

7.1.3. プロトコルのクラスタリング

JMS プロバイダーの中にはクラスタ化が可能なものがあります。**JBossMessaging** もこうしたプロバイダーのひとつで、Enterprise Service Bus の JMS プロバイダーとしてこれを使用するのもその理由です。JMS をクラスタ化することでアーキテクチャーから単一障害性を排除します。次の図を参照してください。

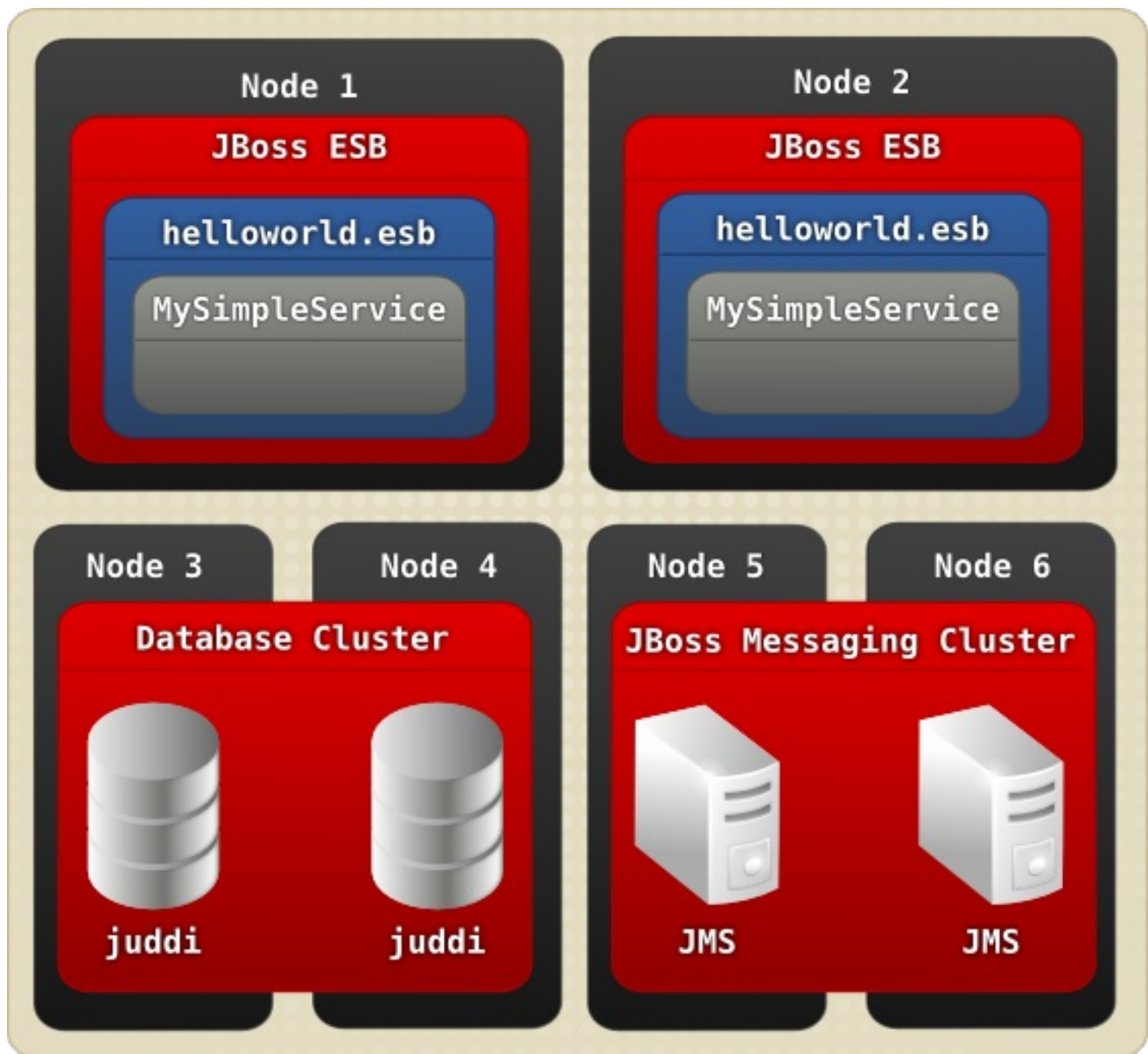
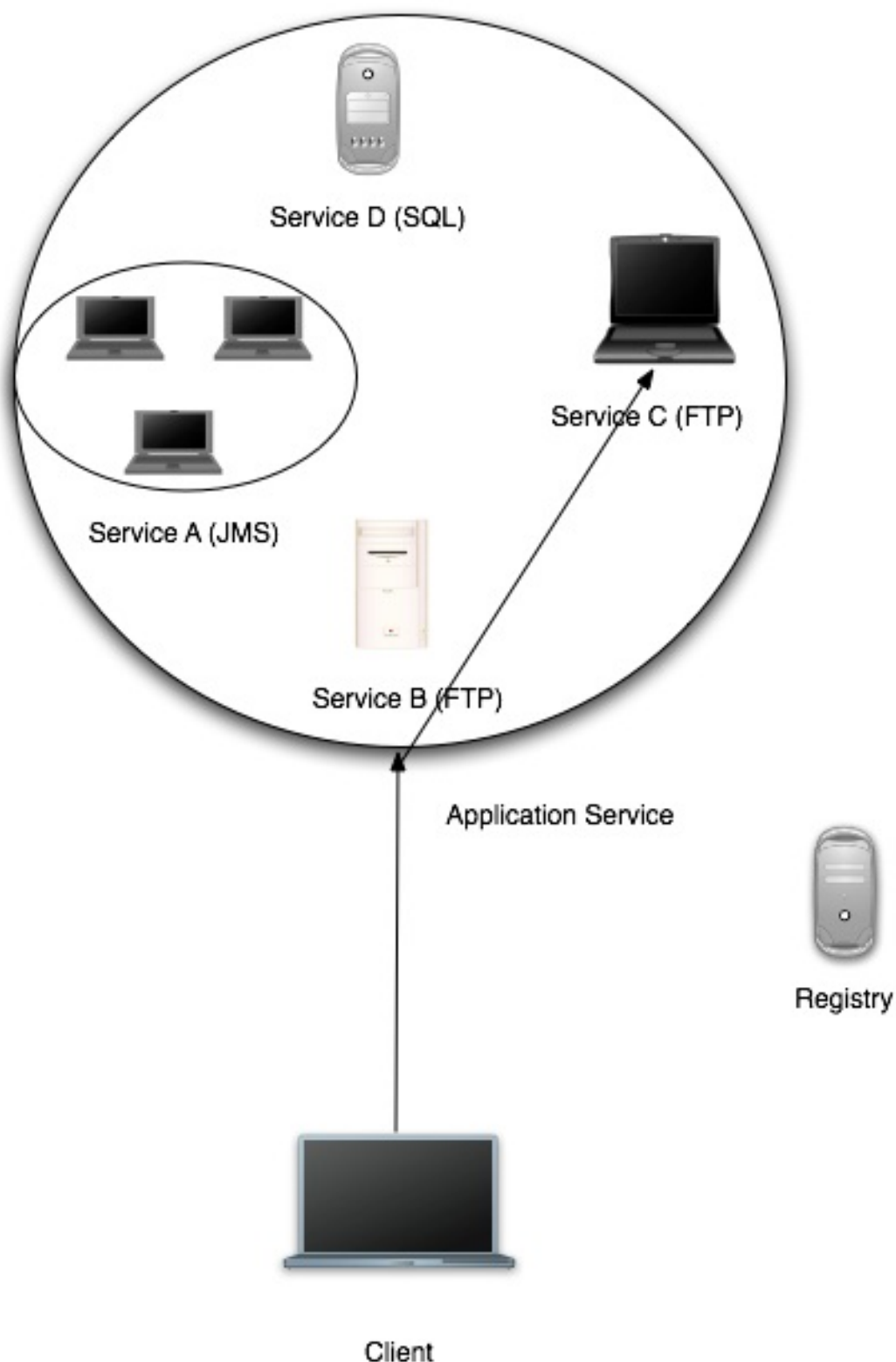


図7.2 JMS を使用したプロトコルクラスタリングの例

JMS クラスタリングを有効にしたい場合は JBossMessaging のクラスタリングに関するドキュメントをお読みください。JBossESB レプリケーションと JMS クラスタリングは以下の図に示すように一緒に使用することができます。この例では、サービス A が単一の JMSEpr によりレジストリ内で指定されています。しかし、クライアントには不透明にその JMSEpr はクラスタ化された JMS キューを参照し、これは 3 つのサービスをサポートするため別々に設定されています。利用度や負荷分散機能に対する連合的な手段となります。実際、サービスのレプリケーションをユーザー (JBossESB レプリケー

ションの方法の場合はクライアントであり、JMS クラスタリングの場合は JBossESB) に見えないようにするのは SOA の原則と一致しています。これら実装の詳細をサービスエンドポイントの背後に隠し規定レベルでは公開しません。



注記

JMS クラスタリングをこの方法で使用している場合、明らかに設定が正しく行われているかを確認する必要があります。たとえば、ESB サービスをすべて JMS クラスター内に配置する場合は ESP レプリケーションの利点を生かすことはできません。

プロトコルクラスタリングの別の例としては **FileSystem** プロトコルの NAS (*Network Attached Storage*) がありますが、ご使用のプロバイダーが単純にクラスタリングをまったく提供できない場合はどうでしょう。このような場合には、サービスに複数のリスナーを追加し、複数の (JMS) プロバイダーを使用することができます。ただし、これにはフェールオーバーと負荷分散機能がプロバイダー全体に必要となります。それでは、これについて次のセクションで見ていくことにします。

7.1.4. クラスタリング

クラスター内の複数のノードで同じサービスを実行したい場合、サービスが完全にクラスター化された環境で動作する前にサービスレジストリキャッシュの再検証を待たなければなりません。このキャッシュ再検証のタイムアウトは **deploy/jbossesb.sar/jbossesb-properties.xml** でセットアップできます。

```
<properties name="core">
  <property name="org.jboss.soa.esb.registry.cache.life" value="60000"/>
</properties>
```

デフォルトのタイムアウトは 60 秒です。

7.1.5. チャンネルのフェールオーバーと負荷分散機能

HelloWorld サービスはプロトコル 1 つ以上のリッスンが可能です。以下では FTP チャンネルを追加しています。

```
...
<service category="FirstServiceESB" name="SimpleListener"
description="Hello World">
  <listeners>
    <jms-listener name="helloWorld" busidref="quickstartEsbChannel"
maxThreads="1"/>
    <jms-listener name="helloWorld2" busidref="quickstartFtpChannel2"
maxThreads="1"/>
  </listeners>
...

```

これでサービスは 2 つの JMS キューを同時にリッスンすることになります。これらのキューは物理的に異なるマシンで JMS プロバイダーにより提供することができます。つまり、2 つのサービス間の冗長な JMS 接続を確立したということになります。この設定ではプロトコルを混合することも可能なため、FTP リスナーを追加することもできます。

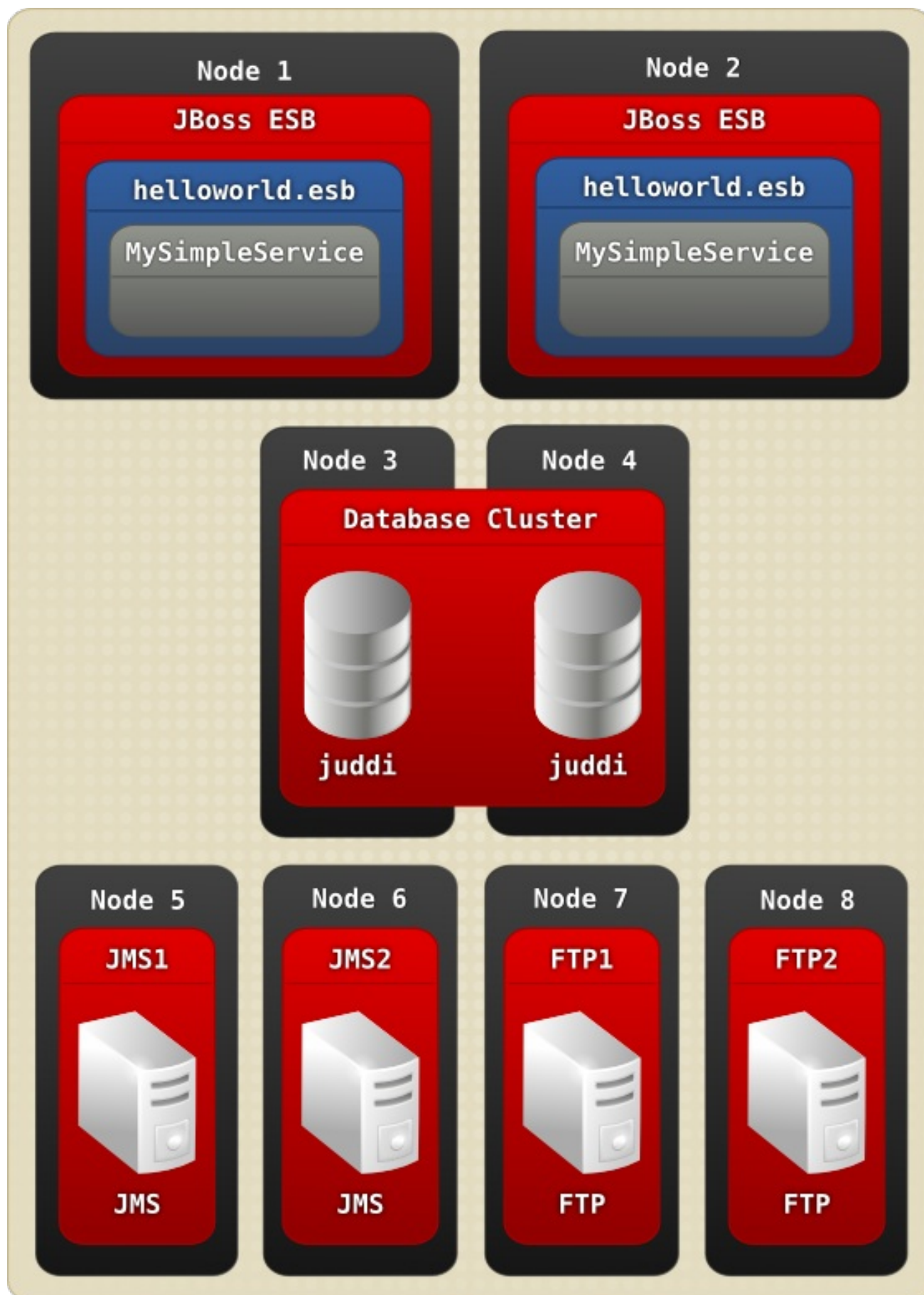


図7.3 2つのFTPサーバーをその混合に追加する

```
...  
<service category="FirstServiceESB" name="SimpleListener"
```



```

description="Hello World">
<listeners>
  <jms-listener name="helloWorld" busidref="quickstartEsbChannel"
maxThreads="1"/>
  <jms-listener name="helloWorld2" busidref="quickstartJmsChannel2"
maxThreads="1"/>
  <ftp-listener name="helloWorld3" busidref="quickstartFtpChannel3"
maxThreads="1"/>
  <ftp-listener name="helloWorld4" busidref="quickstartFtpChannel3"
maxThreads="1"/>
</listeners>
...

```

ServiceInvoker がサービスにメッセージを配信する場合、8 個の EPR から選択することになります (Node1 から 4 EPR, Node2 から 4 EPR)。どれを使用するのかどのように決定するのでしょうか。これについてはポリシーを設定することができます。**jbossesb-properties.xml** で「org.jboss.soa.esb.loadbalancer.policy」を設定することができます。ここでは 3 つのポリシーが提供されています。また、独自のポリシーを作成することも可能です。

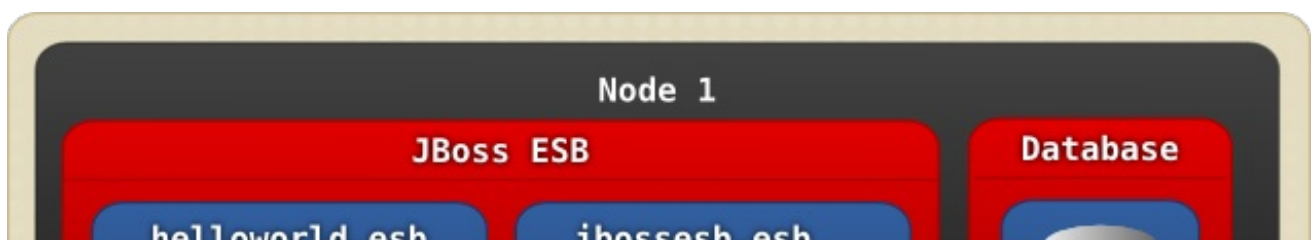
- 1 番目に利用可能。問題のない ServiceBinding が見つかりそれが終了しない限り使用され、一覧内の次の EPR に移動します。このポリシーは 2 つのサービスインスタンス間での負荷分散は提供しません。
- ラウンドロビン。一般的な負荷分散ポリシーで、各 EPR は一覧の順でヒットされます。
- ランダムロビン。他のロビンと似ていますがランダムになります。

ポリシーが動作する EPR 一覧は、終了した EPR が (キャッシュされた) 一覧から削除されていくため小さくなっていく場合があります。一覧が空になった場合または一覧のキャッシュの time-to-live を越えた場合、ServiceInvoker はレジストリから真新しい EPR の一覧を取得します。**org.jboss.soa.esb.registry.cache.life** は jbossesb-properties ファイルで設定でき、デフォルトでは 60,000 ミリ秒に設定されます。その時点で動作している EPR がない場合はメッセージ再配信サービスを利用することができます。

7.1.6. メッセージ再配信

EPR の一覧に終了した EPR 以外何も含まれていない場合、ServiceInvoker は以下の 2 つのうちいずれかを行うことができます。

- メッセージを同期的に配信しようとしている場合、メッセージを DeadLetterService に送信します。デフォルトでは DLQ MessageStore に格納し、呼び出し側にエラーを返送します。処理は停止します。たとえば、JMS キューに送りたい、あるいは通知を受け取りたい場合などは、jbossesb.esb 内の DeadLetterService を設定することができます。
- メッセージを非同期的に送信しようとしている場合 (推奨)、この場合もメッセージを DeadLetterService に送信しますがそのメッセージは RDLVR MessageStore に格納されます。再配信サービス (jbossesb.esb) は再配信試行の最大数に達するまでメッセージ送信の再試行を行います。最大数に達すると、メッセージは DLQ MessageStore に格納され処理は停止します。



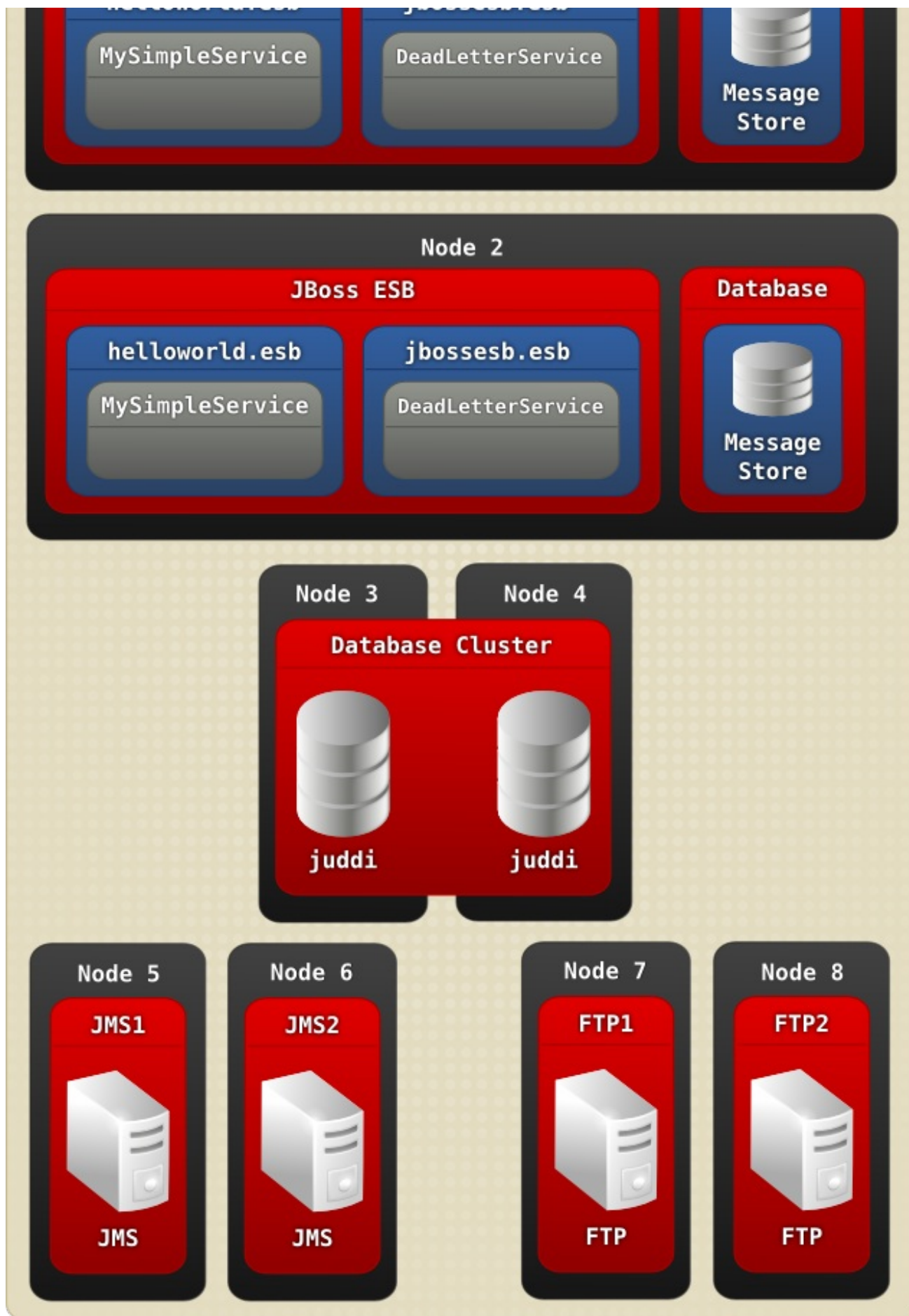


図7.4 メッセージ再配信



注記

DeadLetterService はデフォルトでは、オンとなっていますが、**jbossesb-properties.xml** で、**org.jboss.soa.esb.dls.redeliver** を "false" にするとオフにできます。メッセージベースでこれを管理するには、各メッセージのプロパティで **org.jboss.soa.esb.dls.redeliver** プロパティを設定してください。グローバル設定より優先して、Message プロパティが使用されます。デフォルトは、設定ファイルに設定された値を使用します。

7.2. サービスのスケジュール

JBoss は 2 種類のプロバイダーをサポートしています。

1. バスプロバイダー、JMS や HTTP などメッセージングプロトコルを介してアクション処理パイプラインにメッセージを供給します。このプロバイダータイプは基礎となるメッセージングプロバイダーにより起動されます。
2. スケジュールプロバイダー、スケジュール駆動のモジュールに基づいてアクション処理パイプラインにメッセージを供給します。つまり、基礎となるメッセージ配信のメカニズム (ファイルシステムなど) はメッセージが処理可能になる場合に ESB 起動に対するサポートを提供せず、スケジューラーが定期的にリスナーを起動して新しいメッセージをチェックします。

JBoss ESB は `<schedule-listener>` の他、2 つの `<schedule-provider>` タイプ、`<simple-schedule>` と `<cron-schedule>` を提供しています。`<schedule-listener>` は「composer」クラスで設定され、**org.jboss.soa.esb.listeners.ScheduledEventMessageComposer** インターフェースの実装になります。

7.2.1. Simple Schedule

このスケジュールタイプは次の属性に基づく簡単なスケジュール機能を提供します。

scheduleid

スケジュール用の固有の識別子文字列になります。リスナーからスケジュールを参照するために使用されます。

frequency

すべてのスケジュールリスナーが起動されるべき頻度 (秒単位) になります。

execCount

スケジュールが実行されるべき回数になります。

startDate

スケジュールの開始日と時間です。この属性値の形式は XML スキーマタイプの「dateTime」になります。dateTime を参照してください。

endDate

スケジュールの終了日と時間です。この属性値の形式は XML スキーマタイプの「dateTime」になります。dateTime を参照してください。

例:


```
<providers>
  <schedule-provider name="schedule">
    <simple-schedule scheduleid="1-sec-trigger" frequency="1" execCount="5"
  />
  </schedule-provider>
</providers>
```

7.2.2. Cron Schedule

このスケジュールタイプは Quartz Scheduler CronTrigger 式に基づいたスケジュール機能を提供します。このスケジュールタイプの属性は以下の通りです。

scheduleid

スケジュール用の一意識別子の文字列になります。リスナーからスケジュールを参照するために使用されます。

cronExpression

これは、Quartz Scheduler CronTrigger 式です。

startDate

スケジュールの開始日と時間です。この属性値の形式は XML スキーマタイプの「dateTime」になります。

endDate

スケジュールの終了日と時間です。この属性値の形式は XML スキーマタイプの「dateTime」になります。

以下に例を示します。

```
<providers>
  <schedule-provider name="schedule">
    <cron-schedule scheduleid="cron-trigger" cronExpression="0/1 * * * * ?"
  />
  </schedule-provider>
</providers>
```

この例は、毎分 1 秒目にスケジュールがトリガーされます。

7.2.3. Scheduled Listener

`<scheduled-listener>` または `<cron-schedule>` の設定に基づいてスケジュールされたタスクを実行するには `<scheduled-listener>` を使用することができます。

event-processor クラスで設定さ

れ、**org.jboss.soa.esb.schedule.ScheduledEventListener** または

org.jboss.soa.esb.listeners.ScheduledEventMessageComposer のいずれかの実装になります。

ScheduledEventListener

このインターフェースを実装するイベントプロセッサは単純に「onSchedule」メソッドで起動されます。アクション処理パイプラインは実行されません。

ScheduledEventMessageComposer

このインターフェースを実装するイベントプロセッサはリスナーに関連付けられるアクション処理パイプラインのメッセージを「composing」する機能があります。

このリスナーの属性は次の通りです。

1. name: リスナーインスタンスの名前です。
2. event-processor: 各スケジュール起動で呼び出されるイベントプロセッサークラスです。実装詳細については上記を参照してください。
3. いずれかひとつ
 1. name: リスナーインスタンスの名前です。
 2. scheduleidref: このリスナーの起動に使用するスケジュールの scheduleid です。
 3. schedule-frequency: スケジュールの頻度 (秒単位) です。直接リスナーで簡単にスケジュールを指定する便利な方法になります。

7.2.4. 設定例

<scheduled-listener> と <cron-schedule> に関連する設定例を示します。

```
<?xml version = "1.0" encoding = "UTF-8"?>
<jbossesb
xmlns="http://anonsvn.labs.jboss.com/labs/jbossesb/trunk/product/etc/schemas/xml/jbossesb-1.0.1.xsd">

    <providers>
        <schedule-provider name="schedule">
            <cron-schedule scheduleid="cron-trigger" cronExpression="0/1
* * * * ?" />
        </schedule-provider>
    </providers>

    <services>
        <service category="ServiceCat" name="ServiceName"
description="Test Service">

            <listeners>
                <scheduled-listener name="cron-schedule-listener"
scheduleidref="cron-trigger"
event-processor="org.jboss.soa.esb.schedule.MockScheduledEventMessageComposer"
/>
            </listeners>

            <actions>
                <action name="action"
class="org.jboss.soa.esb.mock.MockAction" />
            </actions>
        </service>
    </services>
</jbossesb>
```



```

        </service>
    </services>

</jbossesb>

```

7.2.5. カスタムのゲートウェイの作成

本章では、Enterprise Service Bus のカスタムリスナーを構築する際に使用可能なメソッド 3 つについて説明しています。

1. **AbstractThreadedManagedLifecycle/AbstractManagedLifecycle Listener**: スレッド／非スレッドリスナーを作成するために、ベースリスナー API を使用するリスナー。管理ライフサイクルベースで実行
2. **Schedule Driven Listener**: 設定したスケジュールを元にトリガーするために `ScheduledEventListener` を使用し、Service Action pipeline のメッセージを生成するリスナー
3. **Groovy Scripted Event Driven Listener**: 外部プロセスでトリガーするイベントを元に起動して、Service Action Pipeline のメッセージを生成するリスナー (例：JMS キューで受信するメッセージ)

7.2.5.1. AbstractThreadedManagedLifecycle/AbstractManagedLifecycle リスナー

ESB リスナーはすべて `AbstractThreadedManagedLifecycle` か、`AbstractManagedLifecycle` クラスを使用して実装します。これらのクラスの継承は単純です。

```

public class MyCustomGateway extends AbstractThreadedManagedLifecycle {

    private ConfigTree listenerConfig;
    private Service service;
    private ServiceInvoker serviceInvoker;

    public MyCustomGateway(final ConfigTree config) throws
ConfigurationException {
        super(config);
        this.listenerConfig = config;

        String serviceCategory =
listenerConfig.getRequiredAttribute(ListenerTagNames.TARGET_SERVICE_CATEGO
RY_TAG);
        String serviceName =
listenerConfig.getRequiredAttribute(ListenerTagNames.TARGET_SERVICE_NAME_T
AG);

        service = new Service(serviceCategory, serviceName);
    }

    protected void doInitialise() throws ManagedLifecycleException {
        // Create the ServiceInvoker instance for the target service....
        try {
            serviceInvoker = new ServiceInvoker(service);
        } catch (MessageDeliverException e) {
            throw new ManagedLifecycleException("Failed to create
ServiceInvoker for Service '" + service + "'.");
        }
    }
}

```



```

    }

    protected void doRun() {
        while(isRunning()) {
            // Wait for a message....
            Object payloadObject = waitForPayload();

            // Send the message to the target service's Action Pipeline
via
            // the ServiceInvoker...
            try {
                Message esbMessage =
MessageFactory.getInstance().getMessage();

                esbMessage.getBody().add(payloadObject);
                serviceInvoker.deliverAsync(esbMessage);
            } catch (MessageDeliverException e) {
                e.printStackTrace();
            }
        }
    }

    private Object waitForPayload() {
        // Wait for a message...
    }
}

```

このゲートウェイは、AbstractThreadedManagedLifecycle クラスを継承して、doRun メソッド (Thread method) を実装します。実行ループの反復時において、doRun メソッドがリスナーの実行ステータスを確認する方法に注目してください。

スレッド化されたリスナーを必要としない場合、リスナーは AbstractManagedLifecycle を継承する必要があります。

このようにカスタムのゲートウェイを設定するには、基盤となる設定タイプのバスプロバイダーバスおよびリスナーを以下のように使用する必要があります。

```

<?xml version = "1.0" encoding = "UTF-8"?>
<jbossesb
xmlns="http://anonsvn.labs.jboss.com/labs/jbossesb/trunk/product/etc/schem
as/xml/jbossesb-1.0.1.xsd"
    parameterReloadSecs="5">

    <providers>
        <bus-provider name="CustomProvider">
            <property name="provider-property" value="buprovider-prop-
value" />

            <bus busid="custom-bus">
                <property name="bus-property" value="bus-prop-value" />
            </bus>
        </bus-provider>
    </providers>

    <services>

```



```

        <service category="Custom" name="Listener" description=""
invmScope="GLOBAL">
            <listeners>
                <listener name="custom-listener" busidref="custom-bus"
is-gateway="true">
                    <property name="gatewayClass"
value="com.acme.listeners.MyCustomGateway" />
                    <property name="listener-property" value="listener-
prop-value" />
                </listener>
            </listeners>
            <actions mep="OneWay">
                ...
            </actions>
        </service>
    </services>

</jbossesb>

```

7.2.5.2. スケジュール駆動型リスナー

simple-schedule or cron-schedule の設定に基づいてスケジュールされたタスクを実行するには scheduled-listener を使用することができます。

scheduled-listener は“event-processor”クラスで設定され、以下のインターフェースの実装になります。

- ScheduledEventListener: このインターフェースを実装するイベントプロセッサは単純に「onSchedule」メソッドで起動されます。アクション処理パイプラインは実行されません。
- ScheduledEventMessageComposer: このインターフェースを実装するイベントプロセッサはリスナーに関連付けられるアクション処理パイプラインのメッセージを「composing」する機能があります。

このリスナーの属性は次の通りです。

1. name: リスナーインスタンスの名前です。
2. event-processor: 各スケジュール起動で呼び出されるイベントプロセッサクラスです。実装詳細については上記を参照してください。
3. いずれかひとつ
 - scheduleidref: このリスナーの起動に使用するスケジュールの scheduleid です (プロバイダー内で設定)。
 - schedule-frequency: スケジュールの頻度 (秒単位) です。直接リスナーで簡単にスケジュールを指定する便利な方法になります。

この例では、「order」ファイルが必要で、ESB に同梱されているファイルリスナーのコンポーネントではニーズが満たされないと仮定します。この場合、ScheduledEventMessageComposer インターフェースを実装することで、独自のカスタムファイルリスナーを記述することができます。

```

Public class OrderFileListener implements ScheduledEventMessageComposer {

```



```

    public void initialize(ConfigTree config) throws
ConfigurationException {
        // TODO: Initialise File filters etc, using the config to access
properties configured on the listener...
    }

    public Message composeMessage() throws SchedulingException {
        Message message = MessageFactory.getInstance().getMessage();

        // TODO: Read one or more order files and populate the data into
the ESB message for pipeline processing...

        return message;
    }

    public Message onProcessingComplete(Message message) throws
SchedulingException {
        // TODO: Post pipeline processing...
        return message;
    }

    public void uninitialized() {
        // TODO: Any relevant cleanup tasks...
    }
}

```

ESB サービスに変換するには、このコードを使用します。

```

<?xml version = "1.0" encoding = "UTF-8"?>
<jbossesb
xmlns="http://anonsvn.labs.jboss.com/labs/jbossesb/trunk/product/etc/schem
as/xml/jbossesb-1.0.1.xsd">

    <providers>
        <schedule-provider name="schedules">
            <simple-schedule scheduleid="ordersPole" frequency="5"
frequencyUnits="seconds" />
        </schedule-provider>
    </providers>

    <services>
        <service category="OrderManagement" name="OrderProcessing"
description="Order Processing Service">

            <listeners>
                <scheduled-listener name="orderFileListener"
scheduleidref="ordersPole" event-processor="com.acme.OrderFileListener"/>
            </listeners>

            <actions>
                <action name="action1" class="..."/>
                <action name="action2" class="..."/>
                <action name="action3" class="..."/>
            </actions>
        </service>
    </services>
</jbossesb>

```



```
</services>

</jbossesb>
```

7.2.5.3. Groovy スクリプトのイベント駆動型リスナー

JBoss ESB でイベント駆動型リスナーをより簡単に実装するには、Groovy スクリプトと groovy-listener 設定経由でリスナーとつなぎます。

groovy-listener の設定は非常に簡単です。以下の設定属性をとります。

1. name: リスナーインスタンスの名前です。
2. script: Groovy スクリプトへのパス (クラスパス上)。

Groovy スクリプトは、事実上ゲートウェイになり、以下のスクリプト変数のバインディングにアクセスできます。

1. config: リスナーの設定 (ConfigTree)。ネスト化されたプロパティ要素の値。これは、アクションパイプラインの構築に必要です。
2. gateway: 基盤の GroovyGateway リスナー (Java) への参照。これにより、リスナーのライフサイクルにアクセスできます。

JBoss ESB 提供の JMS リスナー実装が要件を満たさない場合、Groovy スクリプトと groovy-listener でカスタムの JMS リスナーを組み込むことができます。

javax.jms.MessageListener を実装することで開始します。

```
public class OrderListener implements MessageListener {

    public void onMessage(final Message message) {
        if(message instanceof ObjectMessage) {
            Order order = ((ObjectMessage) message).getObject();

            // Create and populate an ESB message with the order....
            Message esbMessage =
            MessageFactory.getInstance().getMessage();
            esbMessage.getBody().add(order);

            // TODO: Add code to forward the ESB message to the Action
            Pipeline...
        }
    }
}
```

ESB で機能するようにリンクを追加する必要があります。リスナーのライフサイクルの管理、アクションパイプラインインスタンス (およびその他のリソース) の作成用に start と stop メソッドを作成する必要があります。

```
public class OrderListener implements MessageListener {

    private ActionProcessingPipeline pipeline;

    public void start(ConfigTree config) throws ConfigurationException {
        // Create and initialize the pipeline..
    }
}
```



```

        pipeline = new ActionProcessingPipeline(config);
        pipeline.initialise();

        // TODO: Add JMS code for connecting this JMS MessageListener to
the JMS Queue...
    }

    public void onMessage(final Message message) {
        if(message instanceof ObjectMessage) {
            Order order = ((ObjectMessage) message).getObject();

            // Create and populate an ESB message with the order....
            Message esbMessage =
MessageFactory.getInstance().getMessage();
            esbMessage.getBody().add(order);

            // Forward the ESB message to the Action Pipeline...
            boolean success = pipeline.process(message);
            if(!success) {
                // TODO: Handle error....
            }
        }
    }

    public void stop() {
        try {
            // TODO: Add JMS code for disconnecting from JMS Queue....
        } finally {
            if(pipeline != null) {
                pipeline.destroy() ;
            }
        }
    }
}

```

注記

ActionProcessingPipeline クラス経由で直接アクションパイプラインを実行できるように ServiceInvoker を使用できます。これは、本書にて前述した AbstractThreadManagedLifecycle の例と同じです。これは、メッセージに対応するリスナーをサービス上にインストールする必要があります (InVM リスナーで問題なし)、パイプラインが実装済みのリスナースレッドとは非同期で実行されることになります。

OrderListener を ESB にリンクする Groovy スクリプトを実装し、ライフサイクルを管理する (開始と停止) 必要があります。

```

import com.acme.OrderListener;

OrderListener orderListener = new OrderListener();

// Start the listener (passing the config)...
orderListener.start(config);

// Wait until the Groovy Gateway is signaled to stop...
def stopped = false;

```



```
while(!stopped) {
    stopped = gateway.waitUntilStopping(200);
}

// Now stop the listener...
orderListener.stop();
```

最後に、Groovy スクリプトを ESB サービスに設定するには、以下のコードを使用します。

```
<?xml version = "1.0" encoding = "UTF-8"?>
<jbossesb
xmlns="http://anonsvn.labs.jboss.com/labs/jbossesb/trunk/product/etc/schem
as/xml/jbossesb-1.0.1.xsd">

    <services>
        <service category="OrderManagement" name="OrderProcessing"
description="Order Processing Service">

            <listeners>
                <groovy-listener name="orderJmsListener"
script="/com/acme/OrderListener.groovy">
                    <property name="queueName" value="..." />
                </groovy-listener>
            </listeners>

            <actions>
                <action name="action1" class="..." />
                <action name="action2" class="..." />
                <action name="action3" class="..." />
            </actions>
        </service>
    </services>

</jbossesb>
```


第8章 耐障害性と信頼性

本章では、JBoss Enterprise Service Bus の信頼性に関する特徴を説明します。ここでは、本リリースの耐障害性に確認いただけます。また、アプリケーションの耐障害性を向上する方法に対するアドバイスも記載しています。しかし、進める前に重要な用語をまず定義していきます。

ディペンダビリティ とは、配信するサービス (ユーザーによって認識される動作) に正当な信頼が置けるなど、コンポーネントの信用性を意味します。コンポーネントの信頼性は、継続した適切なサービス配信を示す基準となります。システムが提供するサービスが仕様に適合しなくなると障害が発生します。エラーは障害の原因となるシステム状態の一部で、不良はエラーの原因となります。

耐障害性 システムとは、コンポーネントの障害が発生しても特定の目的を達成するように設計されたシステムです。通常、耐障害性を提供するための技術は、整合状態回復のメカニズムや故障したコンポーネントにより生成されるエラーを検出するメカニズムを必要とします。レプリケーションやトランザクションなど、複数の耐障害性の技術が存在します。

8.1. 障害の分類

システム上で稼働しているアプリケーションが適切であるか検証する前に、システムの動作を正式に記述する必要があります。これにより、アプリケーションの動作制限を確立し、この制限を緩和または強化する意味を明確にします。

耐障害性に関して正式な記述を行うメソッドとして、発生すると考えられる障害の種類によってシステムコンポーネントを分類するメソッドを推奨します。

各コンポーネントには特定の入力セットに対するそのコンポーネントの正しい動作を記す関連詳細があります。正常に機能するコンポーネントはこの詳細に一致する出力を生成します。障害のあるコンポーネントからの応答は、必ずしもそこまではっきり限定されているわけではありません。特定の入力に対する特定のコンポーネントからの応答は、指定の値が正しく、指定の制限時間内に生成されていれば、正しいとみなされます。

起こりうる障害を 4 つに分類すると、脱落、値、タイミング、任意になります。

脱落の不良／障害

コンポーネントが別のコンポーネントからの入力に応答せず、予期される出力を生成しない場合、**脱落の不良**とそれに関連する**脱落の障害**が発生していることになります。メッセージを紛失することのある通信リンクなどが、脱落の不良が発生しているコンポーネントの例になります。

値の不良／障害

正しい時間間隔内にコンポーネントが応答しても、値が正しくないような不良を**値の不良**と呼びます (これに関連する障害は**値の障害**と呼ばれます)。破損したメッセージを時間通りに配信する通信リンクには値の不良が発生しています。

タイミングの不良／障害

タイミングの不良は、コンポーネントが正しい値で応答しても指定された間隔で応答しない (早すぎるか遅すぎる) 原因となります。この不良に関連する障害は**タイミングの障害**です。正しい値を生成するのに応答に過度の遅延があるオーバーロードしたプロセッサにはタイミングの障害が発生しています。タイミングの障害は、計算を時間的に制約するシステム内でのみ発生します。

任意の不良／障害

前述の障害クラスは、その値または時間ドメインでコンポーネントがどのように失敗するかを指定するものでした。前述の障害クラスが適応されない方法でコンポーネントが両方のドメインで失敗する可能性もあります。このような出力を生成する障害コンポーネントは**任意の障害** (**ビザンチン障**

害)が発生していると言えます。

任意の不良はコンポーネントの指定動作に対するあらゆる違反の原因となります。他の不良タイプはすべて特定タイプの不良動作については除外します。脱落の不良タイプはもっとも限定的となります。したがって、不良分類の分布帯で言えば脱落と任意の不良はこの分布帯の両端となり、他の不良タイプがその中間に位置することになります。このため、後ろの方の障害分類はそれより前に位置する障害分類の特徴を包含することになります。つまり、脱落不良 (障害) は値またはタイミングの不良 (障害) の特殊なケースとして扱うことができます。こうした順序は以下のように階層で表すことができます。

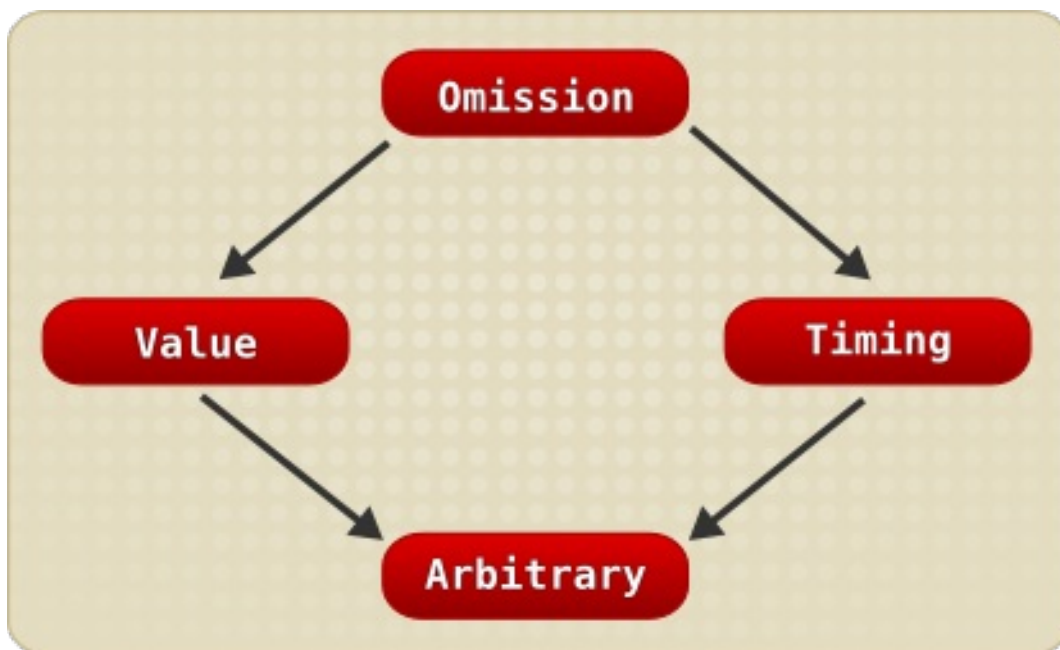


図8.1 不良分類の階層

8.1.1. JBossESB と不良モデル

JBossESB 内では、すべてが任意の障害によって影響を受けます。ご想像の通り、任意の障害を検出するのは本質的に大変困難です。任意の障害に対する耐性をシステムに持たせるプロトコルは存在しますが、何段階もの調整やデジタル署名が必要になる場合がほとんどです。JBossESB の今後のリリースでは、こうした方法の一部がサポートされる見込みです。

値、タイミング、脱落の障害にはアプリケーションに関するセマンティック情報が必要となることが多いため、JBossESB が直接このような障害タイプに対応できることは限られています。しかし、メッセージヘッダー内で `RelatesTo` や `MessageID` などの JBossESB の機能を正しく使用すると、アプリケーションによって、受信したメッセージが受信待ちのメッセージであるか遅延のメッセージであるかを判断することができます。一方向要求に対する非同期の一方向応答など、サービスによる提供が早すぎたメッセージは、基礎のトランスポート実装が原因で損失する可能性があります。たとえば、HTTP などのプロトコルを使用する場合、応答がアプリケーションに渡されるまで応答を保持できる有限のバッファがあります (オペレーティングシステムレベルで設定される)。このバッファの容量を越えると、新しいメッセージを保持するためバッファ内の情報が失われることがあります。この制限によって FTP や SQL などのトランスポートが必ずしも影響を受けるとは限りませんが、同様の動作を生じるような他のリソース制限を受ける可能性があります。

遅延のメッセージに対して耐性を持たせる方が早着のメッセージに対して耐性を持たせるより簡単な場合があります。しかし、アプリケーションの観点では、早着のメッセージが失われると (バッファのオーバーフローなどにより)、無限に遅れるメッセージとの区別がつかなくなります。そのため、メッセージ損失の際に再試行のメカニズムを使用するようアプリケーション (コンシューマーとサービス) を構築する場合、コンシューマーが順序を無視して早い応答を受け取って不適切に処理するという例外

(これにより値の障害が発生します) にてタイミングと脱落の障害に対応できるようにします。メッセージヘッダー内で `RelatesTo` と `MessageID` を使用すると、ペイロード全体を処理しなくても不適切なメッセージシーケンスを見つけることができます (別のオプションを使用することもできます)。

同期された要求と応答の対話パターン内では、応答が一定の時間内に受信されないと RPC で構築されたシステムの多くは自動的に要求を再送信します。しかし、現在の JBossESB は自動的に再送信を行わないため、Couriers または ServiceInvoker 内でタイムアウトのメカニズムを使用し、いつメッセージを再送信するか (または再送信する必要があるか) を判断しなければなりません。高度なトピックの章にある通り、メッセージの配信に影響するようなサービスの障害発生が疑われる場合、そのメッセージを再送信します。



注記

メッセージをサービスに再送信する場合は注意してください。現在、JBossESB には保持される結果やサービス内の再送の検出といった概念はないため、重複するメッセージもサービスへ自動的に配信されます。そのため、サービスが同じメッセージを複数回受け取る可能性があります (損失したのが最初の要求ではなく最初のサービス応答だった場合など)。このように、サービスは同じ作業を実行しようとすることがあります。再送信を使用する場合 (明示的または ServiceInvoker のフェールオーバーメカニズムを使用)、必ず冪等になるようサービス内で複数の要求を処理する必要があります。

トランザクションの使用 (JBossTS で提供されるものなど) やレプリケーションプロトコル (JGroups のようなシステムで提供される) はこうした障害モデルの多くに対して耐障害性を持たせるのに役立ちます。さらに、障害が原因で先に進めない状態の場合、トランザクションを使用するとアプリケーションのロールバックが可能になり、基礎となるトランザクションシステムはまるでその作業がまったく試行されなかったかのように表示してデータの整合性を保証します。現在の JBossESB は JBoss Application Server 内にデプロイされると JBossTS を通じてトランザクション的なサポートを提供します。

8.1.2. 障害検出機能と障害推測機能

理想的な障害検出機能とは、分散システム内でエンティティ (プロセスやマシンなど) の活発さをはっきりと判定できる機能のことです。しかし、障害が発生したシステムと応答が遅いシステムを区別することは不可能なため、一定時間内に障害の検出を保証するのは不可能です。

現在の障害検出機能は、エンティティが使用できるかを判断するためにタイムアウト値を使用します。たとえば、マシンが指定時間内に「are-you-alive?」というメッセージに応答しない場合、障害が発生したと想定します。こうしたタイムアウトに割り当てられる値が正しくない場合 (ネットワークの混雑などが原因で)、正しくない障害が想定され、一部のマシンが別のマシンの障害を「検出」するのに他のマシンは検出しないなど矛盾を招く可能性があります。したがって、ネットワークの混雑やマシンの負荷が最悪な場合など、使用される分散環境内で想定できる最悪の事態に対して、通常こうしたタイムアウトが割り当てられます。しかし、分散システムやアプリケーションが実行の度に予期した通りに動作することはまずありません。したがって、想定する最悪の事態が変化することも考えられる上、障害検出に対して間違った判断をする可能性は常にあります。

障害の検出を保証することは不可能ですが、既知のアクティブなエンティティは相互に通信することができるため、通信できないエンティティは障害が発生していると合意することができます。これが **障害推測機能** の動作になります。あるエンティティが別のエンティティに障害が発生していると予測した場合、残りのエンティティ間でプロトコルが実行され、障害発生の合意を問います。エンティティの障害発生に合意した場合、障害が発生していると見られるエンティティはシステムから除外され、その後このエンティティによる動作は許可されません。1つのエンティティが障害の発生を予測しても、すべてのエンティティが同じような判断をするとは限りません。障害が発生していないエンティティがシステムから除外された場合は、別のプロトコルを実行して活動状態であることを認識させなければなりません。

障害推測機能の長所は、分散環境内で正しく機能しているすべてのエンティティが、障害の発生が疑われる別のエンティティの状態について合意することです。短所は、障害推測のプロトコルは重く、一般的に複数回の合意が必要となることです。また、タイムアウト値に基づいて障害が推測されるため、障害のないエンティティが除外されることもあり、(重大となる可能性のある) リソースの利用や可用性が低下します。

障害検出のメカニズムが時折、誤った答えを返す可能性があるという事実に対する耐性を持つことができるアプリケーションもあります。ただし、これ以外の他のアプリケーションの場合、エンティティが機能しているとの誤った判定はデータ破損などの問題を招くおそれがあり、ミッションクリティカルなアプリケーション (航空機制御システムや原子炉監視など) の場合などは人命に関わる結果となる可能性があります。

現在の JBossESB は障害検出または障害推測をサポートしていません。この短所については今後のリリースで対応していきたいと考えています。現在のところ、前述のような特定のサービスに障害が発生しているのかどうかを判定試行する技術 (MessageID およびタイムアウト/再試行) を使用するコンシューマー側とサービスは利用側で開発して頂く必要があります。アプリケーションに障害の検出から疑わしい障害の処理まで行わせた方がより効率的な場合もあります。

8.2. 信頼性の保証

見てきたように、分散システム内で障害が発生する恐れのある状況は多くあります。このセクションでは障害がどのように JBossESB およびそれにデプロイされたアプリケーションに影響を及ぼすのかについて具体的な例を説明していきます。推奨のセクションでは、こうした障害によりよい耐性を持たせるための JBossESB 設定方法やアプリケーション開発を前提とした設定方法について見ていきます。

JBossESB 内には多くのコンポーネントやサービスがあります。障害発生時に依存するアプリケーションの一部またはすべてに対して認識できない可能性がある障害がいくつかあります。たとえば、コンシューマー側がサービスが機能するために必要な EPR 情報をすべて完全に取得してしまった後にレジストリサービスがクラッシュすると、アプリケーションに不都合な影響はありません。しかし、これより前に障害が発生すると、アプリケーションは先に進めなくなります。したがって、いずれの信頼性保証の判定においても、障害が発生した時期また障害の種類を考慮する必要があります。

信頼性や耐障害性を 100 % 保証することは不可能です。ハードウェアの障害や人的ミスをなくすることはできません。しかし、システムが障害に耐えられる可能性を高くしたり、データの整合性を維持して向上することは可能です。トランザクションやレプリケーションなどの耐障害性技術はパフォーマンスに影響します。アプリケーションの知識に基づいてパフォーマンスと耐障害性の妥協点を見つけるのが最良の方法です。特定の方法をすべてのアプリケーションに対して一様に使用すると、その方法が必要でない状況下ではパフォーマンスの劣化につながります。そのため、JBossESB によってサポートされる耐障害性技術の多くはデフォルトで無効になっています。耐障害性技術は必要に応じて有効にしてください。

8.2.1. メッセージの損失

メッセージの紛失や遅延がどのようにアプリケーションに対して悪影響を与える可能性があるのかについては既に見てきました。また、JBossESB 内でどのようにメッセージがなくなってしまうのかについても例をいくつか見てきました。このセクションではメッセージ紛失についてももう少し詳しく見ていくことにします。

多くの分散システムがポイントツーポイント (1 コンシューマー側と 1 プロバイダー) またはグループベース (複数のコンシューマー側と 1 プロバイダ) で信頼できるメッセージ配信をサポートしています。一般的に信頼性に課されるセマンティックとは、たとえ障害が存在していてもメッセージが配信されるまたはメッセージが受信者に届かなかったことを確実に送信者が知ることができるということになります。信頼できるメッセージング実装を採用しているシステムは受信者にメッセージが配信されるのとそのメッセージが受信者によって処理されるのとは区別する場合が多く、たとえば、単にサービスにメッセージが届くというのと、サービスがメッセージの内容を処理する時間を確保する前にクラッシュが続けて発生した状況とは区別されます。

メッセージの配信や処理で前述の障害セマンティックを提供するトランスポートで、JBossESB 内で使用できるのは JMS のみです。トランザクション処理されるセッションでは (JMSEpr のオプション部分)、障害が存在していてもメッセージの受信や処理を保証することが可能です。サービスによる処理中に障害が発生した場合、メッセージが後で再処理されるよう JMS キューに戻されます。しかし、トランザクション処理されるセッションはトランザクション処理されないセッションより大幅に遅くなることがあるため、使用には注意が必要です。

JBossESB によってサポートされる他のトランスポートは、トランザクションや確実な配信についての保証がないため、メッセージが損失する可能性があります。しかし、ほとんどの場合でメッセージ損失の可能性は低くなります。送信側と受信側の両方で同時に障害が発生しない限り (このような障害が発生することはまずありませんが、起こり得ます)、送信側は JBossESB によってメッセージ配信の障害に関する通知を受けます。処理中に受信側に障害が発生し、応答が预期されていた場合、受信側は最終的にタイムアウトするため再試行できます。



注記

非同期メッセージの配信を使用すると障害の検出／推測が難しくなる可能性があります (理論的には不可)。アプリケーションを開発する際はこの点を考慮にいらしてください。

こうした理由から、高度なトピックの章に説明がある再配信プロトコルとメッセージフェールオーバーが最良のアプローチであると言えます。サービスの障害を予測すると、代替の EPR (1 つが利用可能と仮定) を選択して使用します。しかし、予測した障害が発生していなかった場合は、複数のサービスが同じメッセージで同時に動作する可能性があります。そのため、フェールオーバーにはロバストなアプローチですが、使用する場合は注意が必要です。このアプローチは、同じメッセージを複数回実行することと 1 回実行することが同じ場合など、サービスがステートレスで冪等である場合に最適なアプローチです。

多くのサービスおよびアプリケーションに対してこのタイプの再配信メカニズムは適切に動作します。単一の ERP 全体にわたり提供される堅固さは大きな利点となるでしょう。クライアントとサービスが失敗するまたはサービスに誤って障害が発生したとみなされるなど、動作しない状況の障害モードはかなり珍しいことになります。サービスをべき等にできない場合は、JBossESB がメッセージのトランザクション的配信が維持される結果のなんらかの形式をサポートするまで、JMS を使用するかサービスが再送信を検出して同じ作業を同時に実行している複数のサービスに対処できるコードを使用してください。

8.2.2. エンドポイントの障害を疑う

これまで、障害の検出や予測を判断する難しさについて説明してきました。実際、クラッシュしたマシンと非常に動作の遅いマシンを区別するのは、障害が発生したマシンが回復するまで不可能です。ネットワークを分割して効果的に複数の個別のネットワークとして動作させるため、ネットワークをパーティション化することができます。しかし、ネットワークが分割されると、ネットワークの異なる部分に存在するコンシューマはその部分で利用可能なサービスしか見えなくなります。このような状態は「スプリットブレインシンドローム」とも呼ばれます。

8.2.3. サポート対象となるクラッシュ障害モード

JBossESB は、トランザクションまたは JMS などの、信頼性の高いメッセージ配信プロトコルを使用している場合、システム全体がシャットダウンするという突発故障からも回復できます。

トランザクションや確実なメッセージ配信プロトコルを使用しない場合、関係するエンドポイントの可用性が保証される場合のみ JBossESB は障害に対応できます。

8.2.4. コンポーネント固有

本セクションでは JBossESB 内の特定のコンポーネントおよびサービスについて説明します。

8.2.4.1. ゲートウェイ

ゲートウェイでメッセージが受け取られると、信頼できないトランスポートを使って ESB 内に送信しない限りメッセージが失われることはありません。JMS、FTP、SQL などの JBossESB トランスポートはすべてメッセージを確実に配信するまたはシステムから絶対に削除されないようにするよう設定することができます。残念ながら HTTP はこのようには設定できません。

8.2.4.2. ServiceInvoker

ServiceInvoker は非同期で送信された不達のメッセージを再配信キューに配置します。メッセージの同期配信に失敗すると、送信側へ直ちに通知されます。ServiceInvoker が正しく機能するためには、トランスポートによって配信の失敗を送信側に明確に知らせなければなりません。送信側と受信側で同時に障害が発生すると、メッセージが損失することがあります。

8.2.4.3. JMS ブローカー

JMS ブローカーに配信できないメッセージは再配信キューに置かれます。エンタープライズデプロイメントにはクラスタ化された JMS ブローカーが推奨されます。

8.2.4.4. アクションパイプライン

多くの分散システムと同様、サービスが存在する場所内にあるコンテナが受信するメッセージと、最終的な送り先で処理されるメッセージを区別します。メッセージが正しく配信されても、アクションパイプライン内での処理中にエラーやクラッシュが発生するとメッセージを損失することがあります。前述の通り、受信したメッセージが処理中に削除されないように JBossESB トランスポートの一部を設定し、エラーやクラッシュが発生してもメッセージを損失しないようにすることができます。

8.3. 推奨

上記で説明してきたような障害モデルの概要とそれら障害に対する JBossESB 内の耐性機能を考慮し、次のようなことが推奨されます。

- ステートレスで冪等なサービスを開発するようにしてください。不可能な場合は、アプリケーションが再送信の試行を検出できるように、MessageID を使用してメッセージを特定するようにしてください。メッセージ送信の再試行を行う場合は、同じ MessageID を使用してください。冪等でなく、再送信されたメッセージを受信すると同じ処理を再度行うようなサービスは、なるべくトランザクションを使用して MessageID に対してステート移行を記録するようにします。ステートレスサービスを基にするアプリケーションの方がスケーラビリティが高い傾向にあります。
- ステートフルなサービスを開発する場合はトランザクションと JMS 実装を使用してください(できればクラスタ化する)。
- レジストリのクラスタ化を行い、クラスタ化された耐障害性のあるバックエンドデータベースを使用して単一障害点がないようにします。
- メッセージストアが必ず高可用性のデータベースで支えられているようにします。
- 他のサービスやオペレーションと比較して、より高い信頼性や耐障害性の機能を必要とするサービスおよびサービス上のオペレーションを明確に識別しておきます。これによりこれらのサービスで JMS 以外のトランスポートを対象とすることができるようになり、アプリケーション全体のパフォーマンスを向上させることができます場合があります。JBossESB により

別々の EPR を通じて同時にサービスが使用されるようにすることが可能なため、こうした異なるサービスの特性 (QoS) をアプリケーション固有の必要条件に応じて異なるコンシューマー側に提供することも可能になります。

- ネットワークのパーティションはサービスが失敗しているかのように表示する可能性があるため、クラッシュしたように誤って識別されることに対処できないサービスに対してはこの種の障害が起きる傾向にあるトランスポートの使用は避けてください。
- 場合によっては (HTTP など)、サーバーがメッセージを処理した後、応答する前にサーバーがクラッシュすると、別のサーバーが同じ処理を行う原因となります。これは、サービスが受信したメッセージを処理した後にマシンで障害が発生した場合と、サービスが受信したメッセージを処理する前に障害が発生した場合を区別できないからです。
- 非同期 (一方向) の配信パターンを使用するとサービスの障害検出が困難になります。これは、要求への応答が任意の時間に到達できる場合、一般的に損失メッセージや遅延メッセージの概念がないからです。応答が全くない場合は障害の検出がより難しくなるため、アプリケーションセマンティックに依存してメッセージの不達を判断しなければならない場合もあります (例：銀行口座の残高が予期した金額と異なる)。ServiceInvoker または Couriers を使用して非同期のメッセージを配信する場合、各操作 (deliverAsync など) から返答があってもメッセージがサービスによって動作したとは限りません。
- メッセージストアは再配信プロトコルによって使用されますが、前述の通り、このプロトコルはロバスト性の向上に最善のプロトコルであり、トランザクションや確実なメッセージ配信を使用しません。そのため、障害によってはメッセージが全て損失したり (クラッシュの前にメッセージがストアに書き込まれない)、メッセージが複数回配信される原因となることがあります (再配信のメカニズムがストアからメッセージを取り出して無事配信するが、クラッシュによってメッセージがストアから削除されず、クラッシュから回復した後にメッセージが再度配信される)。
- FTP など一部のトランスポートは、処理済みのメッセージを保持するよう設定することができますが、未処理のメッセージと区別するため独自にマークされます。通常、デフォルトでは処理されたメッセージは削除されますが、障害から回復した際に、アプリケーションが処理されたメッセージを判別できるよう、デフォルトを変更することもできます。

本章の障害に関する説明と反して、障害は頻繁に発生するわけではありません。年月をかけてハードウェアの信頼性は飛躍的に向上され、公式の検証ツールの使用を含め活発なソフトウェア開発が行われることによりソフトウェア関連の問題も少なくなってきました。サービスやアプリケーションを開発、デプロイしていく上で適切な方法を判断するのに役立つよう本章では障害についての説明をしています。パフォーマンスを低減させるような高いレベルでの信頼性や耐障害性が必ずしもすべてに必要なわけではありませんが、間違いなく必要とするものもあるからです。

第9章 サービス設定の定義

9.1. 概要

JBossESB 4.8 の設定は jbossesb-1.2.0 XSD

(<http://anonsvn.jboss.org/repos/labs/labs/jbossesb/trunk/product/etc/schemas/xml/jbossesb-1.2.0.xsd>) をベースとしています。この XSD は常に ESB 設定の最終的な参照となります。

このモデルは次の 2 つの主要セクションから構成されます。

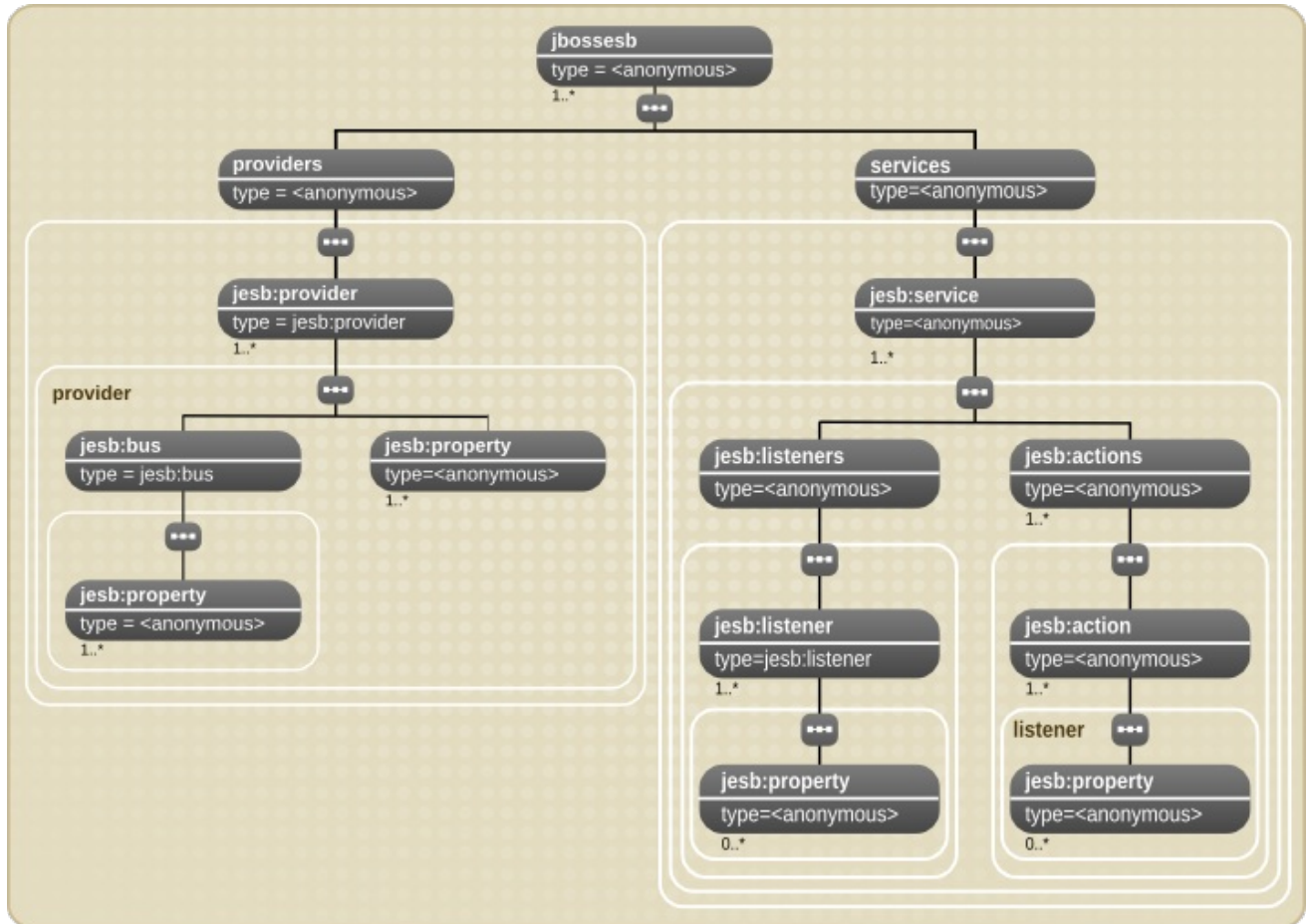


図9.1 JBoss Enterprise Service Bus 設定モデル

1. `<providers>`: モデルのこの部分は主としてモデルの `<services>` セクション内で定義されるメッセージの `<listener>` により使用されるすべてのメッセージ `<bus>` プロバイダーを定義します。
2. `<services>`: モデルのこの部分は主に JBoss ESB の単一インスタンスの制御下にある全サービスを定義します。各 `<service>` インスタンスには "Gateway" または "Message Aware" のいずれかのリスナー定義が含まれます。

このモデルに基づいて構成を行う最も簡単な方法は、**Eclipse** 統合開発環境内の XSD 対応の XML エディターを使用することです。このエディターは構成の編集集中に自動補完の機能を提供します。ファイル上で右クリックして **Open With -> XML Editor** の順で選択します。

9.2. PROVIDERS

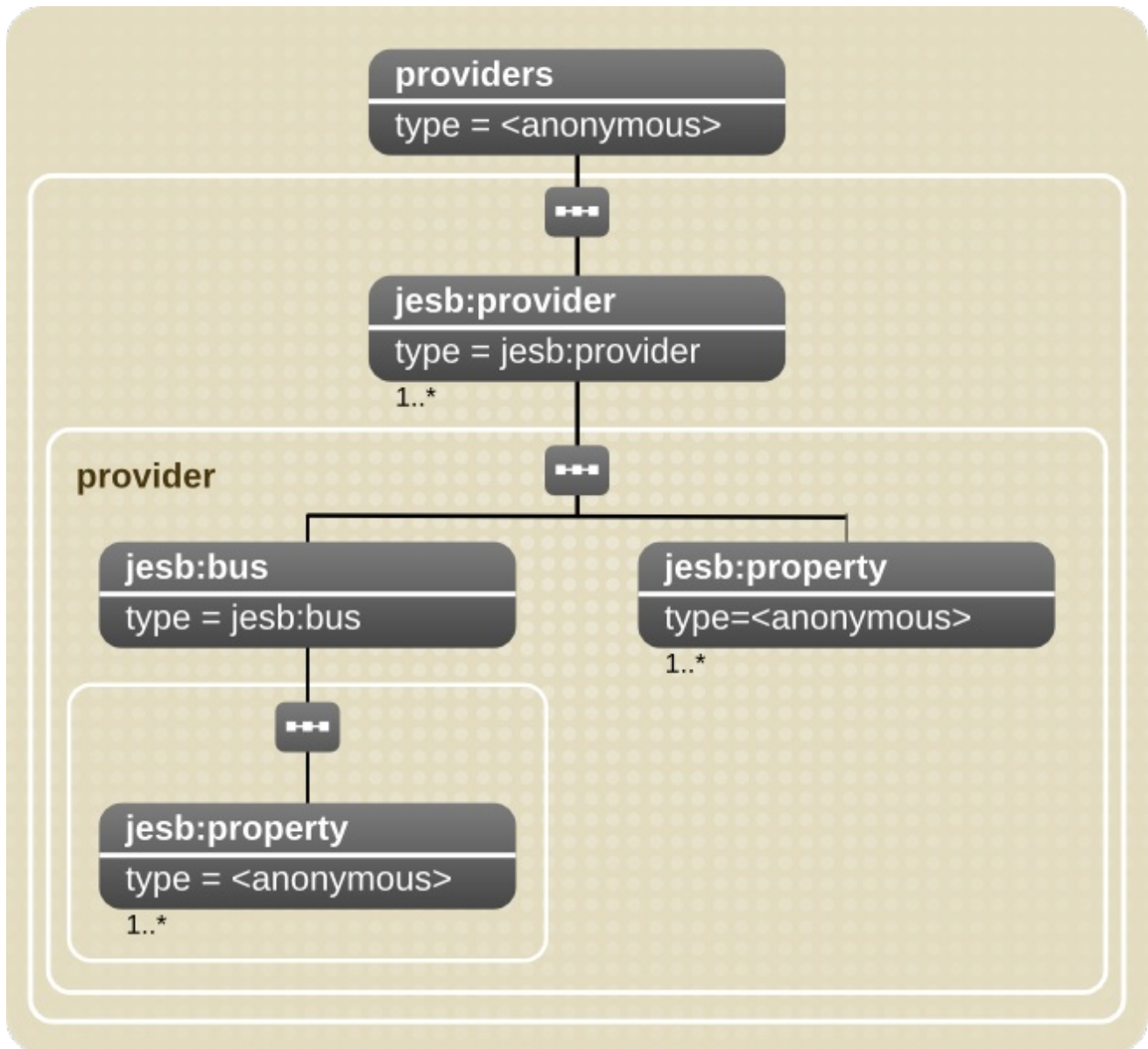


図9.2 プロバイダー構成モデル

設定の **<providers>** 部分では、ESB の単一インスタンスに対するすべてのメッセージ **<provider>** インスタンスを定義します。現在サポートされている 2 つの種類のプロバイダーを以下に示します。

- これらは、「プッシュ」されたメッセージであるリスナーなどに対する「イベント駆動」プロバイダーのプロバイダー詳細を指定します。このプロバイダーの種類の例は **<jms-provider>** です。
- Schedule Provider: メッセージを「プル」するリスナーなどのスケジュール駆動リスナーのプロバイダー設定

バスプロバイダー (**<jms-provider>** など) には複数の **<bus>** 定義を含めることができます。**<provider>** は、その **<property>** で定義される全 **<bus>** で共通となるプロバイダー固有プロパティに関連する **<property>** インスタンスで修飾することもできます (JMS なら "connection-factory"、"jndi-context-factory" など)。同様に、各 **<bus>** インスタンスはその **<bus>** インスタンスに固有となる **<property>** インスタンスで修飾することができます (JMS なら "destination-type"、"destination-name" など)。

例として JMS のプロバイダー構成を以下に示します。


```
<providers>
  <jms-provider name="JBossMQ" connection-
factory="ConnectionFactory">
    <jms-bus busid="Reconciliation">
      <jms-message-filter
        dest-type="QUEUE"
        dest-name="queue/B"
      />
    </jms-bus>
  </jms-provider>
</providers>
```

上記の例は「基本の」 **<provider>** と **<bus>** タイプを使用しています。これに問題はありませんが、実際の設定を作成するにはこれらのタイプの特殊化された拡張を使用することをお勧めします。JMS なら **<jms-provider>** と **<jms-bus>** です。上記の設定でもっとも重要となる部分は **<bus>** インスタンスで定義される **busid** 属性です。これは **<bus>** エlement／タイプで必要とされる属性です (**<jms-bus>** などそのすべての特殊化を含む)。この属性は **<listener>** 構成内で使用され、リスナーがそのメッセージを受け取る **<bus>** インスタンスを参照します。これについては後ほど詳しく説明します。

9.3. サービス

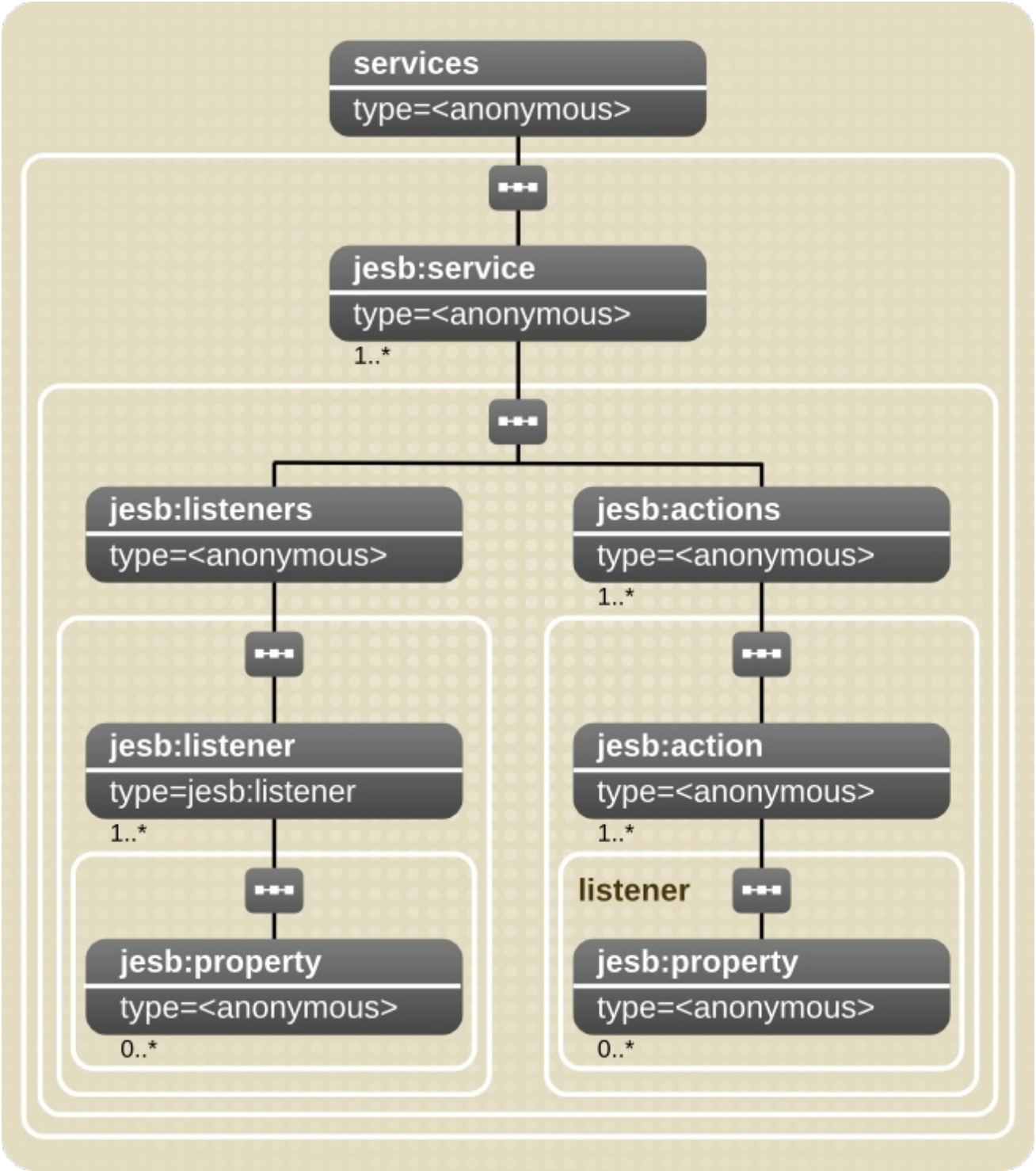


図9.3 サービス構成モデル

構成の `<services>` の部分は ESB のこのインスタンスの管理下にある各サービスを定義します。`<service>` 一連の構成として定義します。`<service>` は次の属性で修飾することもできます。

表9.1 サービス属性

名前	説明	タイプ	必須
name	サービスレジストリ内で登録されるサービスのサービス名	xsd:string	true

名前	説明	タイプ	必須
カテゴリ	サービスレジストリ内で登録されるサービスのサービスカテゴリ	xsd:string	true
詳細	読みやすい形式のサービス詳細、 リジストリ内に格納される	xsd:string	true

<service> は **<listeners>** のセットと **<actions>** のセットを定義することができます。構成モデルは「基本の」 **<listener>** タイプの他、 サポートされるメインのトランスポートそれぞれの特異化も定義します。**<jms-listener>**、**<sql-listener>** など。

「基本の」 **<listener>** は次の属性を定義します。これらの属性定義はすべての **<listener>** 拡張によって継承されます。このように、InVM トランスポートなど JBossESB でサポートするリスナーやゲートウェイすべてに対し設定することができます。

表9.2 リスナーの属性

名前	説明	タイプ	必須
name	リスナーの名前、 この属性は主にログの目的で必要となる	xsd:string	true
busrefid	リスナーインスタンスが受け取るメッセージの元となる <bus> の busid への参照	xsd:string	true
maxThreads	リスナーがアクティブとして持てる同時メッセージ処理スレッドの最大数	xsd:int	True
is-gateway	リスナーインスタンスが「ゲートウェイ」であるかないかは関係ありません。[a]	xsd:boolean	true
[a] メッセージバスは特定のメッセージチャンネル／トランスポートの詳細を定義します。			

リスナーはゼロまたは複数の **<property>** エレメントのセットを定義できます (ちょうど **<provider>** と **<bus>** のエレメント／タイプのようなもの)。これらはリスナー固有のプロパティの定義に使用されます。



注記

サービス内で定義される各ゲートウェイリスナーに対しては、ESB 対応のリスナー (または「ネイティブ」) も定義されなければなりません。ゲートウェイリスナーは双方向のエンドポイントを定義するのではなく ESB への「スタートポイント」を定義するためです。ESB 内からメッセージをゲートウェイに送ることはできません。また、ゲートウェイはエンドポイントではないため、レジストリ内に維持されるエンドポイント参照 (EPR) はありません。

<bus> に対する <listener> 参照の例を次の図で示します (「基本」タイプのみ使用)。

```
<?xml version = "1.0" encoding = "UTF-8"?>
<jbossesb
```



```

xmlns="http://anonsvn.labs.jboss.com/labs/jbossesb/trunk/product/etc/schem
as/xml/jbossesb-1.0.1.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

  xsi:schemaLocation="http://anonsvn.labs.jboss.com/labs/jbossesb/trunk/prod
uct/etc/schemas/xml/jbossesb-1.0.1.xsd
http://anonsvn.jboss.org/repos/labs/labs/jbossesb/trunk/product/etc/schema
s/xml/jbossesb-1.0.1.xsd"
  parameterReloadSecs="5">
    <providers>
      <jms-provider name="JBossMQ" connection-
factory="ConnectionFactory">
        <jms-bus busid="Reconciliation">
          <jms-message-filter
            dest-type="QUEUE"
            dest-name="queue/B"
          />
        </jms-bus>
      <!-- busid --> <jms-bus busid="ReconciliationEsb">
        <jms-message-filter
          dest-type="QUEUE"
          dest-name="queue/C"
        </jms-bus>
      </jms-provider>
    </providers>
    <services>
      <service category="Bank" name="Reconciliation"
        description="Bank Reconciliation Service">
        <listeners>
      <!-- busidref --> <jms-listener name="Bank-Listener"
        busidref="Reconciliation"
        is-gateway="true"/>
        <jms-listener name="Bank-Esb"
          busidref="ReconciliationEsb"/>
        </listeners>
        <actions>
          ....
        </actions>
      </service>
    </services>
  </jbossesb>

```

1 つまたは複数の **<actions>** の一覧がないと、サービスはほとんど何もしません。**<actions>** には一般的にサービスによって受け取られるメッセージのペイロードを処理するロジックが含まれています(そのリスナーを通じて)。代わりに、外部のサービス／エンティティによって消費されるメッセージに対して変換またはルーティングのロジックを含めることもできます。

<action> エlement／タイプは次の属性を定義します。

表9.3 アクションの属性

名前	説明	タイプ	必須
name	アクションの名前、この属性は主にログの目的で必要となる	xsd:string	true

名前	説明	タイプ	必須
クラス	org.jboss.soa.esb.actions.ActionProcessor 実装クラス名	xsd:string	true
プロセス	メッセージ処理に反射的に呼び出される「process」メソッドの名前 (デフォルトは ActionProcessor クラスで定義されるように「process」メソッドとなる)	xsd:int	false

<actions> セット内の **<action>** インスタンスの一覧では、アクションは **<action>** インスタンスが **<actions>** セットに表れる順序で呼び出されます (その「process」メソッドが呼び出される)。各 **<action>** から返されるメッセージは一覧内の次の **<action>** への入力メッセージとして使用されます。

このモデル内のいくつかの他のエレメント／タイプのように、**<action>** タイプもゼロまたは複数の **<property>** エレメントインスタンスを含むことができます。**<property>** エレメント／タイプは標準の名前と値の組み合わせを定義するか、自由形式のコンテンツを含むことができます (xsd:any)。XSD によれば、この自由形式のコンテンツは構成内の場所 (**<provider>**、**<bus>**、**<listener>** のいずれかまたはその派生物のいずれか) に関係なく **<property>** エレメント/タイプの有効な子コンテンツになります。ただし、この自由形式の子コンテンツが使用される **<action>** 定義の **<property>** インスタンス上でのみになります。

上記の **<action>** 定義に記載されているように、アクションは **org.jboss.soa.esb.actions.ActionProcessor class** クラスを実装することで実装されます。このインターフェースの実装はすべて次の形式のパブリックコンストラクターを含まなければなりません。

```
public ActionZ(org.jboss.soa.esb.helpers.ConfigTree configuration);
```

このコンストラクターはアクションの属性を付けて ConfigTree のインスタンスを与えます。アクション **<property>** のインスタンスからの自由形式コンテンツもこれに含まれます。自由形式のコンテンツが ConfigTree インスタンスの子コンテンツとして提供されます。

このため、**<actions>** 構成の例としては次のようになる場合があります。

```
<actions>
  <action name="MyAction-1" class="com.acme.MyAction1"/>
  <action name="MyAction-2" class="com.acme.MyAction2">
    <property name="propA" value="propAVal" />
  </action>
  <action name="MyAction-3" class="com.acme.MyAction3">
    <property name="propB" value="propBVal" />
    <property name="propC">
      <!-- Free form child content... -->
      <some-free-form-element>zzz<some-free-form-element>
    </property>
  </action>
</actions>
```

9.4. 固有タイプの実装

JBoss ESB 構成モデルは「基本」タイプの **<provider>**、**<bus>**、**<listener>** のトランスポート固

有特殊化を定義します (JMS, SQL など)。これにより構成を厳密に検証できるようになり、XSD 対応の XML エディター (Eclipse XML Editor など) を使用するユーザーにとって構成が容易になります。これら特殊化は特に何の設定を行わなくても JBoss ESB でサポートされる各トランスポートの構成要件を明示的に定義します。JBoss ESB の構成を行っている場合は「基本」タイプの代わりにこの特殊化されたタイプを使用することを推奨します。あるいは、公式の JBoss ESB リリース以外で新しいトランスポートがサポートされる場合のみです。

「基本」タイプから構成を行っている場合に適用される同じ基本の原則がトランスポート固有の代替から構成を行う場合にも適用されます。

1. **<jms-provider>** などのプロバイダー設定を定義します。
2. 新しいプロバイダーにバスの構成を追加し (**<jms-bus>**)、固有の busid 属性値を割り当てます。
3. 通常通りに **<services>** を定義して、作成したばかりの新しいバス構成を参照する (busrefid を使用) トランスポート固有のリスナー構成 (**<jms-listener>** など) を追加します (例: **<jms-bus>** を参照する **<jms-listener>**)。

これらトランスポート固有のタイプを使用する場合に適用されるルールは、1 つのタイプのリスナーから別のタイプのバスへのクロス参照はできないということです。つまり、**<jms-bus>** から **<jms-listener>** しか参照できないということです。相互参照されるとランタイムエラーが発生します。

このため、本リリースに配備されるトランスポート固有の実装は、JMS、SQL、FTP、Hibernate、ファイルシステム、スケジュール、JMS/JCA 統合となります。

JMS

<jms-provider>、**<jms-bus>**、**<jms-listener>**、**<jms-message-filter>**

<jms-message-filter> は **<jms-bus>** または **<jms-listener>** エレメントのいずれかに追加できます。**<jms-provider>** と **<jms-bus>** は JMS 接続プロパティを指定し、**<jms-message-filter>** は実際のメッセージのキュー／トピックとセクターの詳細を指定します。

SQL

<sql-provider>、**<sql-bus>**、**<sql-listener>**、**<sql-message-filter>**

<sql-message-filter> は **<sql-bus>** または **<sql-listener>** エレメントのいずれかに追加できます。**<sql-provider>** と **<sql-bus>** は JDBC 接続プロパティを指定し、**<sql-message-filter>** はメッセージ／行の選択と処理プロパティを指定します。

FTP

<ftp-provider>、**<ftp-bus>**、**<ftp-listener>**、**<ftp-message-filter>**

<ftp-message-filter> は **<ftp-bus>** または **<ftp-listener>** のいずれかに追加できます。**<ftp-provider>** や **<ftp-bus>** は FTP アクセスプロパティを指定し、**<ftp-message-filter>** はメッセージ／ファイルの選択と処理プロパティを指定します。

ハイバネート

<hibernate-provider>、**<hibernate-bus>**、**<hibernate-listener>**

<hibernate-message-filter> は **<hibernate-bus>** または **<hibernate-listener>** エレメントのいずれかに追加できます。**<hibernate-provider>** はハイバネート設定プロパティの場所などファイルシステムアクセスのプロパティを指定し、**<hibernate-message-filter>** はリスンするクラス名とイベントを指定します。

ファイルシステム

<fs-provider>、<fs-bus>、<fs-listener>、<fs-message-filter>

<fs-message-filter> は <fs-bus> または <fs-listener> エレメントのいずれかに追加できます。<fs-provider> と <fs-bus> はファイルシステムアクセスプロパティを指定し、<fs-message-filter> はメッセージ／ファイルの選択と処理プロパティを指定します。

スケジュール

<schedule-provider>

これは特殊タイプのプロバイダーで、上記のバスベースのプロバイダーとは異なります。詳細はスケジューリングをご覧ください。

JMS/JCA 統合

<jms-jca-provider>

このプロバイダーは JCA インフローを使って着信メッセージの配信を有効にするため <jms-provider> の代わりに使用できます。これによりトランザクション処理されたフローをアクションパイプラインに導入し、アクションを JTA トランザクション内に包み込みます。

お気づきの通り、現在実装されているトランスポート固有のタイプはすべて「基本」タイプにない追加タイプ、<*-message-filter> を含んでいます。このエレメント/タイプは <*-bus> または <*-listener> いずれかの内側に追加できます。このタイプを両方の場所で指定すると、バス (このバスを使用するすべてのリスナー) にグローバルに、またはリスナーベースで 1 リスナーでローカルにメッセージのフィルタリングを指定することができます。



注記

それぞれのトランスポート固有タイプの属性一覧と詳細を表示するには、各属性に関する全詳細が記載されている jbossesb-1.2.0 XSD (<http://anonsvn.labs.jboss.com/labs/jbossesb/trunk/product/etc/schemas/xml/jbossesb-1.0.1.xsd>) を使用することができます。Eclipse XML Editor など XSD 対応の XML エディターを使用するとこうしたタイプでの作業が非常に容易になります。

表9.4 JMS メッセージフィルタ設定

プロパティ名	説明	必須
dest-type	宛先のタイプ (QUEUE または TOPIC)	Yes
dest-name	Queue または Topic の名前	Yes
selector	複数のリスナーを同じキュー/トピックで登録できるようにします。ただし、リスナーはこのメッセージセレクターに基づいてフィルタリングされます。	No
persistent	JMS の配信モードを永続的にするかどうかを指定します。値は真または偽です。デフォルト値は真です。	No

プロパティ名	説明	必須
acknowledge-mode	JMS セッション確認モード。AUTO_ACKNOWLEDGE、CLIENT_ACKNOWLEDGE、DUPS_OK_ACKNOWLEDGE のいずれかの値になります。デフォルト値は AUTO_ACKNOWLEDGE です。	No
jms-security-principal	JMS 宛先ユーザー名。宛先への接続を作成するときに使用されます。	No
jms-security-credential	JMS 宛先パスワード。宛先への接続を作成するときに使用されます。	No

構成例:

```
<jms-bus busid="quickstartGwChannel">
  <jms-message-filter
    dest-type="QUEUE"
    dest-name="queue/quickstart_jms_secured_Request_gw"
    jms-security-principal="esbuser"
    jms-security-credential="esbpassword"/>
</jms-bus>
```

表9.5 JMS リスナーの構成

プロパティ名	説明	必須
name	リスナーの名前	Yes
busrefid	リッスンする JMS の ID	No
is-gateway	この JMS リスナーインスタンスはゲートウェイですか？それともメッセージ対応のリスナーですか？	No。 デフォルトは false です。
maxThreads	JMS バス上でメッセージをリッスンする際に使用するスレッドの最大数。is-gateway が false の場合のみ該当	No。 デフォルトは 1 です。

プロパティ名	説明	必須
clientId	JMS 接続と関連付けられたクライアント ID。接続とオブジェクトを、プロバイダーによりクライアントの代わりに保たれたステータスを関連付ける際に使用。例：永続的なサブスクリプション	No。 clientId が必須にも拘らず (例：durableSubscriptionName が指定されている場合)、指定されていない場合、デフォルトのリスナー名を使用します。
durableSubscriptionName	永続的なサブスクリプション名	No。 JMS Topics の場合のみ該当

9.5. ファイルシステムプロバイダー設定

以下のファイルシステムプロバイダーの設定オプションは、ファイルシステムメッセージフィルターにあります (fs-message-filter)。これ自体は fs-bus に含まれています。これに関する適切な例は、helloworld_file_action クイックスタートを参照してください。

以下のディレクトリオプションについては、指定された各ディレクトリがなければならず、ファイルの移動や名前の変更を行うにはアプリケーションサーバーのユーザーにはディレクトリ上の書き込み／読み取り権限がなければなりません。

表9.6 ファイルシステムメッセージフィルター設定

プロパティ	説明	必須
directory	受信ファイルに関してモニタリングが行われる FTP ディレクトリ	Yes
input-suffix	受信ファイルのフィルタリングに使用するサフィックス。".esbln" のように 1 文字以上かつ "." が含まれている必要があります。	Yes
work-suffix	ファイルが ESB に処理される場合に使用するサフィックス。デフォルトは ".esblnProcess"。	No
post-suffix	ファイルが ESB により正常に処理された場合に使用するサフィックス。デフォルトは ".esbDone"。	No
post-delete	true の場合、ファイルは処理後に削除される。この場合、post-directory および post-suffix は無効になる。デフォルト値は真。	No
post-directory	ESB による処理後にファイルの移動先となる FTP ディレクトリ。デフォルト値は上記のディレクトリです。	Yes
post-rename	true の場合、ファイルは処理後に名前が変更されます。post-directory および post-suffix は相互排他的である点に注意してください。	No
error-delete	True の場合、処理中にエラーが発生するとこのファイルは削除されます。この場合、error-directory および error-suffix からの影響はありません。デフォルト値は true。	No
error-directory	処理中にエラーが発生した場合にファイルの移動先となる FTP ディレクトリ。デフォルト値は上記のディレクトリ。	Yes
error-suffix	処理中にエラーが発生した場合にファイル名に追加されるサフィックス。デフォルト値は .esbError 。	No

9.6. FTP プロバイダー構成

表9.7 FTP プロバイダー構成

プロパティ	説明	必須
hostname	単純にポート 21 を使用する <host> の <host:port> の組み合わせで問題ありません。	Yes
username	FTP 接続に使用されるユーザー名	Yes
password	上記ユーザーのパスワード	Yes
directory	新しい受信ファイルについてモニタリングされる FTP ディレクトリ	Yes

プロパティ	説明	必須
input-suffix	ESB による消費を目的とするファイルのフィルタに使用されるファイルサフィックス (注: 「.esbIn」のようにドットを付ける)。すべてのファイルが検索されるよう空白文字列にすることも可能。	Yes
work-suffix	別のスレッドやプロセスがファイルを取得しないようファイル処理中に使用されるファイルサフィックス。デフォルト値は .esbInProcess です。	No
post-delete	true の場合、ファイルは処理後に削除される。この場合、post-directory および post-suffix は無効になる。デフォルト値は真。	No
post-directory	ESB による処理後にファイルの移動先となる FTP ディレクトリ。デフォルト値は上記のディレクトリです。	No
post-suffix	処理後にファイル名に追加されるファイルサフィックス。デフォルト値は .esbDone 。	No
error-delete	True の場合、処理中にエラーが発生するとこのファイルは削除されます。この場合、error-directory および error-suffix からの影響はありません。デフォルト値は true。	No
error-directory	処理中にエラーが発生した場合にファイルの移動先となる FTP ディレクトリ。デフォルト値は上記のディレクトリ。	No
error-suffix	処理中にエラーが発生した場合にファイル名に追加されるサフィックス。デフォルト値は .esbError 。	No
protocol	次のいずれかのプロトコル <ul style="list-style-type: none"> • sftp (SSH ファイル転送プロトコル) • ftps (FTP over SSL) • ftp (デフォルト) 	No
passive	FTP接続がパッシブ状態であることを示します。この値を新に設定すると、FTP クライアントが FTP サーバーに対して 2 つの接続を確立します。デフォルト値は false で、クライアントは FTP サーバーが接続するポートを FTP サーバーに指示します。次に FTP サーバーはクライアントに対する接続を確立します。	No
read-only	true の場合、FTP サーバーはファイルに対する書き込み操作を許可しません。この場合、次のプロパティは無効となるので注意: work-suffix、post-delete、post-directory、post-suffix、error-delete、error-directory、および error-suffix。デフォルトは false。詳細については、「読み取り専用 FTP リスナー」のセクションを参照してください。	No

プロパティ	説明	必須
certificate-url	FTPS サーバー検証のためのパブリックサーバー証明書の URL または SFTP クライアント検証のためのプライベート証明書の URL。SFTP 証明書は、デプロイメント成果物内に組み込まれたリソースとして存在します。	No
certificate-name	FTPS サーバー検証のための証明書の一般名	No
certificate-passphrase	SFTP クライアント検証用プライベートキーのパスフレーズ	No

9.7. FTP リスナーの構成

設定されたスケジュール (scheduleidref) を基にリモートのファイルに対してポーリングを行うリスナーをスケジュールします。「サービススケジューリング」を参照してください。

9.7.1. 読み取り専用 FTP リスナー

FTP プロバイダーのプロパティ「read-only」を true に設定するとリモートファイルシステムは書き込み動作を許可しないことをシステムに指示します。パーミッションが特定ファイルに与えられるようなメインフレームコンピュータで FTP サーバーが稼働している場合によく見られるケースです。

読み取り専用実装は JBoss TreeCache を使用して読み取られているファイル名の一覧を保持し、前に読み取られたことがなかったものだけをフェッチします。キャッシュは cacheloader を使用して設定し、安定したストレージに対してそのキャッシュを永続化する必要があります。



注記

ゲートウェイの呼び出しに使用するファイルがどのように作成され、コンテンツが書き込みされるかによっては、コンテンツが完全に書き込まれる前に、リスナーがファイルにアクセスできます。このタイミングの問題を回避するには、一時ファイルをまず作成して、目的のコンテンツで完全に作成されるようにします。一時ファイル内のコンテンツが完了すると、FTP ゲートウェイリスナー向けに設定されたディレクトリ／ファイル名へ移動する必要があります。

キャッシュからファイル名を削除する手段が存在しなければならないので注意してください。定期的に別の場所にファイルを移動するメインフレーム上のアーカイブ機能プロセスなどが該当します。キャッシュからのファイル名削除は、2、3 日置きにキャッシュからファイル名を削除するデータベース手順を設けることなどで行うことができます。キャッシュからファイル名を立ち退かせる場合に cacheloader からそのファイル名を削除する TreeCacheListener を指定するのも別の手段となります。立ち退き期間は設定可能となります。これは FTP リスナー構成でプロパティ (removeFilesystemStrategy-cacheListener) を設定することにより行うことができます。

表9.8 読み取り専用 FTP リスナーの構成

名前	説明
scheduleidref	FTP リスナーによって使用されるスケジュール。「サービススケジューリング」を参照。

名前	説明
remoteFileSystemStrategy-class	実装するクラスでリモートファイルシステムの方法を上書きする: org.jboss.soa.esb.listeners.gateway.remotestrategies.RemoteFileSystemStrategy 。デフォルト値は org.jboss.soa.esb.listeners.gateway.remotestrategies.ReadOnlyRemoteFileSystemStrategy 。
remoteFileSystemStrategy-configFile	ローカルファイルシステムまたはクラスパスに存在する JBoss TreeCache 設定ファイルを指定します。デフォルトではクラスパスのルートに存在する /ftpfile-cache-config.xml という名前のファイルを検索します。
removeFileSystemStrategy-cacheListener	JBoss TreeCacheListener 実装が TreeCache で使用されることを指定する。デフォルト値は TreeCacheListener。
maxNodes	キャッシュに格納されるファイルの最大数。0 は制限無しを意味する。
timeToLiveSeconds	ノードが一掃されるまでのアイドル時間 (秒単位)。0 は制限無しを意味する。
maxAgeSeconds	ノードが一掃されるまでにアイドル時間に関係なくオブジェクトが TreeCache 内に存在する時間 (秒単位)。0 は制限無しを意味する。

構成例:

```
<ftp-listener name="FtpGateway"
  busidref="helloFTPChannel"
  maxThreads="1"
  is-gateway="true"
  schedule-frequency="5">
  <property name="remoteFileSystemStrategy-configFile" value="./ftpfile-
cache-config.xml"/>
  <property name="remoteFileSystemStrategy-cacheListener" value=
"org.jboss.soa.esb.listeners.gateway.remotestrategies.cache.DeleteOnEvictT
reeCacheListener"/>
</ftp-listener>
```

JBoss キャッシュ構成例の一部:

```
<region name="/ftp/cache">
  <attribute name="maxNodes">5000</attribute>
  <attribute name="timeToLiveSeconds">1000</attribute>
  <attribute name="maxAgeSeconds">86400</attribute>
</region>
```

表9.9 構成

プロパティ	説明	コメント
maxNodes	キャッシュに格納されるファイルの最大数。	0 は制限無しを意味します。
timeToLiveSeconds	ノードが一掃されるまでのアイドル時間 (秒単位)。	0 は制限無しを意味します。
maxAgeSeconds	ノードが一掃されるまでにアイドル時間に関係なくオブジェクトが TreeCache 内に存在する時間 (秒単位)。	0 は制限無しを意味します。

helloworld_ftp_action クイックスタートは read-only 構成を示します。クイックスタート実行方法に関する説明を参照するには helloworld_ftp_action クイックスタートディレクトリで「ant help」を実行します (<http://labs.jboss.com/jboss-cache/docs/index.html>)。

9.8. UDP ゲートウェイ

UDP Gateway は、UDP プロトコル経由で送信される ESB 未対応のメッセージを受信するための実装です。ペイロードは、デフォルトの ESB メッセージオブジェクトの場所にあるアクションチェーンへ渡されます。アクションは `esbMessage.getBody().get()` を呼び出し、バイト配列ペイロードをリトリブします。

表9.10 UDP ゲートウェイ設定

プロパティ	説明	コメント
ホスト	リッスンするホスト名/ip	必須
ポート	リッスンするポート	必須
handler Class	org.jboss.soa.esb.listeners.gateway.mina.MessageHandler の具体的な実装	任意。デフォルトは org.jboss.soa.esb.listeners.gateway.mina.DefaultMessageHandler です。
is-gateway	UDPGatewayListener は、ゲートウェイとしてのみ機能できます。	必須

構成例:

```
<udp-listener
  name="udp-listener"
  host="localhost"
  port="9999"

  handlerClass="org.jboss.soa.esb.listeners.gateway.mina.DefaultMessageHandl
```



```
er"
    is-gateway="true"
<udp-listener/>
```

9.9. JBOSS リモートイング (JBR) 設定

JBoss Remoting Gateway は JBoss Remoting (<http://www.jboss.org/jbossremoting/>) をトランスポートオプションとして JBoss ESB にリンクします。JBR 経由 HTTP (S) および Socket (+SSL) に対するサポートを活用します。

JBR プロバイダーの基本設定は以下の通りです。

```
<jbr-provider name="socket_provider" protocol="socket" host="localhost">
    <jbr-bus busid="socket_bus" port="64111"/>
</jbr-provider>
```

そのため、基本的な <jbr-provider> や <jbr-bus> 設定は、非常にシンプルです。<jbr-listener> 経由で <service> 設定から、<jbr-bus> を参照可能です。

```
<listeners>
    <jbr-listener name="soc" busidref="socket_bus" is-gateway="true"/>
</listeners>
```

<jbr-listener> はゲートウェイとしてのみサポートされます。is-gateway を false に設定すると Service デプロイメントエラーが発生します。

JBR Gateway は、<jbr-provider>、<jbr-bus> または <jbr-listener> 要素 (<property> 要素として) のいずれかに設定可能な設定プロパティを多数サポートします。以下にまとめています。

表9.11 構成

名前	説明	デフォルト値
synchronous	対象サービスを同期的に呼び出します。	True
serviceInvokerTimeout	非同期呼び出しのタイムアウト	20000
asyncResponse	非同期のレスポンス	"<ack/>"
securityNS	これは、使用すべき Web Service Security バージョンの名前空間です。この名前空間を使用して、SOAP メッセージのセキュリティヘッダーをマッチします。これは、Enterprise Service Bus がこれらのヘッダーからセキュリティ情報を抽出するためのものです。	http://docs.oasis-open.org/wss/2004/01/oasis-200401http-wss-wssecurity-secext-1.0.xsd

JBR Gateway は JBR 固有の設定プロパティをサポートする点に注意してください。これは、プロパティ名に jbr- のプレフィックスを追加することで実行できます。設定済みのプロトコルに関する JBR 固有の設定については、JBoss Remoting の文書を参照してください。以下の設定例では、HTTPS のキーストアとクライアント認証モードを構成するための JBR 固有設定を指定しています。


```
<jbr-provider name="https_provider" protocol="https" host="localhost">
  <!-- Https/SSL settings -->
  <property name="jbr-KeyStoreURL" value="/keys/myKeystore" />
  <property name="jbr-KeyStorePassword" value="keys_ssl_pass" />
  <property name="jbr-TrustStoreURL" value="/keys/myKeystore" />
  <property name="jbr-TrustStorePassword" value="keys_ssl_pass" />
  <property name="jbr-ClientAuthMode" value="need" />
  <property name="serviceInvokerTimeout" value="20000" />

  <jbr-bus busid="https_bus" port="9433"/>
</jbr-provider>
```

JBR Gateway では、レスポンスヘッダーはすべて、**org.jboss.soa.esb.message.ResponseHeader** クラスのインスタンスとして、**Message.Properties** 内にある必要があります。そのため、JBR Gateway をレスポンスヘッダーに設定する必要がある場合、Gateway レスポンスへ提供された Enterprise Service Bus Message (例：対象サービスの同期呼び出しの後) には、**Message.Properties** 上に設定された **ResponseHeader** クラスのインスタンスを含める必要があります。

9.10. HTTP ゲートウェイ

HTTP ゲートウェイでは、Enterprise Service Bus 上でメッセージ未対応の HyperText Transfer Protocol エンドポイントを公開できます。

Gateway は JBoss Enterprise Service Bus Application Server HTTP Container を使用して、HTTP エンドポイントを公開するため、設定の多くはコンテナレベルで管理されます。バインド／ポートアドレス、SSL などです。

9.10.1. 基本設定

以下のやり方でサービス上に `<http-gateway>` を設定するのが簡単です (プロバイダー設定は必要なし)。

```
<service category="Vehicles" name="Cars" description=""
  invmScope="GLOBAL">
  <listeners>
    <http-gateway name="Http" />
  </listeners>
  <actions mep="RequestResponse">
    <!-- Service Actions.... -->
  </actions>
</service>
```

上記の設定は、**default** HTTP バスプロバイダーを使用します。これは、`busrefid` 属性を定義しないためです。サービス名を使用して、HTTP エンドポイントアドレスを以下の方法で構築します。

`http://<host>:<port>/<.esbname>/http/Vehicles/Cars`

".esb" 拡張子のない、.esb デプロイメントの名前の `<.esbname>` トークン。アドレスの "http" トークンも注意してください。これは、`<http-gateway>` エンドポイント全てに使用されるハードコードされた名前空間のプレフィックスです。

9.10.2. URL パターン

<http-gateway> は、以下のように urlPattern もサポートします。

```
<service category="Vehicles" name="Cars" description=""
  invmScope="GLOBAL">
  <listeners>
    <http-gateway name="Http" urlPattern="esb-cars/*" />
  </listeners>
  <actions mep="RequestResponse">
    <!-- Service Aactions.... -->
  </actions>
</service>
```

これは、サービスに対して HTTP エンドポイントを公開し、以下のアドレスですべての HTTP リクエストをキャプチャーします。

http://<host>:<port>/<.esbname>/http/esb-cars/*

9.10.3. ポート制限

プロパティ allowedPorts は、特定のサービスを 1 つまたは複数グループになった HTTP ポートに限定するために使用することができます。これは、コンマ区切りのポートを一覧に並べることで指定します。

```
<http-gateway name="Http" urlPattern="esb-cars/*">
  <property name="allowedPorts" value="8080,8081">
</http-gateway>
```

これは、サービスに対して HTTP エンドポイントを公開し、以下のポートのみですべての HTTP リクエストをキャプチャーし、それ以外のポートのリクエストは HTTP ステータスコード 404 - Not Found を受け取ります。

- http://<host>:8080/*
- http://<host>:8081/*

9.10.4. 処理リクエスト

<http-gateway> は通常、リクエスト MIME タイプをベースに HTTP Request ペイロードをデコーディングすることができます。これは、jbossesb-properties.xml ファイルから "core.org.jboss.soa.esb.mime.text.types" 設定プロパティを使用し、ペイロードを文字列として (サービス向けに) デコーディングするか、またはアクションを使いデコーディングを処理するサービスを使いバイト配列としてそのまま残すか決定します。

"core.org.jboss.soa.esb.mime.text.types" 設定プロパティはセミコロン区切りの "text" (文字) MIME タイプで、デフォルトは以下のとおりです (ワイルドカードサポートに注意)。

- text/*
- application/xml
- application/*-xml

<http-gateway> は、テキストペイロードをデコーディングする際にリクエストからエンコードされた文字を使用します。

<http-gateway> もペイロード属性をサポートし、上記のデフォルトの MIME タイプをベースとする動作をオーバーライドして使用することができます。この属性を使い、ゲートウェイにペイロードを "bytes" または "string" として処理するよう明示的に指示することができます。

9.10.5. リクエスト情報

HyperText Transfer Protocol Request には、サービスが必要とする情報が多数含まれています (つまり、**POST** の場合、リクエストペイロードのみ)

```
HttpRequest requestInfo = HttpRequest.getRequest(message);
```

HttpRequest は以下のプロパティを公開します (getter メソッドを使用)

表9.12 プロパティ

プロパティ	説明
queryParams	クエリパラメーターを含む <code>java.util.Map<String, String[]></code> 。複数の値を持つパラメーターをサポートするため、値は <code>String[]</code> となります。
ヘッダー	リクエストヘッダーを含む <code>java.util.List<HttpHeader></code>
authType	エンドポイントを保護するのに使用する認証スキーマの名前。または認証がない場合は <code>null</code> 。CGI 変数の <code>AUTH_TYPE</code> の値と同じ。
characterEncoding	このリクエストのボディで使用する文字エンコーディングの名前、またはリクエストが文字エンコーディングを指定しない場合 <code>null</code> 。
contentType	リクエストのボディのコンテンツタイプ (MIME タイプ)、またはタイプが不明な場合は <code>null</code> 。CGI 変数の <code>CONTENT_TYPE</code> の値と同じ。
contextPath	リクエストのコンテキストを示すリクエスト URI の一部。リクエスト URI では、コンテキストパスは常に最初にきます。このパスは "/" で始まりますが、 "/" 文字で終わりません。デフォルト (root) コンテキストのエンドポイントでは、 "" を返します。コンテナーは、この文字列をデコーディングしません (Servlet 仕様を参照)。
pathInfo	このリクエストが出された時にクライアントが送信する URL に関連付けられた追加のパス情報。この追加のパス情報は、 "/" 文字から始まり、エンドポイントパスの後、かつクエリ文字列の前に来ます。追加のパス情報がなかった場合、このメソッドは <code>null</code> を返します。CGI 変数の <code>PATH_INFO</code> の値と同じです (Servlet 仕様を参照)。
pathInfoToken	pathInfo のトークンを含む <code>List<String></code>
queryString	クエリ文字列
requestURI	プロトコル名からクエリ文字列までのこのリクエスト URL の一部。Web コンテナーはこの文字列をデコーディングしません。

プロパティ	説明
requestPath	エンドポイントを呼び出すリクエスト URL の一部。追加のパス情報やクエリ文字列は含みません。CGI 変数の SCRIPT_NAME の値と同じ。urlPattern が "/" に出会った場合、このメソッドは "http" を返します。
localAddr	リクエストを受け取ったインターフェースの IP アドレス
localName	リクエストを受け取った IP インターフェースのホスト名
method	HTTP メソッド
protocol	HTTP プロトコルの名前とバージョン
remoteAddr	クライアントまたはリクエストを最後に送信したプロキシの IP アドレス。CGI 変数の REMOTE_ADDR の値と同じ
remoteHost	クライアントまたはリクエストを最後に送信したプロキシの完全修飾名。エンジンが (パフォーマンスの改善のため) ホスト名を解決できない、あるいは解決しようとしなかった場合、ドット表記の IP アドレスとなります。
remoteUser	ユーザー認証がされている場合、このリクエストを出すユーザーのログイン。または、ユーザー認証がされていない場合は null。ユーザー名が後続のリクエストに送信されるかは、クライアントや認証タイプにより変わります。CGI 変数の REMOTE_USER の値と同じ。
contentLength	リクエストボディの長さ (バイト) および入力ストリームで利用可能な長さ (バイト)、長さが不明な場合は -1。HTTP サブレットの場合、CGI 変数の CONTENT_LENGTH の値と同じ。
requestSessionId	クライアント指定のセッション ID、または指定がない場合は null
scheme	使用するスキーマ。"http" または "https" いずれか
serverName	リクエストの送信先サーバーのホスト名。"Host" ヘッダーの値にある ":" の前の値です。解決済みの名前またはサーバーの IP アドレス。

9.10.6. レスポンス処理

9.10.6.1. 非同期レスポンス処理

このゲートウェイは常に、同期 HTTP クライアントに同期レスポンスを返すため非同期にはなりません。デフォルトでは、このゲートウェイは同期的にサービスパイプラインを呼び出し、HTTP レスポンスをゲートウェイから同期サービスレスポンスとして返します。

このゲートウェイからすると、非同期レスポンスの動作は、ゲートウェイがアクションパイプラインの非同期呼び出し (同期サービス呼び出しではない) の後に、同期 HTTP レスポンスを返しているだけのことです。サービスを非同期的に呼び出すため、同期 HTTP レスポンスの一部としてサービスレスポンスを返すことができません。そのため、ゲートウェイを設定して、非同期レスポンスを出す方法を指示する必要があります。

非同期動作は、<asyncHttpResponse> 要素を <http-gateway> に以下のように追加することで設定します。

```
<listeners>
  <http-gateway name="Http" urlPattern="esb-cars/*">
    <asyncResponse />
  </http-gateway>
</listeners>
```

上記のように設定した場合、ゲートウェイは、HTTP ステータス 200 (OK) と、長さ 0 の HTTP レスポンスペイロードを返します。

<asyncHttpResponse> 要素に "statusCode" 属性を設定するだけで、非同期レスポンス HTTP ステータスコードを設定できます。

```
<listeners>
  <http-gateway name="Http" urlPattern="esb-cars/*">
    <asyncResponse statusCode="202" />
  </http-gateway>
</listeners>
```

上記の長さゼロのペイロードが非同期レスポンスに返されます (デフォルト)。これは、<asyncHttpResponse> 要素に <payload> 要素を指定することでオーバーライド可能です。

```
<listeners>
  <http-gateway name="Http" urlPattern="esb-cars/*">
    <asyncResponse statusCode="202">
      <payload classpathResource="/202-static-response.xml"
        content-type="text/xml"
        characterEncoding="UTF-8" />
    </asyncResponse>
  </http-gateway>
</listeners>
```

表9.13

プロパティ	説明	必須
classpathResource	レスポンスペイロードを含むクラスパスでファイルへのパスを指定します。	必須
contentType	classpathResource 属性が指定したペイロードデータのコンテンツ / MIME タイプを指定します。	必須
characterEncoding	classpathResource 属性が指定したデータの文字エンコーディング	任意

9.10.6.2. 同期レスポンス処理

デフォルトは、このゲートウェイが関連サービスを同期的に呼び出し、サービスレスポンスペイロードを HTTP レスポンスとして返します。

9.10.6.3. レスポンス情報

ゲートウェイが関連のサービスに対して `HttpRequest` オブジェクトインスタンスを作成する方法とあわせ、同期 HTTP ゲートウェイ呼び出しでゲートウェイに対して、関連のサービスは `HttpResponse` オブジェクトを作成することができます。

サービス (アクション) は、以下のようにレスポンスメッセージ上で `HttpResponse` インスタンスを作成、設定することができます。

```
HttpResponse responseInfo = new HttpResponse(HttpServletResponse.SC_OK);

responseInfo.setContentType("text/xml");
// Set other response info ...

// Set the HttpResponse instance on the ESB response Message instance
responseInfo.setResponse(responseMessage);
```

`HttpResponse` オブジェクトは、送出 HTTP ゲートウェイレスポンスヘマッピングされる以下のプロパティを含むことができます。

表9.14

プロパティ	説明
<code>responseCode</code>	ゲートウェイレスポンスに設定される HTTP レスポンス／ステータスコード (http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html)
<code>contentType</code>	レスポンスペイロード MIME タイプ
<code>encoding</code>	レスポンスペイロードのコンテンツエンコーディング
<code>length</code>	レスポンスペイロードのコンテンツの長さ
<code>headers</code>	リクエストヘッダーを含む <code>java.util.List<HttpHeader></code>

このクラスは、`HttpRouter` などの内部のアクションにより使用されるため、`HttpResponse` クラスを使用すると正常に機能し、このゲートウェイを使用してプロキシ操作が簡単に行うことができます。

9.10.6.4. ペイロードエンコーディング

レスポンスペイロードのコンテンツエンコーディングは `HttpResponse` インスタンス経由で設定可能です (上記参照)。

9.10.6.5. レスポンスのステータス

HTTP レスポンスステータスコードは (<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>) `HttpResponse` インスタンス経由で設定されます (上記参照)。

9.10.6.6. レスポンスのタイムアウト

デフォルトでは、このゲートウェイは同期サービスの呼び出しが完了するまで 30,000 ミリ秒 (30 秒) 待機してから、`ResponseTimeoutException` を送出します。デフォルトのタイムアウトをオーバーライドするには、"`synchronousTimeout`" プロパティを設定する必要があります。


```
<listeners>
  <http-gateway name="Http" urlPattern="esb-cars/*">
    <property name="synchronousTimeout" value="120000"/>
  </http-gateway>
</listeners>
```

9.10.6.7. 例外処理

サービス例外 (アクションパイプラインの例外) が ESB 設定経由で個別の HTTP レスポンスコードへマッピングすることができます。

マッピングは、トップレベルの `<http-provider>` で指定することも、`<http-gateway>` に直接指定することも可能で、`<http-provider>` にグローバル定義した例外マッピングをリスナーごとにオーバーライドできます。以下は、`<http-gateway>` 設定に直接行った例外マッピングの例です。

```
<http-gateway name="http-gateway">
  <exception>
    <mapping class="com.acme.AcmeException" status="503" />
  </exception>
</http-gateway>
```

`<http-provider>` レベルでの例外マッピングの設定は、全く同じです。

"{exception-class}={http-status-code}" マッピングを含む、単純な `.properties` 形式ファイルであるマッピングファイルを設定することもできます。ファイルはクラスパスで検索され、`.esb` デプロイメント内でバンドルされる必要があります。以下のように設定されます (今回は `<http-provider>` 上で)。

```
<http-provider name="http">

  <!-- Global exception mappings file... -->
  <exception mappingsFile="/http-exception-mappings.properties" />
</http-provider>
```

9.10.7. セキュリティ処理

セキュリティ制約の設定には、ESB 設定の `<http-provider>` 設定セクションへ各種設定を追加します (つまり、`<http-gateway>` 設定で直接行うことはできません)。

`<http-gateway>` のセキュリティ確保のプロセスは以下のとおりです。

1. 設定の `<http-provider>` セクションで `<http-bus>` を指定します。
2. `<http-bus>` で制約を指定します。
3. "busrefid" 属性を使用して `<http-gateway>` から `<http-bus>` を参照します。



注記

詳細にわたるサンプルについては、"http-gateway" Quickstart を参照してください。

9.10.8. 保護メソッドおよび許容ユーザーのロール

以下のようにログインは、<http-bus> 設定の <protected-methods> や <allowed-roles> セクションを使用してエンドポイントを有効にすることができます。

```
<http-bus busid="secureSalesDeletes">
  <allowed-roles>
    <role name="friend" />
  </allowed-roles>
  <protected-methods>
    <method name="DELETE" />
  </protected-methods>
</http-bus>
```

上記の設定は、有効な "friend" ログインは、"secureSalesDeletes" バスで出された DELETE リクエストに必要であることが記述されています。以下のログインの表は、どの設定がいつ、ログインを強制するかを例をからめて説明しています。

表9.15

指定のメソッド	指定のロール	ログイン必須
No	No	No
No	Yes	全メソッド向け
Yes	Yes	指定済みのメソッドのみ
Yes	No	No。指定のメソッドはすべてブロック

9.10.9. 認証メソッドとセキュリティドメイン

<globals> 要素内の <war-security> 設定で、認証メソッドとセキュリティドメインを設定することができます。

```
<http-provider name="http">
  <http-bus busid="secureFriends">
    <allowed-roles>
      <role name="friend" />
    </allowed-roles>
    <protected-methods>
      <method name="DELETE" />
    </protected-methods>
  </http-bus>

  <auth method="BASIC" domain="java:/jaas/JBossWS" />
</http-provider>
```

メソッド属性は、"BASIC" (default)、"CLIENT-CERT"、"DIGEST" のいずれかになります。



注記

アプリケーションのセキュリティドメインの設定に関する詳細については、JBoss Application Server 文書を参照してください。

9.10.10. トランスポート保証

HTTP Transport Guarantee は、"transportGuarantee" 属性を使用してバス上に指定するだけで http-bus ごとに設定できます。

```
<http-bus busid="secureFriends" transportGuarantee="CONFIDENTIAL">
  <!-- etc etc -->
</http-bus>
```

transportGuarantee の許容値は、"CONFIDENTIAL"、"INTEGRAL"、"NONE" です。

9.11. CAMEL ゲートウェイ



警告

Camel Gateway はテクノロジーレビューのみを目的で提供しています。テクノロジーレビュー機能は完全対応されておらず、機能も完全ではありません。そのため、実稼働環境での利用を目的に設計されているわけではありません。これらの機能は、お客様が今後の製品イノベーションに早い段階でアクセスでき、開発プロセスで機能のテストやフィードバックの提供を行うことができるように提供されています。

Red Hat の JBoss サポートは、これらの機能のご利用時にお客様が直面され報告いただいた問題に対し、商習慣に基づく妥当な努力を払い、解決に努めます。

名前のとおり、Camel Gateway を使うと、メッセージ未対応の *Camel* エンドポイントを公開することができます。Apache Camel の入力機能を使い、Camel メッセージを ESB メッセージに変換して、関連の ESB サービスを呼び出します。

JBoss Enterprise Service Bus で提供される他のゲートウェイと Camel Gateway との最も大きな違いは、Camel Gateway はトランスポートの 1 つのタイプに縛られないということです。Camel が対応する様々なトランスポートをすべて活用するよう設計されているため、しっかり統合されるようになります。Camel が処理する各種トランスポートを参照するには、Camel Component 一覧 (<http://camel.apache.org/components.html>) を参照してください。



注記

Camel コンポーネントごとに、ライブラリ依存も変わってきます。JBoss Enterprise Service Bus には、**camel-core.jar** のみが含まれており、他に必要な依存性は手動で追加する必要があります (その他の camel-* JAR やサードパーティの JAR を含む)。

jboss-esb.xml ファイルで、更新スキーマ (jbossesb-1.3.0.xsd) を使用するように宣言した場合、JBoss ESB の <providers> セクション内で、新しい <camel-provider> セクションが利用できます。


```
<camel-provider name="...">
  <camel-bus busid="...">
    <from uri="..."/>
    <from uri="..."/>
  </camel-bus>
</camel-provider>
```

最も興味深い点は、<camel-bus> 要素内に含まれている部分です。<from uri=""/> 要素のバインドされていない数字をここに追加できます。Camel XML 設定をご存じの方は、ネーティブの Camel XML 設定での作業と全く同じであるため、この要素も使いやすいはずです。また、新しい <camel-gateway> 要素もあり、busidref 属性経由で前述したバスを参照することができます。

```
<camel-gateway name="..." busidref="..."/>
```

<camel-provider> を全く使わずに、<camel-gateway> 要素内で <from uri=""/> 要素を定義することができます。

```
<camel-gateway name="...">
  <from uri=""/>
  <from uri=""/>
</camel-gateway>
```

また、簡略表記の仕組みもあり、ゲートウェイレベルやバスレベルのいずれかで XML 属性として Camel "from" URI を 1 つ指定できます。

```
<camel-gateway name="..." from-uri="..."/>
```

```
<camel-bus name="..." busid="..." from-uri="..."/>
```



注記

<camel-bus> と <camel-gateway> レベルで定義されている Camel "from" URI は、要素の形式または簡略形式のいずれを使用しても、すべて累積される点が重要です。

この時点では、<to uri=""/> 要素はどこにあるのかと感じられるかもしれません。Camel では、あて先を 1 つ以上定義する必要があるためです。JBoss Enterprise Service Bus では、Camel "from" URI はすべて、1 つのルート (ゲートウェイとバスの他のルートに追加) と暗示的な Camel "to" URI (<camel-gateway> を割り当てる関連サービス呼び出し) に変換されます。ゲートウェイとバスにあるルートはすべて、最終的にこのサービス呼び出し、同じ CamelContext 内で実行されます。このライフサイクルは CamelGateway のライフサイクルにリンクされています。



注記

このゲートウェイは、単一の "from" URI に定義可能な Camel のルートにのみ対応している点に留意してください。このゲートウェイが対応している基本的な構文は、from(endpointUri).to(esbService) です。ゲートウェイは、他の Camel コンポーネントへの仲介ルーティングを必要とするルートには対応していません。

Camel コンポーネントによっては、スケジュールされたルーティングタスクを実行します。例：HTTP コンポーネントを使用して定期的に HTTP アドレスやファイルにポーリングしたり、ファイルコンポーネントがファイルシステムディレクトリにポーリングしたりできます。これらのコンポーネントすべてが、ポーリングの頻度を設定する URI オプションに対応しているわけではありません。HTTP コンポーネント

はその一例です。この場合、コンポーネント URI Scheme に "esbschedule:<frequency-in-millis>" を接頭辞として追加するだけです。例：<from uri="esbschedule:5000:http://www.jboss.org" /> は 5 秒ごとに jboss.org をポーリングします。

最後に、<camel-gateway> か、<camel-bus> のいずれかに設定することのできるオプションの属性が 2 種類あります (この場合バスをオーバーライドするゲートウェイ)。async と timeout です。

```
<camel-gateway name="..." from-uri="..." async="false" timeout="30000"/>
```

- async 属性 (デフォルトは false) は、基盤の ServiceInvoker が関連のサービスを同期的または非同期的に呼び出すかを指定します。
- timeout 属性 (デフォルトは 30000) は、ServiceInvoker が同期呼び出しを中断するまでに待機する時間をミリ秒で定義します。



注記

詳細情報は、<http://community.jboss.org/wiki/CamelGateway> の wiki ページを参照してください。

ディストリビューションの .../samples/quickstarts/camel_helloworld/ にクイックスタートもあります。

Apache Camel の詳細は、<http://camel.apache.org/> の Web ページを参照してください。

9.12. 旧構成モデルからの移行

本セクションは XSD ベースではない 旧 JBoss ESB 構成モデルについて熟知している開発者を対象としています。

旧構成モデルは ESB コンポーネント付きの自由形式の XML 構成を使用し、その設定を **org.jboss.soa.esb.helpers.ConfigTree** のインスタンス経由で受け取っていました。新しい構成モデルは XSD ベースとなりますが、基礎となるコンポーネント構成パターンはいまだ **org.jboss.soa.esb.helpers.ConfigTree** のインスタンス経由になります。つまり現在、XSD ベースの構成は ConfigTree スタイルの設定にマッピング/変換されるということになります。

旧モデルの使用に慣れている開発者の方々は次の点に注意して頂く必要があります。

1. 新しい構成モデルに関する全ドキュメントをお読みください。旧モデルの知識に基づいて新しい構成を推測で理解することはできません。
2. 新しい構成で自由形式のマークアップがサポートされる場所は <property> エレメント/タイプのみになります。このタイプは <provider>、<bus>、<listener> のタイプ (およびサブタイプ) で許可されます。ただし、<property> ベースの自由形式マークアップが ConfigTree 設定にマッピングされる場所は <property> が <action> で存在する場所のみになります。この場合、<property> のコンテンツはターゲットの ConfigTree <action> にマッピングされます。しかし、<action> で自由形式の子コンテンツを持つ <property> エレメントが複数ある場合、このコンテンツはすべてターゲット ConfigTree <action> 上で連結されます。
3. 新しいリスナー/アクションのコンポーネントを開発する際はこれらのコンポーネントが依存する ConfigTree ベースの構成が新しい XSD ベースの設定からマッピング可能であることを必ず確認してください。この例としては、ConfigTree 構成モデル内でどのようにその設定をリスナーノード上で属性を使いリスナーコンポーネントに提供することが決定できるか、またはリスナー構成内の子ノードに基づいて提供することが決定できるかを示す例です。<listener> コンポーネント上にある自由形式構成のこのタイプは XSD から ConfigTree へのマッピングではサ

ポートされません。つまり、上記の例の子コンテンツは XSD 設定から ConfigTree スタイルの設定にはマッピングされないということです。実際、XSD 設定は `<property>` に存在しない限り任意のコンテンツを受け取りません。また、`<property>` に存在する場合でも (`<listener>` 上)、マッピングコードによって無視されます。

9.13. 構成

コア内のコンポーネントはすべてその設定パラメータを XML として受け取ります。これらのパラメータがどのようにシステムに提供されるのかは

org.jboss.soa.esb.parameters.ParamRepositoryFactory によって隠されます。

```
public abstract class ParamRepositoryFactory
{
    public static ParamRepository getInstance();
}
```

これは異なる実装を許可する **org.jboss.soa.esb.parameters.ParamRepository** インターフェースの実装を返します。

```
public interface ParamRepository
{
    public void add(String name, String value) throws
        ParamRepositoryException;
    public String get(String name) throws ParamRepositoryException;
    public void remove(String name) throws ParamRepositoryException;
}
```

この JBossESB バージョン内にあるのは

org.jboss.soa.esb.parameters.ParamFileRepository の実装 1 つのみあります。1 ファイルからのパラメータのロードが可能であることを期待します。使用する実装は `org.jboss.soa.esb.paramsRepository.class` プロパティを使って上書きが可能です。



注記

ESB 設定ファイルは **Eclipse** または何らかの XML エディターを使って作成することをお勧めします。JBossESB 設定情報はアノテーション付き XSD でサポートされ、基本的なエディターを使うと作成が容易になるはずです。

第10章 データのデコード - MIME デコーダー

すべての JBossESB ゲートウェイは、「メッセージコンポーザー」のコンセプトをサポートしています。このメッセージコンポーザーとは、MessageComposer を実装し、ESB メッセージインスタンスの構築と関連の ESB サービスへの送信を担当するクラスです。

MIME デコーダーは、MimeDecoder インターフェースを実装するクラスです。MessageComposer 実装が MIME デコーダーを使用して、バイナリエンコードデータの MIME タイプをもとに、バイナリアレィを特定の Java オブジェクトタイプにデコードします。

MimeDecoder の仕組みを使用したゲートウェイには、ファイルと FTP ゲートウェイリスナーなどがあります。これらのゲートウェイは、"mimeType" または "mimeDecoder" プロパティで設定でき、特定の Mime タイプに適切な MimeDecoder 実装のインストールをトリガーします。

10.1. MIMEDECODER の実装

このプロセスは大変簡単です。

1. org.jboss.soa.esb.listeners.message.mime.MimeDecoder インターフェースを実装するクラスを作成します。
2. @MimeType アノテーションで新規作成されたクラスに注釈を付け、アノテーションの値として mime タイプを指定します。
3. META-INF/org.jboss.soa.esb.listeners.message.mime.decoders.lst で新規作成クラスを定義します。このファイルは実行時にクラスパスに存在している必要があります。モジュールにこのファイルがない場合、このファイルをお使いのモジュールソース／リソースに追加して、実行時にその場所に置いておくようにしてください。
4. (任意) MimeDecoder 実装がリスナー設定にアクセスする必要がある場合 (追加設定情報)、org.jboss.soa.esb.Configurable インターフェースを実装するクラスを持たせるようにしてください。

実例:

```
@MimeType("text/plain")
public class TextPlainMimeDecoder implements MimeDecoder, Configurable {

    private Charset encodingCharset;

    public void setConfiguration(ConfigTree configTree) throws
ConfigurationException {
        AssertArgument.isNotNull(configTree, "configTree");

        String encoding = configTree.getAttribute("encoding", "UTF-8");
        encodingCharset = Charset.forName(encoding);
    }

    public Object decode(byte[] bytes) throws MimeDecodeException {
        try {
            return new String(bytes, encodingCharset.name());
        } catch (UnsupportedEncodingException e) {
            throw new MimeDecodeException("Unexpected character encoding
error.", e);
        }
    }
}
```


}

10.2. ゲートウェイ実装での MIMEDECODER の使用

リスナーインスタンス設定 (ConfigTree) と MimeDecoder.Factory クラスファクトリメソッドを使うのが、インストールした Mime デコーダーを使用した MessageComposer の最も簡単な方法です。

```
this.mimeDecoder = MimeDecoder.Factory.getInstanceByConfigTree(config);
```

これは“mimeType” または “mimeDecoder” 設定プロパティのいずれかを指定するリスナー設定に左右されます (ファイルと FTP リスナーにサポートされる):

```
<fs-listener name="FileGateway1" busidref="fileChannel1" is-gateway="true"
              poll-frequency-seconds="10">
    <property name="mimeType" value="text/plain" />
    <property name="encoding" value="UTF-8" />
</fs-listener>
<fs-listener name="FileGateway2" busidref="fileChannel2" is-gateway="true"
              poll-frequency-seconds="10">
    <property name="mimeDecoder"
value="com.acme.mime.ImageJPEGMimeDecoder" />
</fs-listener>
```

MessageComposer インスタンスが単に mimeDecoder インスタンスでデコードメソッドを呼び出すだけで、トランスポートのペイロードを実際にデコードします。

```
Object decodedPayload = mimeDecoder.decode(payloadBytes);
```

作成された ESB Message インスタンスに“decodedPayload” オブジェクトインスタンスを設定します。

10.3. 事前定義された MIMEDECODER 実装

JBossESB 4.8 には、以下の MimeDecoder 実装があります。まだ実装されていない MimeDecoder が必要な場合、「MimeDecoder の実装」の本章で前述されている説明に従ってください。

10.3.1. text/plain

TextPlainMimeDecoder は “text/plain” データを処理して、byte[] を String (デフォルト) または char[] にデコードします。

表10.1 プロパティ

プロパティ	説明	コメント
encoding	byte[] でエンコードされている text/plain データの文字エンコード	デフォルト “UTF-8”

プロパティ	説明	コメント
decodeTo	<p>text/plain データのデコーディング方法</p> <ul style="list-style-type: none">“STRING” (デフォルト): text/plain データを java.lang.String にデコード“CHARS”: text/plain データを char[] にデコード	デフォルト “STRING”

第11章 WEB サービスサポート

11.1. JBOSSWS

JBoss Enterprise Service Bus には、Web サービスのエンドポイントを公開し呼び出すためのコンポーネントを複数あります。

1. **SOAPProcessor** アクションにより、**JBossWS 2.x** 以降の Web サービスエンドポイントを、ESB 上で稼働しているリスナーより公開することができます。これは、独自の Web サービスを提供しないサービスでも公開が可能です。

SOAPProcessor アクションは *SOAP on the bus* と呼ばれます。

2. **SOAPClient** アクションにより、Web サービスエンドポイント上で呼び出しができます。

SOAPClient アクションは *SOAP off the bus* と呼ばれる場合もあります。



注記

これらのコンポーネントについて学習するには、**サービスガイド**を参照してください。このガイドには、設定方法が詳細にわたり記載されています。configure them.

このトピックに関する情報は、JBoss ESB に同梱されている "Wiki" にもあります。JBossESB は現在、JBossWS-Native と JbossWS-CXF スタックの両方をサポートしています。

第12章 事前定義されたアクション

このセクションでは、JBoss ESB にデフォルトで含まれるすべてのアクションのカタログを提供します。

12.1. トランスフォーマーとコンバーター

コンバーター/トランスフォーマーは、メッセージペイロードを相互に変換するアクションプロセッサの分類です。

特別に指定されていない限り、これらのすべてのアクションはメッセージペイロードを取得および設定するために **MessagePayloadProxy** を使用します。

12.1.1. ByteArrayToString

入力タイプ	byte[]
クラス	org.jboss.soa.esb.actions.converters.ByteArrayToString

byte[] ベースのメッセージペイロードを取得して **java.lang.String** オブジェクトインスタンスに変換します。

表12.1 ByteArrayToString のプロパティ

プロパティ	説明	必須
encoding	メッセージバイト配列のバイナリデータエンコーディング。デフォルトは UTF-8 。	No

例12.1 設定例

```
<action name="transform"
class="org.jboss.soa.esb.actions.converters.ByteArrayToString">
  <property name="encoding" value="UTF-8" />
</action>
```

12.1.2. LongToDateConverter

入力タイプ	java.lang.Long/long
出力タイプ	java.util.Date
クラス	org.jboss.soa.esb.actions.converters.LongToDateConverter

ロングベースのメッセージペイロードを取得し、**java.util.Date** オブジェクトインスタンスに変換します。

例12.2 設定例

```
<action name="transform"
class="org.jboss.soa.esb.actions.converters.LongToDateConverter">
```

12.1.3. ObjectInvoke

入力タイプ	ユーザーオブジェクト
出力タイプ	ユーザーオブジェクト
クラス	org.jboss.soa.esb.actions.converters.ObjectInvoke

メッセージペイロードとしてバインドされたオブジェクトを取得し、処理するよう設定された「プロセッサ」に提供します。処理結果は新しいペイロードとしてメッセージに再びバインドされます。

表12.2 ObjectInvoke のプロパティ

プロパティ	説明	必須
class-processor	メッセージペイロードの処理に使用されるプロセッサクラスのランタイムクラス名です。	Yes
class-method	メソッドを処理するために使用されるプロセッサクラスのメソッド名。	No

例12.3 設定例

```
<action name="invoke"
class="org.jboss.soa.esb.actions.converters.ObjectInvoke">
  <property name="class-processor" value="org.jboss.MyXXXProcessor"/>
  <property name="class-method" value="processXXX" />
</action>
```

12.1.4. ObjectToCSVString

入力タイプ	ユーザーオブジェクト
出力タイプ	java.lang.String
クラス	org.jboss.soa.esb.actions.converters.ObjectToCSVString

メッセージペイロードとしてバインドされたオブジェクトを取得し、提供されたメッセージオブジェクトとカンマ区切りの "bean-properties" リストプロパティに基づいたカンマ区切り値 (CSV) 文字列に変換します。

表12.3 ObjectToCSVString プロパティ

プロパティ	説明	必須
bean-properties	出力 CSV 文字列のために CSV 値を取得するために使用するオブジェクト Bean プロパティ名の一覧。オブジェクトは一覧の各プロパティに対して getter メソッドをサポートする必要があります。	Yes
fail-on-missing-property	プロパティがオブジェクトで不明な場合 (つまり、オブジェクトがプロパティの getter メソッドをサポートしない場合) にアクションが失敗するかどうかを示すフラグ。デフォルト値は false です。	No

例12.4 設定例

```
<action name="transform"
  class="org.jboss.soa.esb.actions.converters.ObjectToCSVString">
  <property name="bean-properties"
    value="name,address,phoneNumber"/>
  <property name="fail-on-missing-property"
    value="true" />
</action>
```

12.1.5. ObjectToXStream

入力タイプ	ユーザーオブジェクト
出力タイプ	java.lang.String
クラス	org.jboss.soa.esb.actions.converters.ObjectToXStream

メッセージペイロードとしてバインドされるオブジェクトを取得し、XStream プロセッサを使用して XML に変換します。 <http://xstream.codehaus.org/> を参照してください。

表12.4 ObjectToXStream のプロパティ

プロパティ	説明	必須
class-alias	シリアル化前に XStream.alias(String, Class) の呼出しで使用されるクラスエイリアス。デフォルトでは入力オブジェクトのクラス名に設定されます。	No
exclude-package	生成された XML からパッケージ名を除外します。デフォルト値は true です。class-alias が指定されている場合は適用できません。	No
aliases	XStream が XML エlement をオブジェクトに変換できるように追加のエイリアスを指定します。	No

プロパティ	説明	必須
namespaces	XStream によって生成された XML に追加する必要があるネームスペースを指定します。各 namespace-uri はこのネームスペースが現れるエレメントである local-part に関連付けられます。	No
xstream-mode	使用する XStream モードを指定します。設定可能な値は XPATH_RELATIVE_REFERENCES (デフォルト値)、 XPATH_ABSOLUTE_REFERENCES 、 ID_REFERENCES 、または NO_REFERENCES です。	No
fieldAliases	Xstream に追加されたフィールドエイリアス	No
implicit-collections	Xstream で登録されます。	No
converters	Xstream で登録されるコンバーター一覧	No

例12.5 設定例

```

<action name="transform"
class="org.jboss.soa.esb.actions.converters.ObjectToXStream">
  <property name="class-alias" value="MyAlias" />
  <property name="exclude-package" value="true" />
  <property name="aliases">
    <alias name="alias1" class="com.acme.MyXXXClass1" />
    <alias name="alias2" class="com.acme.MyXXXClass2" />
    <alias name="xyz" class="com.acme.XyzValueObject"/>
    <alias name="x" class="com.acme.XValueObject"/>
    ...
  </property>
  <property name="namespaces">
    <namespace namespace-uri="http://www.xyz.com" local-
part="xyz"/>
    <namespace namespace-uri="http://www.xyz.com/x" local-
part="x"/>
    ...
  </property>
  <property name="fieldAliases">
    <field-alias alias="aliasName" definedIn="className"
fieldName="fieldName"/>
    <field-alias alias="aliasName" definedIn="className"
fieldName="fieldName"/>
    ...
  </property>
  <property name="implicit-collections">
    <implicit-collection class="className" fieldName="fieldName"
      fieldType="fieldType" itemType="itemType"/>
    ...
  </property>
  <property name="converters">
    <converter class="className" fieldName="fieldName"
      fieldType="fieldType"/>

```



```

...
</property>
</action>

```

12.1.6. XStreamToObject

入力タイプ	<code>java.lang.String</code>
出力タイプ	ユーザーオブジェクト ("incoming-type" プロパティによって指定されます)
クラス	<code>org.jboss.soa.esb.actions.converters.XStreamToObject</code>

メッセージペイロードとしてバインドされる XML を取得し、XStream プロセッサを使用してオブジェクトに変換します。 <http://xstream.codehaus.org/> 参照してください。

表12.5 XStreamToObject のプロパティ

プロパティ	説明	必須
class-alias	シリアル化中に使用されるクラスエイリアス。デフォルトでは入力オブジェクトのクラス名に設定されます。	No
exclude-package	XML にパッケージ名を含めるかどうかを示すフラグ。	YES
incoming-type	クラスタイプ	Yes
root-node	XML の実際のルートノードとは異なるルートノードを指定します。XPath 式を取得します。	No
aliases	XStream が XML エレメントをオブジェクトに変換できるように追加のエイリアスを指定します。	No
attribute-aliases	Xstream が XML 属性をオブジェクトに変換できるように追加の属性エイリアスを指定します。	No
fieldAliases	Xstream に追加されたフィールドエイリアス	No
implicit-collections	Xstream で登録されます。	No
converters	Xstream が XML エレメントと属性をオブジェクトに変換できるようにコンバーターを指定します。コンバーターの詳細については、 http://xstream.codehaus.org/converters.html を参照してください。	No

例12.6 設定例

```

<action name="transform"
class="org.jboss.soa.esb.actions.converters.XStreamToObject">
  <property name="class-alias" value="MyAlias" />

```



```

<property name="exclude-package" value="true" />
<property name="incoming-type" value="com.acme.MyXXXClass" />
<property name="root-node" value="/rootNode/MyAlias" />
<property name="aliases">
  <alias name="alias1" class="com.acme.MyXXXClass1"/>
  <alias name="alias2" class="com.acme.MyXXXClass2"/>
  ...
</property>
<property name="attribute-aliases">
  <attribute-alias name="alias1" class="com.acme.MyXXXClass1"/>
  <attribute-alias name="alias2" class="com.acme.MyXXXClass2"/>
  ...
</property>
<property name="fieldAliases">
  <field-alias alias="aliasName" definedIn="className"
fieldName="fieldName"/>
  <field-alias alias="aliasName" definedIn="className"
fieldName="fieldName"/>
  ...
</property>
<property name="implicit-collections">
  <implicit-colletion class="className" fieldName="fieldName"
fieldType="fieldType"
itemType="itemType"/>
  ...
</property>
<property name="converters">
  <converter class="className" fieldName="fieldName"
fieldType="fieldType"/>
  ...
</property>
</action>

```

12.1.7. XsltAction

全ドキュメントに変換を行います。フラグメントごとに変換が必要な場合。

表12.6 XsltAction プロパティ

プロパティ	説明	必須
get-payload-location	メッセージペイロードに含まれるメッセージボディの位置 未指定のデフォルトのペイロード位置を使用する場合	NO
set-payload-location	結果となるペイロードが配置されるメッセージボディの位置。 未指定のデフォルトのペイロード位置を使用する場合	No
templateFile	XSL テンプレートファイルへのパス。デプロイアーカイブまたは URL 内のファイルパスで定義します。	Yes

プロパティ	説明	必須
resultType	<p>結果となるメッセージペイロードとして設定される結果のタイプ。</p> <p>このプロパティは、変換の出力結果を制御します。現在利用可能な値は以下のとおりです。</p> <ul style="list-style-type: none"> ● STRING: String を作成します。 ● BYTES: バイト配列 (byte[]) を作成します。 ● DOM: DOMResult を作成します。 ● SAX: SAXResult を作成します。 ● 上記だけではニーズを満たせない場合、SOURCERESULT を使用してカスタマイズした結果を作成することができます。 <p>メッセージペイロードに SourceResult オブジェクト (org.jboss.soa.esb.actions.transformation.xslt.SourceResult) が含まれている場合、これはペイロードの SourceResult オブジェクトの result 属性として、同じタイプの結果を生み出します。</p> <p>メッセージペイロードが SourceResult オブジェクトで、resultType が SOURCERESULT に設定されていない場合、結果は resultType で指定されているタイプで返されます。開発者は、このタイプに互換があることを確認するようにしてください。</p>	No
failOnWarning	<p>true の場合、変換警告が発生して例外が送出されます。false の場合、問題があることがロギングされます。</p> <p>デフォルトは true</p>	No
uriResolver	<p>URIResolver を実装するクラスの完全修飾クラス名。これは、変換ファクトリに設定されます。</p>	No
factory.feature.*	<p>変換ファクトリに設定されるファクトリ機能。完全修飾 URI である機能名は factory.feature. 接頭辞のあとに指定する必要があります。</p> <p>例：factory.feature.http://javax.xml.XMLConstants/feature/secure-processing</p>	No
Factory.attribute.*	<p>変換ファクトリに設定されるファクトリ属性。属性名は factory.attribute. 接頭辞のあとに指定する必要があります。</p> <p>例：factory.attribute.someVendorAttributename</p>	NO
validation	<p>True の場合、ソース文書が無効になり、例外が送出されます。False の場合、整形式の文書が強制されたとしても、検証が行われます。「XsltAction 検証」 を参照してください。</p> <p>デフォルト値 false</p>	No

プロパティ	説明	必須
schemaFile	使用する入力スキーマファイル (XSD)。クラスパスに置かれている。 「XsltAction 検証」 を参照してください。	No
schemaLanguage	使用する入力スキーマ言語。 「XsltAction 検証」 を参照してください。	No

12.1.7.1. XsltAction 検証

XsltAction 検証の設定には、様々な方法があります。以下に例とともに示しています。

1. Disabled (デフォルト)

これは、明示的に **false** に設定するか、省略され検証を無効にすることができます。

```
<property name="validation" value="false"/>
```

2. DTD

```
<property name="validation" value="true"/>
<property name="schemaLanguage" value="http://www.w3.org/TR/REC-xml"/>
```

または

```
<property name="validation" value="true"/>
<property name="schemaLanguage" value=""/>
```

3. W3C XML Schema または RELAX NG

```
<property name="validation" value="true"/>
```

または

```
<property name="validation" value="true"/>
<property name="schemaLanguage"
value="http://www.w3.org/2001/XMLSchema"/>
```

または

```
<property name="validation" value="true"/>
<property name="schemaLanguage"
value="http://relaxng.org/ns/structure/1.0"/>
```

4. W3C XML Schema または schemaFile が含まれる RELAX NG

```
<property name="validation" value="true"/>
<property name="schemaFile" value="/example.xsd"/>
```

または


```
<property name="validation" value="true"/>
<property name="schemaLanguage"
value="http://www.w3.org/2001/XMLSchema"/>
<property name="schemaFile" value="/example.xsd"/>
```

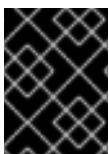
または

```
<property name="validation" value="true"/>
<property name="schemaLanguage"
value="http://relaxng.org/ns/structure/1.0"/>
<property name="schemaFile" value="/example.rng"/>
```

検証が有効かどうかで、XsltAction に対する結果は複数あります。

1. XML が整形形式で、有効な場合
 - 変換が実行されます。
 - パイプラインが継続されます。
2. XML が不正な場合
 - エラーはロギングがされます。
 - SAXParseException -> ActionProcessingException
 - パイプラインが停止されます。
3. XML が整形形式であるが無効な場合
 - 検証が有効でない場合
 - 変換が失敗します。
 - パイプラインが継続されます。
 - 検証が有効な場合
 - エラーはロギングがされます。
 - SAXParseException -> ActionProcessingException
 - パイプラインが停止されます。

12.1.8. SmooksTransformer



重要

SmooksTransformer アクションは今後廃止予定です。より一般的な目的を持ち、より柔軟な Smooks アクションクラスについては、SmooksAction を参照してください。

クラス

`org.jboss.soa.esb.actions.converters.SmooksTransformer`

JBossESB でのメッセージ変換は SmooksTransformer コンポーネントによってサポートされています。これは Smooks データ変換/処理フレームワークを ESB アクション処理パイプラインにプラグインできるようにする ESB アクションコンポーネントです。

SmooksTransformer コンポーネントでは、広範なソース (XML、CSV、EDI など) とターゲット (XML、Java、CSV、EDI など) データ形式がサポートされています。また、1 つのフレームワーク内に広範な変換テクノロジーもサポートされています。

詳細情報は、<http://www.smooks.org> の Web サイトを参照してください。

表12.7 SmooksTransformer リソースの設定:

プロパティ	説明	必須
resource-config	Smooks リソース設定ファイル。	Yes

表12.8 SmooksTransformer メッセージプロファイルプロパティ (オプション)

プロパティ	説明	必須
from	メッセージ交換参加者名、メッセージ生成者	No
from-type	"from" メッセージ交換参加者によって作成されたメッセージタイプ/形式。	No
to	メッセージ交換参加者名、メッセージ消費者	No
to	メッセージ交換参加者名、メッセージ消費者	No
to-type	"to"メッセージ交換参加者によって消費されたメッセージタイプ/形式。	No

上記のすべてのプロパティはメッセージにプロパティとして提供することにより上書きできます (Message.Properties)。

例12.7 設定例: デフォルトの入力/出力

```
<action name="transform"
class="org.jboss.soa.esb.actions.converters.SmooksTransformer">
  <property name="resource-config" value="/smooks/config-01.xml" />
</action>
```

例12.8 設定例: 名前付きの入力/出力

```
<action name="transform"
class="org.jboss.soa.esb.actions.converters.SmooksTransformer">
  <property name="resource-config" value="/smooks/config-01.xml" />
  <property name="get-payload-location" value="get-order-params" />
  <property name="set-payload-location" value="get-order-response" />
</action>
```


例12.9 設定例：メッセージプロファイルの使用

```
<action name="transform"
class="org.jboss.soa.esb.actions.converters.SmooksTransformer">
  <property name="resource-config" value="/smooks/config-01.xml" />
  <property name="from" value="DVDStore:OrderDispatchService" />
  <property name="from-type" value="text/xml:fullFillOrder" />
  <property name="to" value="DVDWarehouse_1:OrderHandlingService" />
  <property name="to-type" value="text/xml:shipOrder" />
</action>
```

Java オブジェクトは、**Message.Body** under their beanId の配下で **Message.Body** にバインドされます。詳細は、<http://community.jboss.org/wiki/MessageTransformation> を参照してください。

12.1.9. SmooksAction

SmooksAction クラス (org.jboss.soa.esb.smooks.SmooksAction) は Smooks 「プロセス」を実行する第2世代 ESB アクションクラスです (単にメッセージを変換する以外のこと (分割など) ができます)。SmooksTransformer アクションは今後の ESB のリリースで廃止 (最後は削除) される予定です。

SmooksAction クラスプロセス (Smooks PayloadProcessor を使用) は、文字列、バイト配列、InputStreams、リーダー、POJO などの広範な ESB メッセージペイロードを処理できます。したがって、Java 対 Java の変換を含む広範な変換を実行できます。また、コンテンツベースのペイロード分割およびルーティング (ESB メッセージルーティングではない) を含むソースメッセージストリームに対する他の種類の操作も実行できます。SmooksAction を使用すると、JBoss ESB 内のすべての Smooks 機能を有効にできます。

Smooks の Web サイトには、Smooks ユーザーガイド (およびその他のドキュメント) があります。また、Smooks のチュートリアルも確認してください。

重要

Smooks はベースの <resource-config> から行うリソース設定のうち特定の設定エラータイプを検出 (エラーの報告もなし) しません。たとえば、リソース (<resource>) が Smooks Visitor 実装で、Visitor クラスの名前のスペルを間違えている場合、Smooks はこのスペルミス部分がクラスであるべきなのかを把握しないため、エラーとして扱いません。Smooks は、Java Visitor 実装だけでなく、多様なリソースタイプをサポートしている点を忘れないでください。

この問題を最も簡単に回避する方法は (JBoss ESB 4.5.GA 以降) 拡張 Smooks 設定の名前空間を事前定義済みの機能すべてに使用することです。例えば、org.milyn.javabean.BeanPopulator <resource-config> を定義することで、Java でバインドする設定を定義する代わりに、http://www.milyn.org/xsd/smooks/javabean-1.2.xsd 設定の名前空間を使用してください (<jb:bean> 設定など)。

新規の Smooks Visitor 機能を実装した場合、この問題を最も簡単に回避する方法はこの新規リソースタイプの拡張設定名前空間を定義することです。また、IDE に組み込んだスキーマサポートを活用できるため、この新しいリソースを簡単に設定することができるという利点もあります。

これらの拡張名前空間設定に関する例については、Smooks v1.1+ ドキュメントを参照してください。

12.1.9.1. SmooksAction の設定

以下は基本的な SmooksAction 設定を示しています。

例12.10 SmooksAction

```
<action name="transform" class="org.jboss.soa.esb.smooks.SmooksAction">
  <property name="smooksConfig" value="/smooks/order-to-java.xml" />
</action>
```

オプションの設定プロパティ:

表12.9 SmooksAction オプションとなる設定プロパティ

プロパティ	説明	デフォルト値
get-payload-location	メッセージペイロードに含まれるメッセージボディの位置	デフォルトのペイロード位置
set-payload-location	結果となるペイロードが配置されるメッセージボディの位置。	デフォルトのペイロード位置
mappedContextObjects	ESB メッセージボディの EXECUTION_CONTEXT_ATTR_MAP_KEY マップにマッピングされるコンマ区切りの Smooks ExecutionContext オブジェクト。デフォルトは空の一覧です。オブジェクトはシリアル化できる必要があります。	

プロパティ	説明	デフォルト値
resultType	結果となるメッセージペイロードとして設定される結果のタイプ。「 結果タイプの指定 」を参照してください。	STRING
javaResultBeanId	resultType=JAVA の場合のみ該当 resultType が "JAVA" の場合に結果としてマッピングされる Smooks Bean コンテキスト beanId。指定されていない場合は、Bean コンテキスト Bean Map が JAVA の結果としてマッピングされます。	
reportPath	Smooks 実行レポートを生成するときのパスおよびファイル名。これは開発での使用を目的としており、本番稼働での使用は目的としていません。	

Smooks Execution Report の詳細情報は、http://www.smooks.org/mediawiki/index.php?title=V1.2:Smooks_v1.2_User_Guide#Checking_the_Smooks_Execution_Process を参照してください。

12.1.9.2. メッセージ入力ペイロード

SmooksAction は ESB メッセージでメッセージペイロードを取得および設定するために ESB MessagePayloadProxy クラスを使用します。したがって、get-payload-location と set-payload-location アクションプロパティを使用してそのように設定されていない限り、SmooksAction はデフォルトのメッセージ位置にあるメッセージペイロードを取得および設定します (つまり、Message.getBody().get() and Message.getBody().set(Object) を使用)。

上述したように、SmooksAction は広範なメッセージペイロードタイプを自動的にサポートします。つまり、アクションチェーンで SmooksAction よりも前に “fixup” アクションを実行しなくても SmooksAction 自体はほとんどのペイロードタイプを処理できます。

12.1.9.3. XML、EDI、CSV などの入力ペイロード

SmooksAction を使用してこれらのメッセージタイプを処理するには、ソースメッセージを以下の形式で提供します。

1. 文字列
2. InputStream
3. Reader
4. バイトアレイ

これ以外に、該当するメッセージタイプを処理するために標準的な Smooks 設定 (ESB 設定ではなく Smooks 設定) を行う必要があります。たとえば、XML ソースではない場合 (EDI、CSV など) はパーサーを設定します。

12.1.9.4. Java 入力ペイロード

提供されたメッセージペイロードがタイプ String、InputStream、Reader、または byte[] のいずれかでない場合、SmooksAction は JavaSource としてペイロードを処理し、Java から XML、Java 対 Java などの変換を実行できるようになります。

Smooks コアランタイムは、XML、EDI、Java などの入力ソースが作成する SAX イベントのストリームを処理し、これらのイベントが Visitor ロジックをトリガーすることで機能します。Java ソースの場合、Smooks は XStream を使用して、この SAX イベントのストリームを生成します。

しかし、出力形式を作成するため、テンプレートのみ (通常、FreeMaker テンプレートなど) を Java Source オブジェクトモデルに適用したい場合もあります。これを行うには、イベントのストリームを生成してはいけない Smooks コアランタイムに通知する必要があります。以下の 2 種類の方法で、これを行うことができます。

1. **Smooks.filterSource** が渡される **JavaSource** インスタンス上で **setEventStreamRequired(false)** を呼び出す

```
JavaSource javaSource = new JavaSource(orderBean);

// Turn streaming off via the JavaSource...
javaSource.setEventStreamRequired(false);

smooks.filterSource(javaSource, result);
```

2. あまは、Smooks 設定の読み込み機能をオフにする

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd">

    <reader>
        <features>
            <setOff
feature="http://www.smooks.org/sax/features/generate-java-event-stream" />
        </features>
    </reader>

    <!-- Other Smooks configurations e.g. a FreeMarker template... ->

</smooks-resource-list>
```

FreeMarker テンプレートの適用時、テンプレートのコンテキスト bean の名前 (テンプレートで使用する名前) は、JavaSource のオブジェクトタイプにより決まります。

- オブジェクトが Map の場合、Map インスタンスはテンプレートコンテキストになり、テンプレート内の bean 名に Map エントリキーを使用するだけでよくなります。
- Map 以外のオブジェクトは、JavaSource クラスがオブジェクトクラス SimpleName を取り、JavaBean プロパティ名をそこから作成します。これは、テンプレート作成に使用するコンテキスト bean の名前です。例えば、Bean クラス名は **com.acme.Order** で、テンプレート作成用のコンテキスト bean 名は **order** になります。

12.1.9.5. 結果タイプの指定

Smooks アクションはさまざまな結果タイプを生成できるため、必要な結果タイプを指定する必要があります。これは、ESB メッセージペイロード位置に再びバインドされる結果に影響します。

デフォルトでは、ResultType は “STRING” ですが、“resultType” 設定プロパティを設定することにより “BYTES”、“JAVA”、または “NORESULT” に設定することもできます。

“JAVA” の resultType を指定すると、Smooks ExecutionContext (特に Bean コンテキスト) から 1 つまたは複数の Java オブジェクトを選択できます。Bean コンテキストから ESB メッセージペイロード位置にバインドされる特定の Bean を指定できるようにすることにより、javaResultBeanId 設定プロパティは resultType プロパティと組み合わせられます。以下は、“order” Bean をメッセージペイロードとして Smooks Bean コンテキストから ESB メッセージにバインドする例です。

例12.11 結果タイプの指定

```
<action name="transform" class="org.jboss.soa.esb.smooks.SmooksAction">
  <property name="smooksConfig" value="/smooks/order-to-java.xml" />
  <property name="resultType" value="JAVA" />
  <property name="javaResultBeanId" value="order" />
</action>
```

12.1.10. PersistAction

入力タイプ	メッセージ
出力タイプ	入力メッセージ
クラス	org.jboss.soa.esb.actions.MessagePersister

これは MessageStore と対話するために使用されます (必要な場合)。

表12.10 PersistAction のプロパティ

プロパティ	説明	必須
classification	メッセージが保存される場所を分類するために使用されます。メッセージプロパティ org.jboss.soa.esb.messagestore.classification がメッセージで定義されている場合は、そのメッセージプロパティが代わりに使用されます。それ以外の場合は、デフォルト値がインスタンス作成時に提供されることがあります。	Yes
message-store-class	MessageStore の実装	Yes
terminal	パイプラインを終了するためにアクションが使用される場合、この値は true (デフォルト値) に設定する必要があります。それ以外の場合は、この値を false に設定します (入力メッセージが処理から返されます)。	No

例12.12 PersistAction

```
<action name="PersistAction"
class="org.jboss.soa.esb.actions.MessagePersister">
  <property name="classification" value="test"/>
```



```
<property name="message-store-class"
value="org.jboss.internal.soa.esb.persistence.format.db.DBMessageStoreImpl"/>
</action>
```

12.2. ビジネスプロセス管理

12.2.1. JBPM - BpmProcessor

入力タイプ	AbstractCommandVehicle.toCommandMessage() で生成される org.jboss.soa.esb.message.Message
出力タイプ	Message – 入力メッセージと同じ
クラス	org.jboss.soa.esb.services.jbpm.actions.BpmProcessor

JBossESB は BpmProcessor アクションを使用して JBPM への呼出しを行うことができます。JBPM から JBossESB を呼び出す方法については **Services Guide** の「JBPM 統合」の章も参照してください。BpmProcessor アクションは JBPM コマンド API を使用して JBPM への呼び出しを行います。

以下の JBPM コマンドが実装されています。

- NewProcessInstanceCommand
- StartProcessCommand
- CancelProcessInstanceCommand
- GetProcessInstanceVariablesCommand

表12.11 BpmProcessor のプロパティ

プロパティ	説明	必須
command	呼び出される JBPM コマンド。 必須 の許容値: <ul style="list-style-type: none"> • NewProcessInstanceCommand • StartProcessInstanceCommand • SignalCommand • CancelProcessInstanceCommand 	Yes
processdefinition	process-definition-id プロパティが使用されない場合の New- および Start-ProcessInstanceCommands の必須プロパティ。このプロパティの値は JBPM にすでにデプロイされ、新しいインスタンスを作成したいプロセス定義を参照する必要があります。このプロパティは Signal- および CancelProcessInstance-Commands に適用されません。	場合による

プロパティ	説明	必須
process-definition-id	processdefinition プロパティが使用されない場合の New- および Start-ProcessInstanceCommands の必須プロパティ。このプロパティの値は新しいインスタンスを作成したい JBPM の processdefinition id を参照する必要があります。このプロパティは Signal- および CancelProcessInstanceCommands に適用されません。	場合による
actor	New- および StartProcessInstanceCommands のみに適用される JBPM actor id を指定するオプションプロパティ。	No
key	JBPM キーの値を指定するオプションプロパティ。たとえば、このキーの値として一意の請求書 id を渡すことができます。JBPM 側では、このキーは “ビジネス” キー id フィールドになります。このキーはプロセスインスタンスに対する文字列ベースのビジネスキープロパティです。ビジネスキーが提供される場合、ビジネスキーとプロセス定義の組み合わせは一意である必要があります。EsbMessage から必要な値を抽出するためにキーの値は MVEL 式を保持できます。たとえば、メッセージのボディに “businessKey” という名前の名前付きパラメーターがある場合は、 “body.businessKey” を使用します。このプロパティは New- および StartProcessInstanceCommands に対してのみ使用されることに注意してください。	No
transition-name	オプションパラメーター。このプロパティは StartProcessInstance- および Signal Commands にのみ適用され、現在のノードから複数の移行が存在する場合にのみ使用されます。このプロパティが指定されていない場合は、ノードからデフォルトの移行が取得されます。デフォルトの移行は JBPM processdefinition.xml の該当するノードに対して定義された移行の一覧の最初の移行です。	No
esbToBpmVars	New- および StartProcessInstanceCommands、SignalCommand のオプションプロパティ。このプロパティは EsbMessage から抽出し、特定のプロセスインスタンスに対する JBPM コンテキストに設定する必要がある変数の一覧を定義します。この一覧はマッピングエレメントから構成されます。各マッピングエレメントは以下の属性を持つことができます。 <ul style="list-style-type: none"> ● esb: EsbMessage のどこからでも値を抽出する MVEL 式を含むことができる必須属性。 ● bpm: JBPM 側で使用される名前を含むオプション属性。省略すると、esb 名が使用されます。 ● default: esb MVEL 式が EsbMessage で設定された値を見つけることができない場合にデフォルト値を保持できるオプション属性。 	No

12.3. スクリプト

スクリプトアクションプロセッサはスクリプト言語を使用してアクション処理ロジックの定義をサポートします。

12.3.1. GroovyActionProcessor

クラス	<code>org.jboss.soa.esb.actions.scripting.GroovyActionProcessor</code>
-----	--

Groovy アクション処理スクリプトを実行し、メッセージ、payloadProxy、アクション設定、およびロガーを入力変数として受け取ります。

表12.12 GroovyActionProcessor のプロパティ

プロパティ	説明	必須
script	Groovy スクリプトへのパス (クラスパス上)。	
supportMessageBasedScripting	メッセージ内でスクリプトの使用を許可します。	
cacheScript	スクリプトをキャッシュします。デフォルトの設定は true です。	No

表12.13 GroovyAction Processor Script Binding の変数

変数	説明
message	メッセージ
payloadProxy	メッセージペイロードのユーティリティ (MessagePayloadProxy)
config	アクション設定 (ConfigTree)
logger	GroovyActionProcessor のスタティック Log4J Logger (Logger)。

例12.13 設定例

```
<action name="process"
class="org.jboss.soa.esb.scripting.GroovyActionProcessor">
  <property name="script" value="/scripts/myscript.groovy"/>
</action>
```

12.3.2. ScriptingAction

クラス	<code>org.jboss.soa.esb.actions.scripting.ScriptingAction</code>
-----	--

Bean スクリプトフレームワーク (BSF) を使用してスクリプトを実行すると、メッセージ、payloadProxy、アクション設定、ロガーを入力変数として受け取ります。

- JBoss ESB 4.8 には BSF 2.3.0 が含まれています。このバージョンは、BSF 2.4.0 に比べ言語サポートが少なくなっています (例えば、Groovy がない、非機能の Rhino など)。今後のバージョンには、Groovy と Rhino をサポートする BSF 2.4.0 が含まれる予定です。

2. BSF はスクリプトをプリコンパイル、キャッシュ、および再使用する API を提供しません。このため、ScriptingAction のを実行するたびに再びコンパイルを行うことになります。パフォーマンス要件を評価する場合はこのことを考慮してください。
3. アプリケーションに BeanShell スクリプトを含める場合は、.bsh の代わりに .beanshell 拡張子を使用することが推奨されます。.beanshell 拡張子を使用しないと、JBoss BSHDeployer がそのスクリプトを取得することがあります。

表12.14 ScriptingAction のプロパティ

プロパティ	説明	必須
script	スクリプトへのパス (クラスパス上)。	
supportMessageBasedScripting	メッセージ内でスクリプトの使用を許可します。	
language	オプションのスクリプト言語 (拡張機能の減少よりも優先されます)。	No

表12.15 ScriptingAction Processor Script Binding の変数

変数	説明
message	メッセージ
payloadProxy	メッセージペイロードのユーティリティ (MessagePayloadProxy)
config	アクション設定 (ConfigTree)
logger	ScriptingAction の静的な Log4J Logger (Logger)

例12.14 ScriptingAction

```
<action name="process"
class="org.jboss.soa.esb.scripting.ScriptingAction">
  <property name="script" value="/scripts/myscript.beanshell"/>
</action>
```

12.4. サービス

ESB サービス内で定義されたアクション。

12.4.1. EJBProcessor

入力タイプ	EJB のメソッド名とパラメータ
-------	------------------

出力タイプ	EJB 固有のオブジェクト
クラス	org.jboss.soa.esb.actions.EJBProcessor

入力メッセージを取得してステートレスセッション Bean の呼び出しにその内容を使用します。このアクションは EJB2.x と EJB3.x に対応します。

表12.16 EJBProcessor のプロパティ

プロパティ	説明	必須
ejb3	これが EJB3.x セッション Bean の呼び出しの場合	
ejb-name	EJB の ID。ejb3 が true の場合はオプションです。	
jndi-name	関連する JNDI ルックアップです。	
initial-context-factory	JNDI ルックアップのメカニズムです。	
provider-url	関連するプロバイダです。	
method	呼び出す EJB メソッド名。	
lazy-ejb-init	EJB がデプロイメント時ではなくランタイム時に遅延して初期化されるべきかどうか。デフォルトは false。	No
ejb-params	メソッドを呼び出すときにメソッドが存在する入力メッセージの位置で使用するパラメーターの一覧。	
esb-out-var	出力の位置 (デフォルト値は DEFAULT_EJB_OUT)。	No

例12.15 EJB 2.x の設定例

```
<action name="EJBTest" class="org.jboss.soa.esb.actions.EJBProcessor">
  <property name="ejb-name" value="MyBean" />
  <property name="jndi-name" value="ejb/MyBean" />
  <property name="initial-context-factory"
value="org.jnp.interfaces.NamingContextFactory" />
  <property name="provider-url" value="localhost:1099" />
  <property name="method" value="login" />
  <!-- Optional output location, defaults to "DEFAULT_EJB_OUT"
  <property name="esb-out-var" value="MY_OUT_LOCATION"/> -->
  <property name="ejb-params">
    <!-- arguments of the operation and where to find them in the
message -->
    <arg0 type="java.lang.String">username</arg0>
    <arg1 type="java.lang.String">password</arg1>
  </property>
</action>
```


例12.16 EJB 3.x の設定例

```
<action name="EJBTest" class="org.jboss.soa.esb.actions.EJBProcessor">
  <property name="ejb3" value="true" />
  <property name="jndi-name" value="ejb/MyBean" />
  <property name="initial-context-factory"
value="org.jnp.interfaces.NamingContextFactory" />
  <property name="provider-url" value="localhost:1099" />
  <property name="method" value="login" />
  <!-- Optional output location, defaults to "DEFAULT_EJB_OUT"
  <property name="esb-out-var" value="MY_OUT_LOCATION"/> -->
  <property name="ejb-params">
    <!-- arguments of the operation and where to find them in the
message -->
    <arg0 type="java.lang.String">username</arg0>
    <arg1 type="java.lang.String">password</arg1>
  </property>
</action>
```

12.5. ルーティング

ルーティングアクションは2つ以上のメッセージ交換参加者間でのメッセージの条件ルーティングをサポートします。

12.5.1. Aggregator

クラス	org.jboss.soa.esb.actions.Aggregator
-----	---

メッセージ集約アクション。Aggregator Enterprise Integration Pattern の実装については、<http://www.enterpriseintegrationpatterns.com/Aggregator.html> を参照

このアクションは正しい相関データを持つすべてのメッセージに依存します。このデータは“aggregatorTag” (Message.Properties) という名前のプロパティとしてメッセージに設定されます。ContentBasedRouter と StaticRouter アクションを参照してください。

このデータは形式は以下のとおりです。

```
[UUID] ":" [message-number] ":" [message-count]
```

すべてのメッセージがアグリゲータによって受け取られた場合は、Message.Attachment 一覧 (名前なし) の一部としてすべてのメッセージに含まれる新しいメッセージが返されます。それ以外の場合、アクションは null を返します。

表12.17 アグリゲータのプロパティ

プロパティ	説明	必須
timeoutInMillis	集約プロセスがタイムアウトするまでのミリ秒単位の時間です。	No

例12.17 設定例

```
<action class="org.jboss.soa.esb.actions.Aggregator" name="Aggregator">
  <property name="timeoutInMillis" value="60000"/>
</action>
```

12.5.2. EchoRouter

単に受信メッセージペイロードを情報ログストリームにエコーし、処理メソッドから入力メッセージを返します。

12.5.3. HttpRouter

クラス	<code>org.jboss.soa.esb.actions.routing.http.HttpRouter</code>
-----	--

このアクションは、外部の (ESB 非対応) HTTP エンドポイントを ESB アクションパイプラインから呼び出しできます。このアクションは、背後で Apache Commons HttpClient を使用します。

表12.18 Apache Commons HttpRouter

プロパティ	説明	必須
unwrap	true (デフォルト) の場合は送信前にメッセージオブジェクトからメッセージペイロードを抽出します。 false の場合は、MessageType に従い、シリアル化されたメッセージを XML または Base64 でコンコードされた Java シリアル化オブジェクトとして送信します。	No
endpointUrl	メッセージが転送されるエンドポイント	Yes
http-client-property	HttpRouter は、HttpClientFactory を使用して HttpClient インスタンスを作成、設定します。ローカルのファイルシステム、クラスパス、URI でプロパティファイルを参照するファイルプロパティを使用することで、ファクトリ設定を指定します。これがどのように行われるか、以下の例を参照してください。ファクトリプロパティに関する詳細情報は、 http://www.jboss.org/community/docs/DOC-9969 を参照してください。	No
method	現在 GET と POST のみのサポートです。	Yes
responseType	応答が返される形式を指定します。STRING か BYTES になり、デフォルト値は STRING です。	No
headers	リクエストに追加。複数の <code><header name="test" value="testvalue" /></code> 要素をサポートします。	No
MappedHeaderList	対象のエンドポイントに伝搬されるべきヘッダー名をコンマ区切りの一覧にしたもの。ヘッダーの値は、http-gateway から ESB に入るリクエストまたは、現在の ESB メッセージのプロパティ内にあるリクエストにある値をリトリブします。	No

例12.18 設定例


```

<action name="httprouter"
class="org.jboss.soa.esb.actions.routing.http.HttpRouter">
  <property name="endpointUrl" value="http://host:80/blah">
    <http-client-property name="file" value="/ht.props"/>
  </property>
  <property name="method" value="GET"/>
  <property name="responseType" value="STRING"/>
  <property name="headers">
    <header name="blah" value="blahval" ></header>
  </property>
</action>

```

12.5.4. JMSRouter

クラス	<code>org.jboss.soa.esb.actions.routing.JMSRouter</code>
-----	--

受信メッセージを JMS にルーティングします。

表12.19 JMSRouter

プロパティ	説明	必須
unwrap	true の場合は送信前にメッセージオブジェクトからメッセージペイロードを抽出します。 false (デフォルト値) の場合は、シリアル化されたメッセージオブジェクトが送信されます。	No
jndi-context-factory	使用する JNDI コンテキストファクトリ。デフォルト値は org.jnp.interfaces.NamingContextFactory です。	No
jndi-URL	使用する JNDI URL。デフォルト値は 127.0.0.1:1099 です。	No
jndi-pkg-prefix	使用する JNDI 命名パッケージ接頭辞。デフォルト値は org.jboss.naming:org.jnp.interfaces です。	No
connection-factory	使用する ConnectionFactory の名前です。デフォルトは ConnectionFactory です。	No
persistent	persistent: JMS DeliveryModdy。 true (デフォルト値) または false を設定します。	No
priority	使用される JMS 優先度。デフォルト値は javax.jms.Message.DEFAULT_PRIORITY です。	No
time-to-live	使用される JMS Time-To-Live。デフォルト値は javax.jms.Message.DEFAULT_TIME_TO_LIVE です。	No
security-principal	JMS 接続の作成時に使用するセキュリティプリンシパル。	Yes
security-credentials	JMS 接続の作成時に使用するセキュリティクレデンシャル。	Yes

プロパティ	説明	必須
property-strategy	JMSPropertiesSetter interface の実装 (デフォルト値をオーバーライドする場合)。	No
message-prop	メッセージで設定されるプロパティには message-prop という接頭辞が付けられます。	No
jndi-prefixes	コンマで区切った接頭辞の文字列。これら接頭辞を持つプロパティは JNDI 環境に追加されます。	No
jndiName	メッセージが送信されるべきキューまたはトピック名を定義します。	No

12.5.5. EmailRouter

クラス	<code>org.jboss.soa.esb.actions.routing.email.EmailRouter</code>
-----	--

設定された e-メールアカウントへ受信メッセージをルーティングします。

表12.20 EmailRouter プロパティ

プロパティ	説明	必須
unwrap	true の場合は送信前にメッセージオブジェクトからメッセージペイロードを抽出します。false (デフォルト値) の場合は、シリアル化されたメッセージオブジェクトが送信されます。	
host	SMTP サーバーのホスト名。指定されていない場合は、jbossesb-properties.xml のデフォルトプロパティ 'org.jboss.soa.esb.mail.smtp.host' が使用されます。	
port	SMTP サーバーのポート。指定されていない場合は、jbossesb-properties.xml のデフォルトプロパティ 'org.jboss.soa.esb.mail.smtp.port' が使用されます。	
username	SMTP サーバーのユーザー名。指定されていない場合は、jbossesb-properties.xml のデフォルトプロパティ 'org.jboss.soa.esb.mail.smtp.user' が使用されます。	
password	SMTP サーバーの上記ユーザー名に対するパスワード。指定されていない場合は、jbossesb-properties.xml のデフォルトプロパティ 'org.jboss.soa.esb.mail.smtp.password' が使用されます。	
auth	true の場合、AUTH command を使用してユーザー認証を試行します。指定されていない場合は、jbossesb-properties.xml のデフォルトプロパティ 'org.jboss.soa.esb.mail.smtp.auth' が使用されます。	
from	送信元の e-メールアドレス	
sendTo	送信先の e-メールアカウント	

プロパティ	説明	必須
subject	E-メールの件名	
messageAttachmentName	メッセージペイロードを含む添付のファイル名 (オプション)。指定されない場合は、メッセージペイロードがメッセージボディに含まれます。	
message	電子メールメッセージを構成する ESB メッセージコンテンツに追加する文字列 (オプション)	
ccTo	カンマ区切り一覧形式の電子メールアドレス (オプション)	
attachment	送信メールに追加される添付ファイルを含む子要素	

例12.19 設定例

```
<action name="send-email"
class="org.jboss.soa.esb.actions.routing.email.EmailRouter">
  <property name="unwrap" value="true" />
  <property name="host" value="smtpHost" />
  <property name="port" value="25" />
  <property name="username" value="smtpUser" />
  <property name="password" value="smtpPassword" />
  <property name="from" value="jbossesb@xyz.com" />
  <property name="sendTo" value="system2@xyz.com" />
  <property name="subject" value="Message Subject" />
</action>
```

12.5.6. ContentBasedRouter

クラス	org.jboss.soa.esb.actions.ContentBasedRouter
-----	---

コンテンツベースのメッセージルーティングアクション。

このアクションは、以下のルーティングルールプロバイダタイプをサポートします。

- XPath: 単純な XPath ルール。アクションのインライン定義、または .properties 形式ファイルを外部から定義
- Drools: Drools ルールファイル (DSL)。XPath ベースの DSL に対する事前定義サポート

表12.21 ContentBasedRouter プロパティ

プロパティ	説明	必須
-------	----	----

プロパティ	説明	必須
cbrAlias	コンテンツベースルーティングプロバイダーのエイリアス。対応値は、"Drools" (デフォルト)、"XPath"、"Regex"。	
ruleSet	外部定義のルールファイル。Drools ルールプロバイダーが使用されている場合は、Drools DSL ファイル。XPath または Regex プロバイダーが使用されている場合は、.properties ルールファイル。	
ruleLanguage	CBR 評価ドメイン固有言語 (DSL) ファイル。Drools ルールプロバイダーのみに該当	
ruleReload	ルールファイルを毎回再ロードするかどうかを示すフラグです。デフォルトは "false" です。	
ruleAuditType	Drools に監査ロギングを実行させる任意のプロパティ。ログを Drools Eclipse プラグインに読み込ませ確認することができます。有効な値は CONSOLE、FILE、THREADED_FILE で、デフォルトでは監査ロギングが実行されます。	
ruleAuditFile	監査ロギングのファイルパスを定義する任意のプロパティ。FILE あるいは THREADED_FILE ruleAuditType のみに適用されます。デフォルトは "event" です。JBoss Drools は ".log" を自動で追加します。このファイルのデフォルトの場所は "." (現在の作業ディレクトリ、つまり JBoss では bin/ ディレクトリ) となっています。	
ruleAuditInterval	監査イベントを監査ログにフラッシュする頻度を定義する任意のプロパティ。これは THREADED_FILE ruleAuditType のみに適用されます。デフォルトは 1000 (ミリ秒) です。	
destinations	<p><route-to> 設定のコンテナプロパティ。ルールが外部定義される場合、この設定は以下の形式になります。</p> <pre><route-to destination-name="express" service-category="ExpressShipping" service-name="ExpressShippingService"/></pre> <p>ルールが設定内にインライン定義されている場合、この設定は以下の形式を取ります (Drools プロバイダーへのサポートはなし)。</p> <pre><route-to service- category="ExpressShipping" service-name="ExpressShippingService" expression="/order[@statusCode='2']" /></pre>	
namespaces	<p>XPath ルールプロバイダーなどで必要な <namespace> 設定のコンテナプロパティ。<namespace> 設定は以下の形式をとります。</p> <pre><namespace prefix="ord" uri="http://acme.com/order" /></pre>	

表12.22 ContentBasedRouter プロセスのメソッド

プロパティ	説明	必須
process	メッセージに集約データを追加しないでください。	

「Aggregator」

Regex は XPath と同じ方法で設定します。唯一の違いは、XPath 表現の代わりに正規表現を使用することです。

コンテンツベースのルーティングに関する詳細情報は、[サービスガイド](#)の、「コンテンツベースルーティングとは」の章を参照してください。

12.5.7. StaticRouter

クラス	<code>org.jboss.soa.esb.actions.StaticRouter</code>
-----	---

スタティックメッセージルーティングアクション。これは基本的にコンテンツベースルーターの簡略化バージョンです (ただし、コンテンツベースルーティングルールをサポートしない点を除く)。

表12.23 StaticRouter のプロパティ

プロパティ	説明
destinations	<p><route-to> 構成のコンテナプロパティです。</p> <pre><route-to destination-name="express" service-category="ExpressShipping" service-name="ExpressShippingService"/></pre>

表12.24 StaticRouter プロセスのメソッド

method	説明
process	メッセージに集約データを追加しないでください。

「Aggregator」を参照してください。

例12.20 設定例 - StaticRouter

```
<action name="routeAction"
class="org.jboss.soa.esb.actions.StaticRouter">
  <property name="destinations">
    <route-to service-category="ExpressShipping" service-name="ExpressShippingService"/>
    <route-to service-category="NormalShipping" service-name="NormalShippingService"/>
  </property>
</action>
```


12.5.8. SyncServiceInvoker

クラス	<code>org.jboss.soa.esb.actions.SyncServiceInvoker</code>
-----	---

同期メッセージルーティングアクション。このアクションは、設定サービスに同期呼び出しを行い、呼び出しの応答をアクションパイプラインに渡し、後続のアクション (があれば、後続のアクション) により処理が行われるか、サービスが RequestResponse サービスの場合応答として渡されます。

表12.25 SyncServiceInvoker プロパティ

プロパティ	説明	必須
service-category	サービスカテゴリ	Yes
service-name	サービス名	Yes
failOnException	アクションが対象サービスの呼び出しからの例外で失敗させるかどうか。False に設定している場合は、アクションはパイプラインに入力メッセージを単に返して、サービス処理を継続できます。失敗のステータスを把握する必要がある場合は、このパラメーターを true に設定して、通常の "faultTo" の仕組みを使用してパイプラインが失敗できるようにします (デフォルトは true)。	No
suspendTransaction	このアクションは、トランザクションがアクティブな状態にある場合に実行されると失敗します。このプロパティを True に設定されている場合、トランザクションは一時停止される場合があります。デフォルトは、False です。	No
ServiceInvokerTimeout	呼び出しのタイムアウト (ミリ秒)。タイムアウトされると、例外が発生し "failOnException" 設定に従ったアクションが起こされます。デフォルトは 3000 です。	No

例12.21 設定例 - SyncServiceInvoker

```
<action name="route"
class="org.jboss.soa.esb.actions.SyncServiceInvoker">
  <property name="service-category" value="Services" />
  <property name="service-name" value="OM" />
</action>
```

12.5.9. StaticWireTap

クラス	<code>org.jboss.soa.esb.actions.StaticWireTap</code>
-----	--

StaticWiretap アクションは StaticRouter とは異なります。違いは、StaticWiretap はアクションチェーン上でリッスンして、それ以下のアクションを実行することができます。しかし、StaticRouter アクションは、使用時にアクションチェーンを終了します。そのため、StaticRouter チェーンで最後のアク

ションでなければなりません。

表12.26 StaticWireTap のプロパティ

プロパティ	説明	必須
destinations	<p><route-to> 構成のコンテナプロパティです。</p> <pre><route-to destination-name="express" service-category="ExpressShipping" service- name="ExpressShippingService"/></pre>	

表12.27 StaticWireTap プロセスのメソッド

method	説明
process	メッセージに集約データを追加しないでください。

「[Aggregator](#)」を参照してください。

例12.22 設定例 - StaticWireTap

```
<action name="routeAction"
class="org.jboss.soa.esb.actions.StaticWiretap">
  <property name="destinations">
    <route-to service-category="ExpressShipping" service-
name="ExpressShippingService"/>
    <route-to service-category="NormalShipping" service-
name="NormalShippingService"/>
  </property>
</action>
```

12.5.10. Email WireTap

クラス	org.jboss.soa.esb.actions.routing.email.EmailWiretap
-----	--

表12.28 Email WireTap プロパティ

プロパティ	説明
host	SMTP サーバーのホスト名。指定されていない場合は、jbossesb-properties.xml のデフォルトプロパティ 'org.jboss.soa.esb.mail.smtp.host' が使用されます。
port	SMTP サーバーのポート。指定されていない場合は、jbossesb-properties.xml のデフォルトプロパティ 'org.jboss.soa.esb.mail.smtp.port' が使用されます。

プロパティ	説明
username	SMTP サーバーのユーザー名。指定されていない場合は、jbossesb-properties.xml のデフォルトプロパティ 'org.jboss.soa.esb.mail.smtp.user' が使用されます。
password	SMTP サーバーの上記ユーザー名に対するパスワード。指定されていない場合は、jbossesb-properties.xml のデフォルトプロパティ 'org.jboss.soa.esb.mail.smtp.password' が使用されます。
auth	true の場合、AUTH command を使用してユーザー認証を試行します。指定されていない場合は、jbossesb-properties.xml のデフォルトプロパティ 'org.jboss.soa.esb.mail.smtp.auth' が使用されます。
from	送信元の e-メールアドレス
sendTo	送信先の e-メールアカウント
subject	E-メールの件名
messageAttachmentName	メッセージペイロードを含む添付のファイル名 (オプション)。指定されない場合は、メッセージペイロードがメッセージボディに含まれます。
message	電子メールメッセージを構成する ESB メッセージコンテンツに追加する文字列 (オプション)。
ccTo	カンマ区切り一覧形式の電子メールアドレス (オプション)
attachment	送信メールに追加される添付ファイルを含む子要素

例12.23 設定例 - Email Wiretap

```
<action name="send-email"
class="org.jboss.soa.esb.actions.routing.email.EmailWiretap">
  <property name="host" value="smtpHost" />
  <property name="port" value="25" />
  <property name="username" value="smtpUser" />
  <property name="password" value="smtpPassword" />
  <property name="from" value="jbossesb@xyz.com" />
  <property name="sendTo" value="systemX@xyz.com" />
  <property name="subject" value="Important message" />
</action>
```

12.6. NOTIFIER

クラス	org.jboss.soa.esb.actions.Notifier
-----	---

アクションパイプライン処理の結果に基づいて、設定で指定された一連の通知ターゲットに通知を送信します。

アクションパイプラインは、2つの段階 (通常の処理の後に結果の処理) で機能します。最初の段階では、パイプラインが、パイプラインの最後に達するまで、またはエラーが発生するまで順番に各アクション (デフォルトではプロセスと呼ばれます) のプロセスメソッドを呼び出します。この時点でパイプラインは逆方向に処理され (第2段階)、それぞれの前のアクションで結果メソッドを呼び出します (デフォルトは `processException` or `processSuccess`)。これは現在のアクション (成功した最後のアクションまたは例外が発生したアクション) で開始され、パイプラインの先頭に達するまで逆方向に移動します。Notifier は第1段階でメッセージの処理を行わず (no-op)、第2段階で指定された通知を送信するアクションです。

Notifier クラスの設定は `NotificationTargets` の一覧を指定するために使用できる `NotificationList` エレメントを定義するために使用されます。タイプ "ok" の `NotificationList` では、アクションのパイプライン処理が成功したときに通知を受け取るターゲットが指定されます。タイプ "err" の `NotificationList` では、これまでに説明したアクションパイプライン処理セマンティクスに応じてアクションパイプライン処理で例外が発生したときに通知を受け取るターゲットが指定されます。"err" と "ok" は大文字と小文字を区別します。

`NotificationTarget` に送信された通知はターゲット固有ですが、実質的にアクションパイプライン処理を行う ESB メッセージのコピーから構成されます。通知ターゲットタイプとそのパラメータの一覧は、このセクションの最後で示されます。

パイプラインを処理するアクションの各ステップで成功または失敗を通知する機能が必要な場合は、Notifier クラスを使用する `<action>` の代わりに各 `<action>` エレメントで "okMethod" と "exceptionMethod" 属性を使用します。

例12.24 設定例 - Notifier

```
<action name="notify" class="org.jboss.soa.esb.actions.Notifier"
  okMethod="notifyOK">
  <property name="destinations">
    <NotificationList type="OK">
      <target class="NotifyConsole" />
      <target class="NotifyFiles" >
        <file name="@results.dir@/goodresult.log" />
      </target>
    </NotificationList>
    <NotificationList type="err">
      <target class="NotifyConsole" />
      <target class="NotifyFiles" >
        <file name="@results.dir@/badresult.log" />
      </target>
    </NotificationList>
  </property>
</action>
```

通知はさまざまなタイプのターゲットに送信できます。以下の表は `NotificationTarget` タイプとそのパラメータを示しています。

12.6.1. NotifyConsole

クラス	NotifyConsole
-----	----------------------

コンソールに ESB メッセージのコンテンツを出力することにより通知を実行します。

例12.25 NotifyConsole

```
<target class="NotifyConsole" />
```

12.6.2. NotifyFiles

クラス	NotifyFiles
目的	ESB メッセージのコンテンツを指定された一連のファイルに書き込むことによって通知を実行します。
属性	なし
子	ファイル
子属性	<ul style="list-style-type: none">● append – 値が true の場合は既存のファイルに通知を付け加えます● URI – ファイルを指定する任意の有効な URI です

例12.26 NotifyFiles

```
<target class="NotifyFiles" >  
  <file append="true" URI="anyValidURI"/>  
  <file URI="anotherValidURI"/>  
</target>
```

12.6.3. NotifySqlTable

クラス	NotifySqlTable
目的	既存のデータベース表に記録を挿入することで通知を行います。 データベースの記録には ESB メッセージのコンテンツと、 オプションでネストされた <column> エレメントを使って指定される他の値が含まれます。
属性	<ul style="list-style-type: none">● driver-class● connection-url● user-name● password● table – 通知レコードが保存されるテーブル● dataColumn – ESB メッセージコンテンツが保存されるテーブルコラムの名前

子	列
子属性	<ul style="list-style-type: none"> • name – 追加の値を保存する表の列名です • value – 保存する値です

例12.27 NotifySqlTable

```
<target class="NotifySqlTable" driver-class="com.mysql.jdbc.Driver"
connection-url="jdbc:mysql://localhost/db"
user-name="user"
password="password"
table="table"
dataColumn="messageData">
<column name="aColumnName" value="aColumnValue"/>
</target>
```

12.6.4. NotifyQueues

クラス	NotifyQueues
目的	ESB メッセージ (添付されたプロパティを含む) を JMS メッセージに変換し、JMS メッセージをキューのリストに送信することによって通知を実行します。追加のプロパティは <messageProp> エレメントを使用して添付できます。
属性	なし
子	queue
子属性	<ul style="list-style-type: none"> • jndiName – キューの JNDI 名です。 Required • jndi-URL – JNDI プロバイダ URL です Optional • jndi-context-factory - JNDI 初期コンテキストファクトリです Optional • jndi-pkg-prefix – JNDI パッケージのプレフィックスです Optional • connection-factory - JMS 接続ファクトリの JNDI 名 です。 Optional、デフォルト設定は ConnectionFactory になります。
子	messageProp

子属性	<ul style="list-style-type: none">● name - 追加される新しいプロパティ名です。● value - 新しいプロパティの値です
-----	---

例12.28 NotifyQueues

```
<target class="NotifyQueues" >
<messageProp name="aNewProperty" value="theValue"/>
<queue jndiName="queue/quickstarts_notifications_queue" />
</target>
```

12.6.5. NotifyTopics

クラス	NotifyTopics
目的	ESB メッセージ (添付されたプロパティを含む) を JMS メッセージに変換し、JMS メッセージをトピックの一覧に公開することにより通知を実行します。追加のプロパティは <messageProp> エレメントを使用して添付できます。
属性	なし
子	topic
子属性	<ul style="list-style-type: none">● jndiName – キューの JNDI 名です。 Required● jndi-URL – JNDI プロバイダの URL です。 Optional● jndi-context-factory - JNDI の初期コンテキストファクトリです。 Optional● jndi-pkg-prefix – JNDI のパッケージプレフィックスです。 Optional● connection-factory - JMS 接続ファクトリの JNDI 名です。 Optional、デフォルトの設定は ConnectionFactory になります。
子	messageProp
子属性	<ul style="list-style-type: none">● name - 追加される新しいプロパティ名です。● value - 新しいプロパティの値です

例12.29 NotifyTopics

```
<target class="NotifyTopics" >
<messageProp name="aNewProperty" value="theValue"/>
```



```
<queue jndiName="queue/quickstarts_notifications_topic" />
</target>
```

12.6.6. NotifyEmail

クラス	NotifyEmail
目的	ESB メッセージコンテンツ (オプションで任意の添付ファイルも) を含む電子メールを送信することにより通知を実行します。
属性	なし
子	topic
子属性	<ul style="list-style-type: none"> • from – e-メールアドレス (javax.email.InternetAddress)。必須 • sendTo – カンマ区切り一覧形式の電子メールアドレス、必須 • ccTo – カンマ区切り一覧形式の電子メールアドレス (オプション) • subject – E-メールの件名 (必須) • message – 電子メールメッセージを構成する ESB メッセージコンテンツに追加する文字列 (オプション) • host - SMTP サーバーのホスト名。指定されていない場合は、jbossesb-properties.xml のデフォルトプロパティ org.jboss.soa.esb.mail.smtp.host が使用されます (オプション) • port - SMTP サーバーのポート。指定されていない場合は、jbossesb-properties.xml のデフォルトプロパティ org.jboss.soa.esb.mail.smtp.port が使用されます (オプション) • username - SMTP サーバーのユーザー名。指定されていない場合は、jbossesb-properties.xml のデフォルトプロパティ org.jboss.soa.esb.mail.smtp.user が使用されます (オプション) • password - SMTP サーバーのパスワード。指定されていない場合は、jbossesb-properties.xml のデフォルトプロパティ org.jboss.soa.esb.mail.smtp.password が使用されます (オプション) • auth - true の場合は、AUTH コマンドを使用してユーザー認証を試行します。指定されていない場合は、jbossesb-properties.xml のデフォルトプロパティ org.jboss.soa.esb.mail.smtp.auth が使用されます (オプション) • msgAttachmentName - メッセージペイロードを含む添付のファイル名。指定されない場合は、メッセージペイロードがメッセージボディに含まれます (Optional)。
子	添付です。 Optional

子テキスト	添付されるファイルの名前
-------	--------------

例12.30 NotifyEmail

```
<target class="NotifyEmail" from="person@somewhere.com"
sendTo="person@elsewhere.com"
subject="theSubject">
<attachment>attachThisFile.txt</attachment>
</target>
```

12.6.7. NotifyFTP

クラス	NotifyFTP
目的	ESB メッセージコンテンツを含むファイルを作成し、FTP を使用してリモートファイルシステムに転送することにより通知を実行します。
属性	なし
子	ftp
子属性	<ul style="list-style-type: none">● URI – 有効な FTP URL● filename – リモートシステムの ESB メッセージコンテンツを含むファイル名です。

例12.31 NotifyFTP

```
<target class="NotifyFTP" >
  <ftp URL="ftp://username:pwd@server.com/remote/dir"
filename="someFile.txt" />
</target>
```

12.6.8. NotifyFTPList

クラス	NotifyFTPList
-----	---------------

目的	<p>NotifyFTPList は NotifyFTP を継承し、単一のファイル名を取り、ESB メッセージオブジェクトにあるファイル名を一覧表示する機能を追加します。</p> <p>メッセージペイロードにあるファイルには、ファイル一覧 (完全パス) が含まれている必要があります。この一覧は反復され、"listFiles" プロパティが false の場合、一覧内のファイルはすべて、設定された 配信先の FTP サーバーディレクトリに送信されます。"listFiles" プロパティが true の場合は、各行には変換予定のファイル名が含まれており、ファイルが一行ごとに読み込まれます。</p> <p>そのため、以下を提供してください。</p> <ol style="list-style-type: none"> 1. 変換したい単一ファイル (listFiles = false の単一文字列ペイロード) 2. 変換したいファイル一覧 (listFiles = false の List<String> ペイロード) 3. 変換したい単一のファイル一覧 (listFiles = true の単一文字列ペイロード) 4. 変換したい一覧ファイルの一覧 (listFiles = true の List<String> ペイロード)
属性	なし
子	ftp
子属性	<ul style="list-style-type: none"> ● URI – 有効な FTP URL ● filename – リモートシステムの ESB メッセージコンテンツを含むファイル名です。 ● listFiles – メッセージペイロードに名前指定されているファイルが一覧ファイルの場合は true、そうでない場合は false。デフォルトは false ● deletelistFile – 一覧ファイルが削除される場合は true、されない場合は false。デフォルトは false

例12.32 NotifyFTP

```
<target class="NotifyFTPList">
  <ftp URL="ftp://username:password@localhost/outputdir"
    filename="{org.jboss.soa.esb.gateway.file}">
    listFiles="true"
    deletelistFile="true"
  </ftp>
</target>
```

12.6.9. NotifyTCP

クラス	NotifyTCP
目的	<p>TCP 経由でメッセージ送信。各接続は通知の期間のみ保持されます。</p> <p>文字列データペイロードの送信のみに対応。暗黙的 (文字列として)、またはバイトアレイとしてエンコード (byte[])</p>

属性	なし
子	デスティネーション (複数のデスティネーションに対応)
子属性	<ul style="list-style-type: none"> ● URI – データの書き込み先となる tcp アドレス、デフォルトポートは 9090

例12.33 NotifyFTP

```
<target class="NotifyTcp" >
  <destination URI="tcp://myhost1.net:8899" />
  <destination URI="tcp://myhost2.net:9988" />
</target>
```

12.7. WEBSERVICES/SOAP

12.7.1. SOAPProcessor

JBoss Webservices SOAP プロセッサー

このアクションは任意の JBossESB ホストリスナーからの JBossWS ホスト Web サービスエンドポイントの起動をサポートします。つまり、ESB は、Web サービスエンドポイントをまだ公開していないサービスに対して Web サービスエンドポイントを公開するために使用できます。これは、ターゲットサービスの呼び出しをラップするシンサービスラッパー Web サービス (JSR 181 実装など) を記述することによって行えます (ESB 上で実行されているエンドポイント (リスナー) を使用してサービスが公開されます)。また、これらのサービスは ESB によってサポートされている任意のトランスポートチャネル (http、ftp、jms など) を介して起動できます。



注記

SOAPProcessor は JBossWS-Native と JBossWS-CXF スタック両方をサポートします。

12.7.1.1. SOAPProcessor アクション設定

SOAPProcessor アクションに対する設定は非常に簡単です。このアクションは 1 つの必須プロパティ値のみを必要とします ("jbossws-endpoint" プロパティ)。このプロパティは SOAPProcessor が公開 (呼び出し) する JBossWS エンドポイントを指定します。

```
<action name="JBossWSAdapter"
class="org.jboss.soa.esb.actions.soap.SOAPProcessor">
  <property name="jbossws-endpoint" value="ABI_OrderManager" />
  <property name="jbossws-context" value="ABIV1OrderManager_war" />
  <property name="rewrite-endpoint-url" value="true" />
</action>
```

jbossws-endpoint

これは SOAPProcessor を公開する JBossWS エンドポイントです。必須。

jbossws-context

このオプションのプロパティは、Webservice のデプロイメントのコンテキスト名で、JBossWS エンドポイントを一意に特定するために使用することができます。

rewrite-endpoint-url

オプションの "rewrite-endpoint-url" プロパティは HTTP エンドポイントでの負荷分散をサポートします。この場合、Web サービスエンドポイントコンテナは WSDL の HTTP(S) エンドポイントアドレスをロードバランサーのものに設定するよう構成されます。"rewrite-endpoint-url" プロパティは、このような状況で HTTP エンドポイントアドレスの書き換えを無効にするために使用できます。これは非 HTTP プロトコルには影響しません。デフォルトは true です。

12.7.1.2. 依存関係

1. JBoss Application Server 4.2.3.GA
2. soap.esb サービス。これはディストリビューションの lib フォルダーにあります。

12.7.1.3. Web サービスエンドポイントデプロイメント

このアクションを使用して ESB リスナーを使用することにより任意の JBossWS Web サービスエンドポイントを公開できます。これには.esb デプロイメントの内部 (つまり、Web サービス .war は .esb 内部にバンドルされます) と外部 (たとえば、スタンドアロン Web サービス .war デプロイメント、.ear 内部にバンドルされる Web サービス .war デプロイメント) からデプロイされるエンドポイントが含まれます。ただし、これは .esb デプロイメントが JBoss Application Server にインストールされている場合のみこのアクションを使用できることを意味します (つまり JBossESB Server でサポートされません)。

12.7.1.3.1. エンドポイントの公開

管理ガイドの「規定の公開」を参照してください。

12.7.2. SOAPClient

SOAPClient アクションは JAXWS クライアントクラスを生成し、ターゲットサービスを呼び出すために Wise Client Service を使用します。

構成例:

```
<action name="soap-wise-client-action"
class="org.jboss.soa.esb.actions.soap.wise.SOAPClient">
  <property name="wsdl" value="http://host:8080/OrderManagement?wsdl"/>
  <property name="SOAPAction" value="http://host/OrderMgmt/SalesOrder"/>
</action>
```

表12.29 SOAPClient オプションのプロパティ

プロパティ	説明
wsdl	使用される WSDL
operationName	webservice WSDL で指定したオペレーション名

プロパティ	説明
SOAPAction	エンドポイントオペレーション。OperationName が優先されます。
EndPointName	起動された EndPoint。Web サービスは複数のエンドポイントを持つことができます。指定されない場合は、wsdl で最初に指定されたものが使用されます。
SmooksRequestMapper	要求に定義された java 対 java のマッピングを定義する smooks 設定ファイルを指定します。
SmooksResponseMapper	応答に対して定義された java 対 java のマッピングを定義するために smooks 設定ファイルを指定します。
serviceName	オブジェクト生成をキャッシュしたり、すでに生成されたオブジェクトを使用したりするために wise によって使用されるシンボリックなサービス名。提供されない場合、wise は wsdl のサーブレット名を使用します。
username	Web サービスが BASIC 認証 HTTP によって保護されている場合に使用されるユーザー名。
password	Web サービスが BASIC 認証 HTTP によって保護されている場合に使用されるパスワード。
smooks-handler-config	<p>SOAP 要求または応答 (特にヘッダの) の変換が必要になることがよくあります。これは一部の標準 SOAP ハンドラを追加するだけで行えます。Wise は JAXWS Soap ハンドラ (smooks に基づいたカスタムのハンドラまたは事前定義されたハンドラ) をサポートします。</p> <p>SOAP 要求の変換 (送信前) は SOAPClient アクションの Smooks 変換設定プロパティを設定することによってサポートされます。</p>
custom-handlers	また、一連のカスタム標準 JAXWS Soap ハンドラを提供することもできます。このパラメータは SoapHandler インターフェースを実装するクラスの一覧を受け取ります。クラスは完全修飾名を提供し、セミコロンによって区切る必要があります。
LoggingMessages	送信された soap メッセージや受信された応答を参照することはデバッグに役立ちます。Wise は System.out. Boolean 値で交換されるすべてのメッセージを出力する JAX-WS ハンドラを使用することによってこの目的を達成します。



重要

SOA の問題および HTTP 500 エラーが呼び出された webservice にある場合、JBoss Enterprise Service Bus SOAP UI は以下を行います。

1. "WARN [SOAPClient] Received status code '500' on HTTP SOAP (POST) request to" を出力します。
2. 不良を無視して、次のアクションに続きます。

12.7.2.1. SOAP 操作パラメーター

SOAP 操作パラメーターは以下の 2 つの方法で提供されます。

- デフォルトのボディ位置に設定された Map インスタンスとして使用されます (Message.getBody().add(Map))
- 名前付きボディ位置に設定された Map インスタンスとして使用されます (Message.getBody().add(String, Map))。ここで、ボディ位置の名前は "paramsLocation" アクションプロパティの値として指定されます。

パラメーター Map 自体も以下の 2 つの方法で値を設定できます。

1. 任意のタイプの一連のオブジェクト。この場合は、Smooks 設定をアクション属性 SmooksRequestMapper で指定する必要がある、Smooks は java 対 java の変換に使用されます。
2. 一連の文字列ベースのキー値ペア (<String, Object>)。ここでキーはキーの値が入力される wsdl (または生成されたクラス) で指定された SOAP パラメータの名前です。SOAP 応答メッセージ消費 Consumption

SOAP 応答オブジェクトインスタンスは、以下のいずれかの方法で ESB メッセージインスタンスに追加できます。

- デフォルトのボディ位置 (Message.getBody().add(Map))
- 名前付きボディ位置 (Message.getBody().add(String, Map))。ここで、ボディ位置の名前は "responseLocation" アクションプロパティの値として指定されます。

また、応答オブジェクトインスタンスには、以下のいずれかの方法で (SOAP 応答から) 値を入力できます。

1. 一連の任意のタイプのオブジェクト。この場合は、smooks 設定をアクション属性 SmooksResponseMapper で指定する必要がある、smooks は java 対 java の変換に使用されます。
2. 一連の文字列ベースのキー値ペア (<String, Object>)。ここで、キーはキーの値を入力する wsdl (または生成されたクラス) で指定された SOAP 回答の名前です。SOAP 要求/応答に対する JAX-WS ハンドラ

SOAPClient の使用例については、以下のクイックスタートを参照してください。

- webservice_consumer_wise は基本的な用途を示します。
- webservice_consumer_wise2 は、'SmooksRequestMapper' と 'SmooksResponseMapper' の使用方法を示します。
- webservice_consumer_wise3 は 'smooks-handler-config' の使用方法を示します。
- webservice_consumer_wise4 は 'custom-handlers' の用途を示します。

Wise の詳細については、Web サイト (<http://www.javainuxlabs.org/drupal/>) を参照してください。

12.7.2.1.1. JAXB Annotation Introductions

ネイティブ JBossWS SOAP スタックは SOAP に対するバインドに JAXB を使用します。これは、JBossWS エンドポイントを構築するために注釈なしタイプセットを使用できないことを意味します。この問題を解決するために "JAXB Annotation Introductions" と呼ばれる JBossESB と JBossWS の機能が提供されています。これは基本的に JAXB アノテーションに「紹介」する XML 設定を定義できることを意味します。

この XML 設定はエンドポイントデプロイメントの“META-INF”ディレクトリ内の“jaxb-intros.xml”という名前のファイルにパッケージ化する必要があります。

12.7.2.2. クイックスタート

このアクションの使用方法をデモするいくつかのクイックスタートが JBossESB ディストリビューション (サンプル/クイックスタート) で利用できます。“webservice_bpel”のクイックスタートをご覧ください。

12.7.3. SOAPClient

SOAP クライアントアクションプロセッサ

ターゲットサービスに対してメッセージを構築し、値を入力するために soapUI クライアントサービスを使用します。このアクションはサービスにメッセージをルーティングします。<http://www.soapui.org/>を参照してください。

12.7.3.1. エンドポイント操作の仕様

エンドポイント操作の指定は簡単にできます。これを行うには以下のように SOAPClient アクションで “wsdl” プロパティと “operation” プロパティを指定します。

```
<action name="soapui-client-action"
class="org.jboss.soa.esb.actions.soap.SOAPClient">
  <property name="wsdl"
value="http://localhost:18080/acme/services/RetailerCallback?wsdl"/>
  <property name="operation" value="SendSalesOrderNotification"/>
</action>
```

12.7.3.2. SOAP 要求メッセージの構築

SOAP 操作パラメーターは以下の 2 つの方法で提供されます。

1. デフォルトのボディ位置に設定された Map インスタンスとして使用されます (Message.getBody().add(Map))
2. 名前付きボディ位置の Map インスタンス (Message.getBody().add(String, Map))。ここで、そのボディ位置の名前は “get-payload-location” アクションプロパティの値として指定されます。

パラメーター Map 自体も以下の 2 つの方法で値を設定できます。

1. OGNL フレームワークを使用してアクセスされる一連のオブジェクト (SOAP メッセージパラメーターに対する)。OGNL の使用の詳細については、以降で説明します。
2. 文字列ベースのキー値ペア (<String, Object>)。ここで、キーはキーの値が入力される SOAP パラメーターを識別する OGNL 式です。OGNL の詳細については、以降で説明します。

上述したように OGNL は提供されたパラメーター Map から SOAP メッセージに挿入する SOAP パラメーター値を選択するために使用するメカニズムです。SOAP メッセージ内の固有のパラメーターの OGNL 式は、SOAP ボディ内のそのパラメーターの位置に依存します。以下のメッセージでは、customerNumber パラメーターを表現する OGNL 式は、“customerOrder.header.customerNumber”です。

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
```



```

xmlns:cus="http://schemas.acme.com">
<soapenv:Header/>
<soapenv:Body>

    <cus:customerOrder>
        <cus:header>
            <cus:customerNumber>123456</cus:customerNumber>
        </cus:header>
    </cus:customerOrder>

</soapenv:Body>
</soapenv:Envelope>

```

パラメータに対して OGNL 式が計算されると、このクラスは完全な OGNL 式でキーとされるオブジェクトに対して提供されたパラメータマップをチェックします (上記のオプション 1)。このようなパラメータオブジェクトがマップに存在しない場合、このクラスは OGNL ツールキットにマップと OGNL 式インスタンスを提供することによってパラメータをロードしようとします (上記のオプション 2)。これによって値が提供されない場合は、SOAP メッセージ内のこのパラメータ位置が空白のままになります。

上記のサンプルメッセージを取得し、"customerNumber" に値を入力するために「オプション 1」の方法を使用するには、キー "customerOrder" 下のパラメータマップにオブジェクトインスタンス ("Order" オブジェクトインスタンスなど) を設定する必要があります。"customerOrder" オブジェクトインスタンスには、"header" プロパティ ("Header" オブジェクトインスタンスなど) を含める必要があります。"header" プロパティの背後のインスタンス ("Header" オブジェクトインスタンスなど) は "customerNumber" プロパティを持つ必要があります。

コレクションに関連付けられた OGNL 式は少し異なる方法で構築されます。これは例を使用して説明するのが最もわかりやすいケースです。

```

<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:cus="http://schemas.active-
endpoints.com/sample/customerorder/2006/04/CustomerOrder.xsd"
  xmlns:stan="http://schemas.active-
endpoints.com/sample/standardtypes/2006/04/StandardTypes.xsd">

    <soapenv:Header/>
    <soapenv:Body>
        <cus:customerOrder>
            <cus:items>
                <cus:item>
                    <cus:partNumber>FLT16100</cus:partNumber>
                    <cus:description>Flat 16 feet 100
count</cus:description>
                    <cus:quantity>50</cus:quantity>
                    <cus:price>490.00</cus:price>
                    <cus:extensionAmount>24500.00</cus:extensionAmount>
                </cus:item>
                <cus:item>
                    <cus:partNumber>RND08065</cus:partNumber>
                    <cus:description>Round 8 feet 65
count</cus:description>
                    <cus:quantity>9</cus:quantity>
                    <cus:price>178.00</cus:price>
                    <cus:extensionAmount>7852.00</cus:extensionAmount>
                </cus:item>
            </cus:items>
        </cus:customerOrder>
    </soapenv:Body>
</soapenv:Envelope>

```



```

        </cus:item>
      </cus:items>
    </cus:customerOrder>
  </soapenv:Body>

```

```
</soapenv:Envelope>
```

上記のオーダーメッセージにはオーダー「アイテム」のコレクションが含まれます。コレクションの各エントリは "item" エlement によって表されます。オーダーアイテム "partNumber" に対する OGNL 式は "customerOrder.items[0].partnumber" と "customerOrder.items[1].partnumber" として構築されます。このことからわかるように、コレクションエントリ Element ("item" Element) は OGNL 式に明示的に現われず、インデックス表記で暗黙的に表されます。オブジェクトグラフという意味では (上記のオプション 1)、これは "items" リスト (リストまたはアレイ) を含むオーダーオブジェクトインスタンス (マップでは "customerOrder" がキーとされます) と "OrderItem" インスタンスであるリストエントリ ("partNumber" などのプロパティを含みます) で表すことができます。

オプション 2 (上述) はオブジェクトモデル (オプション 1) を作成せずに SOAP メッセージに値を入力するクイックかつダーティーな方法を提供します。SOAP 操作パラメータに対応する OGNL 式はオプション 1 のものとまったく同じです (ただし、オブジェクトグラフナビゲーションが関係しない点は除く)。OGNL 式は Map へのキーとして使用され、対応するキー値はパラメータです。

構築され、値が入力される SOAP メッセージテンプレートを表示するには、パラメーター Map に "dumpSOAP" パラメーターを追加します。これは、開発者にとって非常に役に立ち、開発で使用するべきです。

12.7.3.3. SOAP 応答メッセージ消費

SOAP 応答オブジェクトインスタンスは以下のいずれかの方法で ESB メッセージインスタンスに添付できます。

1. デフォルトのボディ位置 (Message.getBody().add(Map))
2. 名前付きボディ位置 (Message.getBody().add(String, Map))。ここでボディ位置の名前は "set-payload-location" アクションプロパティの値として指定されます。

応答オブジェクトインスタンスには以下の 3 つのいずれかの方法で値を入力できます (SOAP 応答から)。

1. XStream ツールキットにより作成し、値が入力されるオブジェクトグラフ。また、JAXB と JAXB のアノテーション導入を使用して応答のアンマーシャル化のサポートを追加することも計画されています。
2. 一連の文字列ベースのキー値ペア (<String, String>)。ここで、キーは SOAP 応答 Element を識別する OGNL 式であり、値は SOAP メッセージからの値を表す文字列です。
3. オプション 1 または 2 がアクション設定で指定されていない場合は、ロー SOAP 応答メッセージ (文字列) がメッセージに添付されます。

オブジェクトグラフ (上記のオプション 1) に値を入力するメカニズムとして XStream を使用することは簡単であり、XML と Java オブジェクトモデルがお互いに互換性がある限り問題ありません。

XStream を使用する方法 (オプション 1) は以下のようにアクションで設定されます。

```

<action name="soapui-client-action"
class="org.jboss.soa.esb.actions.soap.SOAPClient">
  <property name="wsdl"

```



```

value="http://localhost:18080/acme/services/RetailerService?wsdl"/>
  <property name="operation" value="GetOrder"/>
  <property name="get-payload-location" value="get-order-params" />
  <property name="set-payload-location" value="get-order-response" />
  <property name="responseXStreamConfig">
    <alias name="customerOrder" class="com.acme.order.Order"
      namespace="http://schemas.acme.com/services/CustomerOrder.xsd"
    />
    <alias name="orderheader" class="com.acme.order.Header"
      namespace="http://schemas.acme.com/services/CustomerOrder.xsd"
    />
    <alias name="item" class="com.acme.order.OrderItem"
      namespace="http://schemas.acme.com/services/CustomerOrder.xsd"
    />
  </property>
</action>

```

上記の例には、要求パラメータ Map と応答オブジェクトインスタンスのデフォルトでない名前付き位置を指定する方法の例も含まれています。

また、上記の XStream の設定オプション以外にファイル名マッピングと XStream の注釈付きクラスを指定する機能も提供されます。

```

<property name="responseXStreamConfig">
  <fieldAlias name="header" class="com.acme.order.Order"
    fieldName="headerFieldName" />
  <annotation class="com.acme.order.Order" />
</property>

```

エレメントのローカル名が Java クラスのフィールド名に対応しない場合はフィールドマッピングを使用して XML エlement と Java フィールドをマッピングできます。

SOAP 応答データを OGNL キーマップ (上記のオプション 2) に抽出し、ESB メッセージに添付するには、以下のように "responseXStreamConfig" プロパティを "true" の値を持つ "responseAsOgnlMap" プロパティに置き換えます。

```

<action name="soapui-client-action"
class="org.jboss.soa.esb.actions.soap.SOAPClient">
  <property name="wsdl"
value="http://localhost:18080/acme/services/RetailerService?wsdl"/>
  <property name="operation" value="GetOrder"/>
  <property name="get-payload-location" value="get-order-params" />
  <property name="set-payload-location" value="get-order-response" />
  <property name="responseAsOgnlMap" value="true" />
</action>

```

ロー SOAP メッセージを文字列 (オプション 3) として返すには、"responseXStreamConfig" プロパティと "responseAsOgnlMap" プロパティの両方を省略します。

12.7.3.4. HttpClient 設定

SOAPClient は Apache Commons HttpClient を使用して、SOAP リクエストを実行します。HttpClientFactory を使用して、HttpClient インスタンスを構築、設定します。SOAPClient での HttpClientFactory 設定の指定は非常に簡単です。以下のように "wsdl" プロパティに別のプロパティを追加するだけです。


```
<property name="wsdl" value="https://localhost:18443/active-
bpel/services/RetailerCallback?wsdl">
  <http-client-property name="file" value="/localhost-https-
18443.properties" >

  </http-client-property>
</property>
```

"file" のプロパティの値は、ファイルシステム、クラスパス、URI ベースのリソース (この順番) ベースのリソースとして評価されます。このリソースには、標準の java プロパティ形式の HttpClient 設定が含まれています。

これらのプロパティは、以下のようにアクション設定で直接設定することができます。

```
<property name="http-client-properties">
  <http-client-property name="http.proxyHost" value="localhost"/>
  <http-client-property name="http.proxyPort" value="8080"/>
</property>
```

12.7.4. SOAPProxy

SOAPProxy は外部の WS エンドポイント (例:.NET、別の外部 Java ベース AS、LAMP でホスト) の消費と、ESB 経由で WS エンドポイントを再度公開することにフォーカスをあてます。ESB は、最終コンシューマー／クライアント (例:.NET WinForm アプリケーション) と最終プロデューサー (例: RoR-hosted WS) の間にあります。この仲介の役目は、以下の問題を解決する抽象層を提供するためです。

- クライアントとサービス間の疎結合を強化します。クライアントもサービスも相手の存在を認識していません。
- クライアントは、リモートサービスのホスト名 / IP アドレスへ直接接続できなくなります。
- クライアントは、インバウンド／アウトバウンドパラメーターを変更する修正済みの WSDL を確認します。少なくとも、クライアントがもとのプロキシ化されたエンドポイントではなく、ESB の公開するエンドポイントを参照するように、WSDL が微調整されます。
- SOAP エンベロープ／ボディの変換は、インバウンドのリクエストとアウトバウンドのレスポンス両方に対する ESB アクションチェーン経由で開始できます。
- クライアントが ESB にある 2 つ以上のプロキシエンドポイント (それぞれ、自身の WSDL または変換、ルーティング要件) に接続できるため、サービスのバージョンングが可能です。ESB は適切なメッセージを適切なエンドポイントに送信し、最終レスポンスを提供します。
- ContentBasedRouter 経由の複雑なコンテキストベースルーティング

これ以外のメカニズムは適切ではありません。

- SOAPClient を使用して、ミラリングを行うわけではなく、外部の Web サービスを呼び出します。
- SOAPProducer は、内部デプロイされた JBoss WS サービスのみを実行します。
- HttpRouter は、簡単な WS プロキシ化には、過剰な手動設定が必要になります。
- EBWS は、SOAP エンベロープをストリッピングして、ボディにのみ渡します。

SOAPProxy アクション

- Web サービスのプロデューサーおよびコンシューマーの両方となります。
- 必要なのは、外部 wsdl を参照するプロパティのみです。
- wsdl は、オプションの wsdlTransform プロパティを介して自動的に変換されます。
- SOAP は http とつながっているとされています。wsdl が読み込まれ、http トラnsポートが定義されている場合は、それが使用されます。他のトラnsポート (jms) についても検討が必要です。
- http を使用する場合、HttpRouter プロパティのいずれかがオプションでオーバーライドとして適用されます。

wsdl が http トラnsポートを指定する場合、HttpRouter プロパティのいずれかを適用可能です。

その他の使用可能な設定プロパティについては、特定の SOAPProxyTransport 実装自体を参照します。

表12.30

プロパティ	説明	必須
wsdlTransform	柔軟な wsdl 変換ができるように <smooks-resource-list> xml 設定ファイル	No
wsdlCharset	UTF-8 でない場合、オリジナルの wsdl (インポートされたリソース) がエンコードされた文字セット。基盤のプラットフォームでサポートされたエンコーディングである場合、UTF-8 に変換されます。	No
endpointUrl	HttpRouter プロパティの例。ドメイン名を一致させることが SSL 証明書にとって重要な場合便利です。	No
file	Apache Commons HTTPClient プロパティファイル。SSL 経由で Web サービスへプロキシ化する場合に便利です。	No
clientCredentialsRequired	Basic 認証の認証情報がエンドクライアントから来る必要があるかどうか、ファイル内で指定された認証情報が代わりに使用されるかどうか。デフォルトは true です。	No

プロパティ	説明	必須
wsdl	<p>WS エンドポイントが再書き込みされ、ESB からの新しい wsdl として公開されるオリジナルの wsdl url。<definitions><service><port><soap:address ロケーション属性のプロトコルにあわせて、プロトコル固有の SOAPProxyTransport 実装を使用しています。</p> <p>値は、5つの異種スキーマをベースに場所を参照します。</p> <ul style="list-style-type: none"> • http:// <p>外部の web サーバーから wsdl をプルしたい場合</p> <p>例: http://host/foo/HelloWorldWS?wsdl</p> • https:// <p>SSL 経由で外部の web サーバーから wsdl をプルしたい場合</p> <p>例: https://host/foo/HelloWorldWS?wsdl</p> • file:// <p>wsdl がディスクに置かれている場合、ESB JVM でアクセス可能</p> <p>例: file:///tmp/HelloWorldWS.wsdl</p> <p>注記: 上記の例のスラッシュ 3 つ。これがあるため、絶対パスと相対ファイルパスを指定することができます。</p> • classpath:// <p>ESB アーカイブ内で wsdl のパッケージをしたい場合</p> <p>例: classpath:///META-INF/HelloWorldWS.wsdl</p> <p>注記: 上記の例のスラッシュ 3 つ。これがあるため、絶対パスと相対 classloader リソースパスを指定することができます。</p> • internal:// <p>wsdl が ESB デプロイメントと同じ JVM 内で JBossWS Web サービスにより提供される場合</p> <p>例: internal://jboss.ws:context=foo,endpoint=HelloWorldWS</p> <p>注記: 上記の用途で http または https の代わりに、このスキーマを使用すべきです。これは、サーバーの起動時に Tomcat が受信 http/s リクエストをまだ受け入れない可能性があるため、wsdl にサービスを提供できません。</p> 	Yes

例12.34 設定例 - 基本的なシナリオ

```
<action name="proxy"
class="org.jboss.soa.esb.actions.soap.proxy.SOAPProxy">
  <property name="wsdl" value="http://host/foo/HelloWorldWS?wsdl"/>
</action>
```

例12.35 設定例 - Basic 認証および SSL

```
<action name="proxy"
class="org.jboss.soa.esb.actions.soap.proxy.SOAPProxy">
  <property name="wsdl" value="https://host/foo/HelloWorldWS?wsdl"/>
  <property name="endpointUrl"
value="https://host/foo/HelloWorldWS"/>
  <property name="file" value="/META-INF/httpclient-
8443.properties"/>
  <property name="clientCredentialsRequired" value="true"/>
</action>
```

12.8. その他

その他のアクションプロセッサ

12.8.1. SystemPrintln

メッセージのコンテンツを出力する単純なアクション (**System.out.println**)。

メッセージコンテンツを XML としてフォーマットしようとします。

入力タイプ	java.lang.String
クラス	org.jboss.soa.esb.actions.SystemPrintln
プロパティ	<ul style="list-style-type: none"> • message - メッセージのプレフィックス。 Required • printfull - true の場合は、メッセージ全体が出力されます。その他の場合は、バイトアレイと添付ファイルのみが出力されます。 • outputstream - true の場合は System.out が使用されます。その他の場合は System.err が使用されます。

例12.36 SystemPrintln

```
<action name="print-before"
class="org.jboss.soa.esb.actions.SystemPrintln">
  <property name="message" value="Message before action XXX" />
</action>
```


12.8.2. SchemaValidationAction

これは、スキーマ (XSD や RELAX NG) ベースの XML メッセージの検証を行うための単純なアクションです。

入力タイプ	<code>java.lang.String</code>
クラス	<code>org.jboss.soa.esb.actions.validation.SchemaValidationAction</code>
プロパティ	<ul style="list-style-type: none"> • <code>schema</code> - 検証スキーマファイルのクラスパスのパス (例 <code>.xsd</code>). • <code>schemaLanguage</code> - (オプション) スキーマタイプ/言語。デフォルト: <code>"http://www.w3.org/2001/XMLSchema"</code> (XSD)

例12.37 SystemPrintln

```
<action name="val"
class="org.jboss.soa.esb.actions.validation.SchemaValidationAction">
  <property name="schema" value="/com/acme/validation/order.xsd"/>
</action>
```

12.9. HTTPCLIENT 設定

HttpClient 設定は、プロパティ設定を指定することで行います。以下は、Java プロパティファイルとして指定した場合のプロパティ例です。

```
# Configurators
configurators=HttpProtocol,AuthBASIC

# HttpProtocol config...
protocol-socket-
factory=org.apache.commons.httpclient.contrib.ssl.EasySSLProtocolSocketFactory
keystore=/packages/jakarta-tomcat-5.0.28/conf/chap8.keystore
keystore-passw=xxxxxx
https.proxyHost=localhost
https.proxyPort=443

# AuthBASIC config...
auth-username=tomcat
auth-password=tomcat
authscope-host=localhost
authscope-port=18443
authscope-realm=ActiveBPEL security realm
```

12.9.1. コンフィギュレーター

HttpClientFactory が必要とするプロパティは、`configurators` プロパティのみで、コンフィギュレーター

実装をコンマ区切りで指定します。各コンフィギュレーター実装は `HttpClient` インスタンスの様々なアスペクトを設定し、`org.jboss.soa.esb.http.Configurator` クラスの拡張、`configure(HttpClient, Properties)` メソッドの提供を行います。

表12.31 事前定義されたアクション

コンフィギュレーター	説明	必須
HttpProtocol	ソケットファクトリや SSL キーストア情報などの <code>HttpClient</code> ホスト、ポート、プロトコル情報の設定	Yes
AuthBasic	<code>HttpClient</code> に対する HTTP Basic 認証の設定	No
AuthNTLM	<code>HttpClient</code> に対する NTLM 認証の設定	No

`configurators` プロパティで指定した一覧にクラス名を追加することで、追加のコンフィギュレーターを作成、設定可能です。

12.9.1.1. HttpProtocol

HTTP トランスポートプロパティの設定

表12.32 プロパティ

プロパティ	説明	必須
HttpProtocol	ソケットファクトリや SSL キーストア情報などの <code>HttpClient</code> ホスト、ポート、プロトコル情報の設定	Yes
target-host-url	http/https エンドポイントの対象 URL	Yes
https.proxyHost	https 接続のプロキシホスト	No
https.proxyPort	Https 接続のプロキシポート、デフォルトのポートは 443	No
http.proxyHost	HTTP 接続のプロキシホスト	No
http.proxyPort	Http 接続のプロキシポート、デフォルトのポートは 80	No

プロパティ	説明	必須
protocol-socket-factory	<p>ソケットファクトリのオーバーライド。ProtocolSocketFactory または ProtocolSocketBuilderFactory インターフェースの実装</p> <p>http のデフォルト値は、httpclient DefaultProtocolSocketFactory で、https のデフォルト値は、contributed StrictSSLProtocolSocketFactory です。</p> <p>ESB コードベースで提供されている ProtocolSocketBuilderFactory の実装は、AuthSSLProtocolSocketBuilderFactory と SelfSignedSSLProtocolSocketBuilderFactory の 2 つです。それぞれ、AuthSSLProtocolSocketFactory ファクトリの設定と、自己署名の SSLContext を設定します。</p>	No
keystore	キーストアの場合	No
keystore-passw	キーストアのパスワードまたは暗号化ファイル	No
keystore-type	キーストアのタイプ、デフォルトは jks	No
truststore	TrustStore の場合	No
truststore-passw	トラストストアのパスワードまたは暗号化ファイル	No
truststore-type	トラストストアのタイプ、デフォルトは jks	No

12.9.1.2. AuthBASIC

HTTP Basic 認証プロパティの設定

表12.33 プロパティ

プロパティ	説明	必須
auth-username	認証ユーザー名	Yes
auth-password	認証パスワード	Yes
authscope-host	認証スコープホスト	Yes
authscope-port	認証スコープポート	Yes
authscope-domain	認証スコープドメイン	Yes

12.9.1.3. AuthNTLM

HTTP Basic 認証プロパティの設定

表12.34 プロパティ

プロパティ	説明	必須
ntauth-username	認証ユーザー名	Yes
ntauth-password	認証パスワード	Yes
ntauthscope-host	認証スコープホスト	Yes
ntauthscope-port	認証スコープポート	Yes
ntauthscope-domain	認証スコープドメイン	Yes
ntauthscope-realm	認証スコープレalm	No

第13章 カスタムアクションの開発

JBoss ESB は、カスタムのアクションを様々な方法で開発することができます。アクションごとに利点があり、以下のように分類できます。

- ライフサイクルアクション、`org.jboss.soa.esb.actions.ActionLifecycle` または `org.jboss.soa.esb.actions.ActionPipelineProcessor` を実装
- Java bean アクション、`org.jboss.soa.esb.actions.BeanConfiguredAction` を実装
- 注釈付きのアクション
- レガシーアクション

実装ごとの違いを理解するには、以下を把握する必要があります。

- アクションがどのように設定されるか
- いつアクションがインスタンス化されるか、実装のスレッドセーフティについて
- ライフサイクルイベントの視覚性があるかどうか
- アクションメソッドが直接呼び出されるか、リフレクションで呼び出されるか

13.1. プロパティを使用したアクションの設定

通常、アクションはタスクを実行するために外部の設定が必要なテンプレートとして動作します。たとえば、**PrintMessage** アクションは出力内容を示す `message` という名前のプロパティや出力回数を示す `repeatCount` というプロパティを受け取ることができます。多くの場合、`jboss-esb.xml` ファイル内のアクション設定は以下のようになります。

```
<action name="PrintAMessage" class="test.PrintMessage">
  <property name="information" value="Hello World!" />
  <property name="repeatCount" value="5" />
</action>
```

このマッピングがどのように行われるかを変更する場合、その方法はゲートウェイの種類によって異なります。

13.2. ライフサイクルアクション

ライフサイクルアクションは、ライフサイクルインターフェース (`org.jboss.soa.esb.actions.ActionLifecycle` や `org.jboss.soa.esb.actions.ActionPipelineProcessor`) からきています。ActionLifecycle は、パイプラインライフサイクルメソッド (`initialise` と `destroy`) を実装し、ActionPipelineProcessor で `process`、`processSuccess`、`processException` などのメッセージ処理メソッドを含むようにこのインターフェースを拡張します。

このインターフェースは管理されたライフサイクルを持つステートレスアクションの実装をサポートします。これらのインターフェースのいずれかを実装するクラスの単一インスタンスがパイプラインごとにインスタンス化され (アクション毎の設定)、スレッドセーフでなければなりません。`initialise` や `destroy` パイプラインのライフタイムの間、アクションがリソース管理を行えるように、メソッドをオーバーライドすることができます。たとえば、`initialise` メソッドで必要なリソースをキャッシュし、`destroy` メソッドで消去するなどです。

これらのアクションは、パラメーターとして **ConfigTree** インスタンスを1つとるコンストラクターを定義する必要があります。これは、パイプライン内の特愛知のアクションの設定を表記します。

アクションが適切なインターフェースを実装し、メソッド名が設定でオーバーライドされないという前提で、パイプラインはメソッドを直接呼び出します。インターフェースで実装されていないメソッドや、アクション設定でオーバーライドされたメソッドは、リフレクションを使用して呼び出されます。

開発を簡素化するには、コードベースに抽象ベースクラスが2つ提供されています。それぞれ、適切なインターフェースを実装し、**process** メソッド以外の空のスタブメソッドを提供します。この2つのクラスは、**org.jboss.soa.esb.actions.AbstractActionPipelineProcessor** と **org.jboss.soa.esb.actions.AbstractActionLifecycle** で、以下のように使用できます。

```
public class ActionXXXProcessor extends AbstractActionPipelineProcessor {
    public ActionXXXProcessor(final ConfigTree config) {
        // extract configuration
    }

    public void initialise() throws ActionLifecycleException {
        // Initialize resources...
    }

    public Message process(final Message message) throws
    ActionProcessingException {
        // Process messages in a stateless fashion...
    }

    public void destroy() throws ActionLifecycleException {
        // Cleanup resources...
    }
}
```

13.3. JAVA BEAN アクション

アクションプロパティを設定する他の方法はプロパティ名に対応するアクションにセッターを追加し、フレームワークがそれらにデータを自動的に入力するよう許可することです。アクション Bean にデータを自動的に入力するために、アクションクラスは

org.jboss.soa.esb.actions.BeanConfiguredAction マーカーインターフェースを実装する必要があります。たとえば、以下のクラスの動作は上記のものと同じです。

```
public class PrintMessage implements BeanConfiguredAction {
    private String information;
    private Integer repeatCount;
    public setInformation(String information) {
        this.information = information;
    }
    public setRepeatCount(Integer repeatCount) {
        this.repeatCount = repeatCount;
    }
    public Message process(Message message) {
        for (int i=0; i < repeatCount; i++) {
            System.out.println(information);
        }
    }
}
```




注記

setRepeatCount() の Integer パラメーターは XML で指定された String 形式から自動的に変換されます。

ロードプロパティの **BeanConfiguredAction** メソッドは単純な引数を取るアクションに適しており、**ConfigTree** メソッドは XML 形式のデータを直接扱う必要がある場合に適しています

これらのアクションは、ライフサイクルメソッドをサポートせず、パイプラインを通過する全メッセージについてはリフレクション経由で呼び出します。

13.4. 注釈付きアクションクラス

JBoss Enterprise Service Bus には、クリーンな **action** 実装をより簡単に構築できるアクションアノテーションがあります。これにより、インターフェースや抽象クラスの実装、**ConfigTree** タイプ (**jboss-esb.xml** ファイルの設定情報) の処理などの複雑な部分がなくなります。注釈付きアクションのインスタンスは、パイプラインごとにインスタンス化され (アクション毎の設定)、スレッドセーフでなければなりません。このパイプラインは常に、リフレクションを使用してアクションメソッドを使用して呼び出します。

以下がそのアノテーションです。

- @Process
- @ConfigProperty
- @Initialize
- @Destroy
- @BodyParam
- @PropertyParams
- @AttachmentParam
- @OnSuccess
- @OnException

13.4.1. @Process

最も簡単な実装は、@Process のアノテーション付きの単一メソッドと基本的な *plain old Java object* (POJO) でアクションを作成します。

```
public class MyLogAction {

    @Process
    public void log(Message message) {
        // log the message...
    }
}
```


@Process アノテーションは、有効な ESB **action** としてクラスを特定します。クラス内に複数のメソッドがある場合、メッセージインスタンスを処理するのに使用するメソッドを特定します (または、メッセージの一部。これについては、@BodyParam、@PropertyParam and @AttachmentParam アノテーションの説明時に詳しくみていきます)。

この **action** インスタンスを **pipeline** に設定するには、ロー／ベースレベルの **action** 実装と同じプロセスを使用します (**AbstractActionPipelineProcessor** を継承する、または **ActionLifecycle** を実装するもの、あるいはその他のサブタイプや抽象実装)：

```
<service .....>
  <actions>
    <action name="logger" class="com.acme.actions.MyLogAction" />
  </actions>
</service>
```

@Process のアノテーションがついた複数のメソッドに **action** 実装が関連付けられている場合、process 属性を使用して、メッセージインスタンスの処理に使用するものはどれか指定します。

```
<service .....>
  <actions>
    <action name="logger" class="com.acme.actions.MyLogAction"
              process="log" />
  </actions>
</service>
```

13.4.2. @Process メソッドの戻り値

以下を返すように @Process メソッドを実装することができます。

- void: これは、上記の Logger アクション実装にあるように、戻り値がないという意味です。
- message: これは ESB メッセージインスタンスです。 **action pipeline** 上でアクティブな／現在のインスタンスになります。
- その他のタイプ。メソッドが ESB メッセージインスタンスを返さない場合、返されたオブジェクトインスタンスは、 **action pipeline** にある現在の ESB メッセージインスタンスに設定されます。メッセージがどこに設定されるかは、set-payload-location <action> 設定プロパティにより変わります。これは、通常の **MessagePayloadProxy** ルールに従いデフォルト設定します。

13.4.3. @Process メソッドパラメーター

@Process メソッドを使用して、様々な方法でパラメーターを指定します。

1. メソッドパラメーターとして ESB メッセージインスタンスを指定
2. 任意のパラメータータイプを 1 つ以上指定。Enterprise Service Bus フレームワークは、アクティブ／現在のパイプラインメッセージインスタンス内でそのタイプのデータを検索します。まず、メッセージボディ、次にプロパティ、最後に添付を検索し、そのパラメーターの値としてこのデータを渡します (該当するものが見つからない場合は **null**)。

最初のオプションに関する例は、Logger アクションの上記の例に記述されています。2 つ目のオプションの例を以下に示しています。


```
public class OrderPersister {

    @Process
    public OrderAck storeOrder(OrderHeader orderHeader,
        OrderItems orderItems) {
        // process the order parameters and return an ack...
    }
}
```

この例では、@Process メソッドは、**OrderHeader** や **OrderItem** オブジェクトインスタンスの作成や、現在のメッセージへの添付について **pipeline** の 1 つ前のアクションに左右されます (より現実的な実装では、XML または EDI ペイロードを注文インスタンスにデコーディングする一般的なアクション実装を持ちます。**OrderPersister** は、単独のパラメーターとして注文インスタンスを取ります)。以下に例を示します。

```
public class OrderDecoder {

    @Process
    public Order decodeOrder(String orderXML) {
        // decode the order XML to an Order instance...
    }
}

public class OrderPersister {

    @Process
    public OrderAck storeOrder(Order order) {
        // persist the order and return an ack...
    }
}
```

サービス設定で 2 つのアクションをつなぎます。

```
<actions>
    <action name="decode" class="com.acme.orders.OrderDecoder" />
    <action name="persist" class="com.acme.orders.OrderPersister" />
</actions>
```

アノテーションが少ないため、オプション 2 のほうが読み込みが簡単ですが、実行時に適切なパラメーターをメッセージ内から検索する処理が**決定的でない**ため、リスクがあります。このため、Red Hat は @BodyParam、@PropertyParam、@AttachmentParam アノテーションをサポートします。

これらの @Process メソッドパラメーターのアノテーションを使用して、明示的にメッセージ内のどこから @Process メソッドの個別のパラメーター値がリトリブされているか定義します。名前の通り、これらのアノテーションはそれぞれ、パラメーターごとに名前付きのロケーションを指定することができます。

```
public class OrderPersister {

    @Process
    public OrderAck storeOrder(
        @BodyParam("order-header") OrderHeader orderHeader,
        @BodyParam("order-items") OrderItems orderItems) {
```



```

        // process the order parameters and return an ack...
    }
}

```

指定のメッセージロケーションに値が含まれていない場合、このパラメーターについて null が渡されます (@Process メソッドインスタンスはこの処理方法を決定)。反対に指定のロケーションに誤ったタイプの値が含まれている場合、**MessageDeliverException** がスローされます。

13.4.4. @ConfigProperty

アクションは、カスタム設定がある程度必要になるものがほとんどです。ESB アクション設定では、プロパティは <action> 要素の <property> サブ要素として提供されます。

```

<action name="logger" class="com.acme.actions.MyLogAction">
  <property name="logFile" value="logs/my-log.log" />
  <property name="logLevel" value="DEBUG" />
</action>

```

これらのプロパティを活用するため、低／基本レベルのアクション実装を使用する必要があります (**AbstractActionPipelineProcessor** の継承または、**ActionLifecycle** の実装)。これは、**ConfigTree** クラスと連携します (コンストラクター経由でアクションに提供します)。アクションを実装するには、以下の手順に従います。

1. **ConfigTree** インスタンスを提供するアクションクラスへコンストラクターを定義します。
2. **ConfigTree** インスタンスから該当のアクション設定プロパティをすべて取得します。
3. 必須のアクションプロパティを確認して、<action> 設定に指定されていない場所で例外をあげます。
4. プロパティの全値を文字列 (**ConfigTree** で提供) から、アクション実装で使用される適切なタイプにデコーディングします。例えば、**java.lang.String** から **java.io.File** に、**java.lang.String** から **Boolean** に、**java.lang.String** から **long** などにデコーディングします。
5. 設定値が対象のプロパティタイプにデコーディングできない場合、例外を出します。
6. 考えられる各種設定すべてに単体テストを実装して、さきほど表示したタスクが正しく完了するようにします。

上記のタスクは一般的に実行するのは難しくありませんが、作業が増え、エラーが発生しやすく、アクションごとに設定の間違いを処理する方法も異なってきます。結果として、比較すると全体的に明確さが欠けますが、多くのコードを追加する必要がある場合もあります。

アノテーション付きのアクションは、**@ConfigProperty** を使用してこのような問題に対処します。2 種の必須設定プロパティ (logFile and logLevel) を持つ、MyLogActions 実装を拡張します。

```

public class MyLogAction {

    @ConfigProperty
    private File logFile;

    @ConfigProperty
    private LogLevel logLevel;
}

```



```

    public static enum LogLevel {
        DEBUG,
        INFO,
        WARN
    }

    @Process
    public void log(Message message) {
        // log the message at the configured log level...
    }
}

```



注記

(フィールドではなく) "setter" メソッド上で @ConfigProperty アノテーションを定義することも可能です。

必要事項は以上です。Enterprise Service Bus がアクションをデプロイすると、デコーディングされた値にある実装とマップの両方を検証します (enum のサポートと上記の LogLevel enum)。@ConfigProperty アノテーションを処理するアクションフィールドを検索します。開発者は、**ConfigTree** クラスを処理したり、余分のコードを開発したりする必要がありません。

デフォルトでは、@ConfigProperty アノテーションを処理するクラスフィールドはすべて必須です。必須ではないフィールドは、以下の 2 種のいずれかの方法で処理されます。

1. フィールドの @ConfigProperty アノテーションで **use = Use.OPTIONAL** を指定
2. フィールドの @ConfigProperty アノテーションで defaultVal を指定 (任意)

ログアクションのプロパティをオプションのみにするには、以下のようなアクションを実装します。

```

public class MyLogAction {

    @ConfigProperty(defaultVal = "logs/my-log.log")
    private File logFile;

    @ConfigProperty(use = Use.OPTIONAL)
    private LogLevel logLevel;

    public static enum LogLevel {
        DEBUG,
        INFO,
        WARN
    }

    @Process
    public void log(Message message) {
        // log the message...
    }
}

```

@ConfigProperty アノテーションは、2 つの追加フィールドをサポートします。

1. name: これを使用して、アクションインスタンスで指定の名前のフィールドを生成する際に使用するアクション設定プロパティ名を明示的に指定します。
2. choice: このフィールドを使用して、許容の設定値を制限します。これは、列挙型を使用することで可能です (LogLevel)

この名前フィールドは、古いアクション (低／基本レベルの実装タイプを使用) を新しいアノテーションベースの実装に以降する際などに使用できます。プロパティの古い設定名 (後方互換との関係で変更不可) は有効な Java フィールド名へのマッピングは行いません。ログアクションを例にとります。以下がログアクションの古いアクションだと仮定します。

```
<action ...>
  <property name="log-file" value="logs/my-log.log" />
  <property name="log-level" value="DEBUG" />
</action>
```

このプロパティ名は、有効な Java フィールド名にマッピングを行わないため、@ConfigProperty アノテーションで名前を指定します。

```
public class MyLogAction {

    @ConfigProperty(name = "log-file")
    private File logFile;

    @ConfigProperty(name = "log-level")
    private LogLevel logLevel;

    public static enum LogLevel {
        DEBUG,
        INFO,
        WARN
    }

    @Process
    public void log(Message message) {
        // log the message...
    }
}
```

13.4.5. プロパティの値のデコード

Bean 設定のプロパティ値は、文字列の値からデコーディングされます。適切な POJO bean プロパティタイプと一致させるには、以下のシンプルなルールを使用します。

1. プロパティタイプに単一引数の文字列コンストラクターがある場合、これを使用します。
2. Primitive な場合、オブジェクトタイプの単一引数文字列コンストラクターを使用します。例えば、int の場合、整数のオブジェクトを使用します。
3. 列挙型の場合 **Enum.valueOf** を使用して設定の文字列を列挙型の値に変換します。

13.4.6. @Initialize と @Destroy

ときに、アクション実装はデプロイメント時に初期化タスクを実行する必要があります。また、アンデプロイ時にはクリーンアップも行う必要があります。このような理由から、@Initialize および @Destroy メソッドアノテーションがあります。

これについて例を挙げます。デプロイメント時に、ロギングアクションはチェックをいくつか行います (例えばファイルやディレクトリが存在するかなど)。アンデプロイされると、アクションはクリーンアップタスクを実行する必要があります (ファイルを閉じるなど)。以下がこれらのタスクを実行するためのコードです。

```
public class MyLogAction {

    @ConfigProperty
    private File logFile;

    @ConfigProperty
    private LogLevel logLevel;

    public static enum LogLevel {
        DEBUG,
        INFO,
        WARN
    }

    @Initialize
    public void initializeLogger() {
        // Check if file already exists... check if parent folder
        // exists etc...
        // Open the file for writing...
    }

    @Destroy
    public void cleanupLogger() {
        // Close the file...
    }

    @Process
    public void log(Message message) {
        // log the message...
    }
}
```

注記

@ConfigProperty アノテーションはすべて、ESB デプロイヤーが @Initialize メソッドを呼び出す前に処理されます。そのため、@Initialize メソッドは、カスタマイズの初期化を行う前に準備が整ったこれらのフィールドに依存します。

注記

これらのアノテーションを両方使用して、メソッドを指定する必要はありません。必要がある場合のみ指定します。つまり、メソッドが初期化のみ必要な場合、@Initialize アノテーションのみを使用します (@Destroy アノテーションで注釈をつけた matching メソッドを提供する必要はありません)。



注記

単一のメソッドを指定して、@Initialize と @Destroy 両方でアノテーションをつけることができます。



注記

@Initialize メソッドに **ConfigTree** パラメーターをオプションで指定可能です。これを行うと **ConfigTree** インスタンスを基本とするアクションにアクセスできるようになります。

13.4.7. @OnSuccess と @OnException

アクションが設定されている **pipeline** の中で、実行が成功した際、または失敗した際にこれらのメソッドを実行すべきか指定するため、これらのアノテーションを使用します。

```
public class OrderPersister {

    @Process
    public OrderAck storeOrder(Order order) {
        // persist the order and return an ack...
    }

    @OnSuccess
    public void logOrderPersisted(Message message) {
        // log it...
    }

    @OnException
    public void manualRollback(Message message,
                               Throwable theError) {
        // manually rollback...
    }
}
```

これらのアノテーションの場合、パラメーターはパラメーターに渡すかはオプションです。上記のパラメーターの一部、すべてを提供、何も提供しないといった選択ができます。Enterprise Service Bus フレームワークは、それぞれの場合で該当のパラメーターを解決します。

13.5. レガシーアクション

上記のカテゴリに属さないアクションはすべて、レガシーとしてパイプラインに処理されます。これは、以前の Rosetta コードベースの動作を継承します。実際には、以下ようになります。

- **ConfigTree** パラメーターを 1 つとコンストラクターを提供することで、このアクションの設定を行います。
- このパイプラインを経由するすべてのメッセージに対してインスタンス化されます。
- このアクションには、ライフサイクルメソッドの情報はありません。
- **process** メソッドの呼び出しは、常にリフレクションで行われます。

第14章 コネクターおよびアダプター

14.1. はじめに

JBoss Enterprise Service Bus のすべてのクライアントとサービスがネイティブで使用するプロトコルとメッセージ形式を理解できるわけではありません。したがって、ESB 対応エンドポイント (JBossESB を理解するもの) と ESB 対応エンドポイント (JBossESB を理解しないもの) との間を橋渡しする必要があります。このような橋渡しをする技術はさまざまな分散システムで長年使用され、コネクター、ゲートウェイ、またはアダプターと呼ばれることがよくあります。

JBossESB の目的の 1 つは、広範なクライアントとサービスが対話できるようにすることです。JBossESB では、このために JBossESB または任意の ESB を使用してこのようなクライアントとサービスをすべて作成する必要がありません。JBossESB 内には相互運用性のバスの抽象的な表記が存在するため、JBossESB 対応ではない場合があるエンドポイントでもバスに「プラグイン」できます。



注記

これ以降で「ESB 内」または「ESB 内部」という言葉は ESB 対応エンドポイントのことを意味します。

すべての JBossESB 対応クライアントとサービスはメッセージを使用して相互に通信します。メッセージは情報交換の標準的な形式であり、ヘッダー、ボディ (ペイロード)、添付、および他のデータを含みます。また、すべての JBossESB 対応サービスはエンドポイント参照 (EPR) を使用して識別されます。

レガシー相互運用性シナリオの場合に JBossESB などの SOA インフラストラクチャーで ESB 対応クライアントが ESB 対応サービスを使用したり、ESB 対応クライアントが ESB 非対応サービスを使用したりできることが重要です。この相互有用性を実現するために JBossESB が使用するコンセプトはゲートウェイを使用することです。ゲートウェイは ESB 対応の世界と ESB 非対応の世界を橋渡しし、メッセージを EPR に変換したり、逆に EPR をメッセージに変換したりできるサービスです。

JBossESB は現在ゲートウェイとコネクターをサポートします。次のセクションでは、両方のコンセプトを調べ、その使用方法を説明します。

14.2. ゲートウェイ

JBossESB のすべてのユーザーが ESB に対応しているわけではありません。ユーザーが ESB によって提供されるサービスと対話できるように、JBossESB にはゲートウェイ (非 ESB クライアントとサービスからのメッセージを受け取り、必要な宛先にメッセージをルーティングできる特殊なサーバー) のコンセプトが存在します。

ゲートウェイは、ESB 対応リスナーと非常によく似た動作をする特殊なリスナープロセスです。ただし、いくつかの重要な違いがあります。

- ゲートウェイクラスはファイル、JMS メッセージ、SQL テーブルなどに含まれる任意のオブジェクトを取得できる一方で (各「ゲートウェイクラス」は特定のトランスポートに特化されています)、JBossESB リスナーはこのドキュメントの「メッセージ」セクションで説明されたように JBossESB 正規化メッセージのみを処理できます。ただし、これらのメッセージには任意のデータを含めることができます。
- 「メッセージ作成」アクションを実行するには 1 つのアクションクラスのみが呼び出されます。ESB リスナーはアクション処理パイプラインを実行できます。
- 「取得される」オブジェクトは ESB メッセージオブジェクトを返す単一の「compose クラ

ス」(アクション)を呼び出すために使用されます(このオブジェクトは ESB 対応サービスであるターゲットサービスに配信されます)。設定時に定義されたターゲットサービスはレジストリによって実行時に EPR (または EPR の一覧)に変換されます。基礎となるコンセプトはレジストリによって返された EPR は ESB メッセージのヘッダに含まれる 'toEPR' と類似するということですが、受信オブジェクトは「ESB 非対応」であり、toEPR を動的に決定することができないため、この値は設定時にゲートウェイに提供され、すべての送信メッセージに含まれます。

いくつかの既成の compose クラスが存在します。デフォルトの「file」 compose クラスはファイルの内容をメッセージボディにパッケージ化します (JMS メッセージの場合と同様)。SQL テーブル行のデフォルトのメッセージ compose クラスは設定で指定されたすべての列の内容を java.util.Map にパッケージ化します。

ほとんどの場合はこれらのデフォルトの compose クラスで十分ですが、ユーザーが独自のメッセージ compose クラスを提供することは比較的簡単です。唯一の要件は a) 単一の ConfigTree 引数を受け取るコンストラクターを持つこと、b) メッセージ compose メソッド (デフォルト名は 'process' ですが、コンストラクター呼出し時に提供される ConfigTree 内の <action> エレメントの 'process' 属性で異なった名前を設定することができます) を提供することです。処理メソッドはタイプ Object の単一の引数を取り、メッセージ値を返す必要があります。

JBossESB 4.5 以降、**FileGateway** は **file-filter-class** 設定属性を受け入れ、ゲートウェイが使用するファイルを選択する際に使用する FileFilter 実装を定義できるようになります。インスタンスが **org.jboss.soa.esb.listeners.gateway.FileGatewayListener.FileFilterInit** のタイプの場合、ユーザー定義の **FileFilter** インスタンスは、ゲートウェイが初期化します。この場合、**init** メソッドが呼び出され、ゲートウェイ **ConfigTree** インスタンスに渡されます。

デフォルトでは、入力サフィックスが設定ファイルに定義され、サフィックスが返された場合、以下の **FileFilter** 実装が定義され、**FileGateway** により使用されます。あるいは、入力サフィックスがない場合、作業サフィックス、エラーサフィックス、ポストサフィックスと一致しない限り、どのファイルでも受け入れられます。

14.2.1. ゲートウェイデータマッピング

非 JBossESB メッセージはゲートウェイが受け取ったときにメッセージに変換する必要があります。これをどのように行うか、受け取ったデータをメッセージのどこに置くかはゲートウェイの種類によって異なります。この変換をどのように行うかはゲートウェイの種類によって異なります。デフォルトの変換方法は以下で説明されています。

- 入力メッセージが JMS TextMessage である場合は、関連付けられた String がデフォルトで指定されたボディ部分に置かれます。ObjectMessage または BytesMessage である場合、コンテンツは Body 部分で指定された BytesBody.BYTES_LOCATION 内に置かれます。
- ローカルファイルゲートウェイ: コンテンツは BytesBody.BYTES_LOCATION で指定された Body 部分内に置かれます。
- Hibernate ゲートウェイ: コンテンツは ListenerTagNames.HIBERNATE_OBJECT_DATA_TAG で指定されたボディ部分内に置かれます。
- リモートファイルゲートウェイ: コンテンツは BytesBody.BYTES_LOCATION で指定されたボディ部分内に置かれます。



注記

InVM transport の導入により、同じアドレス空間 (VM) 内にサービスをゲートウェイとしてデプロイできるようになり、ゲートウェイとリスナー間の対話が効率的になりました。

14.2.2. ゲートウェイデータマッピングの変更方法

このマッピングがどのように行われるかを変更する場合、その方法はゲートウェイの種類によって異なります。

- **org.jboss.soa.esb.listeners.message.MessageComposer** インターフェースのインスタンスは変換を行います。デフォルトの動作を変更するには、独自の `compose` メソッドと `decompose` メソッドを定義する適切な実装を提供します。新しい **MessageComposer** 実装は `composer-class` 属性名を使用して設定ファイルで提供する必要があります。
- これらの実装は作成クラスを定義するために反射的な方法を使用します。独自のメッセージ composer クラスを提供し、どのインスタンスを使用するかをゲートウェイに通知するために設定ファイルで `composer-class` 属性名を使用します。メッセージが必要な場合はクラスのどの操作を呼び出すかをゲートウェイに通知するために `composer-process` 属性を使用できます。このメソッドはオブジェクトを受け取り、メッセージを返す必要があります。指定されていない場合は、プロセスのデフォルト名が使用されます。



注記

メッセージ作成を再定義するのにどのメソッドを使用するかに関係なく、ボディだけでなくメッセージのコンテンツを完全に制御することに注意してください。たとえば、元のコンテンツの内容や送信者などに基づいて新しく作成されたメッセージに対して `ReplyTo` または `FaultTo` EPR を定義する場合は、ヘッダーも変更することを考慮する必要があります。

14.3. JCA を介した接続

JCA メッセージインフローを ESB ゲートウェイとして使用できます。この統合では MDB ではなく ESB の軽量インフロー統合を使用します。サービスに対してゲートウェイを有効にするには、最初にエンドポイントクラスを実装する必要があります。このクラスは **org.jboss.soa.esb.listeners.jca.InflowGateway** クラスを実装する必要がある Java クラスです。

```
public interface InflowGateway
{
    public void setServiceInvoker(ServiceInvoker invoker);
}
```

エンドポイントクラスはデフォルトのコンストラクターまたは `ConfigTree` パラメーターを受け取るコンストラクターを持つ必要があります。この Java クラスはバインドする JCA アダプターのメッセージタイプも実装する必要があります。以下は JMS アダプターにフックする単純なエンドポイントクラスの例です。

```
public class JmsEndpoint implements InflowGateway, MessageListener
{
    private ServiceInvoker service;
    private PackageJmsMessageContents transformer = new
```



```

PackageJmsMessageContents();

    public void setServiceInvoker(ServiceInvoker invoker)
    {
        this.service = invoker;
    }

    public void onMessage(Message message)
    {
        try
        {
            org.jboss.soa.esb.message.Message esbMessage =
transformer.process(message);
            service.deliverAsync(esbMessage);
        }
        catch (Exception e)
        {
            throw new RuntimeException(e);
        }
    }
}

```

このクラスに対して定義されたゲートウェイ 1 つに対して `JmsEndpoint` クラスの 1 つのインスタンスが作成されます。これはプールされる MDB とは異なります。このクラスの 1 つのインスタンスのみが各受信メッセージにサービスを提供するため、スレッドセーフコードを記述する必要があります。

設定時に ESB は **ServiceInvoker** を作成し、エンドポイントクラスの **setServiceInvoker** メソッドを呼び出します。次に ESB は JCA エンドポイントを有効にし、エンドポイントクラスインスタンスがメッセージを受信できる状態になります。`JmsEndpoint` の例では、インスタンスが JMS メッセージを受け取り ESB メッセージタイプに変換し、ターゲットサービスで呼び出す `ServiceInvoker` インスタンスを使用します。



注記

JMS エンドポイントクラスは、ESB の配布が **org.jboss.soa.esb.listeners.jca.JmsEndpoint** 下に存在する状態で提供されます。このクラスを任意の JMS JCA インフローアダプタとともに何度も使用することができます。

14.3.1. 構成

JCA インフローゲートウェイは **jboss-esb.xml** ファイルに設定されます。以下に例を示します。

```

<service category="HelloWorld_ActionESB"
    name="SimpleListener"
    description="Hello World">
    <listeners>
        <jca-gateway name="JMS-JCA-Gateway"
            adapter="jms-ra.rar"
            endpointClass="org.jboss.soa.esb.listeners.jca.JmsEndpoint">
        <activation-config>
            <property name="destinationType" value="javax.jms.Queue"/>
            <property name="destination" value="queue/esb_gateway_channel"/>
        </activation-config>
    </listeners>
    </service>

```



```

    </jca-gateway>
    ...
</service>

<service category="HelloWorld_ActionESB"
  name="SimpleListener"
  description="Hello World">
  <listeners>
    <jca-gateway name="JMS-JCA-Gateway"
      adapter="jms-ra.rar"
      endpointClass="org.jboss.soa.esb.listeners.jca.JmsEndpoint">
      <activation-config>
        <property name="destinationType" value="javax.jms.Queue"/>
        <property name="destination" value="queue/esb_gateway_channel"/>
      </activation-config>
    </jca-gateway>
  ...
</service>

```

JCA ゲートウェイは <jca-gateway> エlement に定義されます。JCA ゲートウェイはこの XML Element の設定可能な属性です。

表14.1 jca-gateway 設定属性

属性	必須	説明
name	yes	ゲートウェイの名前
アダプタ	yes	使用しているアダプタの名前。JBoss では、デプロイした RAR のファイル名 (jms-ra.rar など) になります。
endpointClass	yes	エンドポイントクラスの名前
messagingType	no	アダプタのメッセージインターフェース。指定しない場合は、ESB がエンドポイントクラスに基づいた名前を付けます。
実行済み	no	デフォルトでは真に設定されます。JTA トランザクション内でメッセージを呼び出すかどうかを設定します。

<activation-config> Element は <jca-gateway> 内に定義する必要があります。この Element はアクションプロパティと同じ構文を持つ 1 つまたは複数の <property> Element を受け取ります。エンドポイントクラスにメッセージを送信するには <activation-config> 下のプロパティが使用されます。これは、JCA を MDB とともに使用する場合と変わりません。

<jca-gateway> 内には <property> Element を好きなだけ持つこともできます。このオプションは、追加の設定をエンドポイントクラスに渡すために提供されています。これらはコンストラクタに渡された ConfigTree によって読み取ることができます。

14.3.2. 標準のアクティベーションプロパティのマッピング

ESB プロパティの数は自動的に ActivationMapper を使用してアクティベーション設定にマッピングされます。このプロパティ、ロケーション、目的については以下の表にまとめられています。

表14.2 標準アクティベーションプロパティのマッピング

属性	Location	説明
maxThreads	jms-listener	同時に処理できる最大メッセージ数
dest-name	jms-message-filter	JMS 宛先名
dest-type	jms-message-filter	JMS 宛先タイプ、QUEUE または TOPIC
selector	jms-message-filter	JMS メッセージセクター
providerAdapterJNDI	jms-jca-provider	プロバイダーアダプターの JNDI ロケーションは、JCA インフローが使用してリモートの JMS プロバイダーにアクセスできます。これは、JBoss 固有のインターフェースで、デフォルトの JCA インフローアダプターによりサポートされていますが、必要であれば、他のインフローアダプターによる使用も可能です。

アクティベーション仕様へのプロパティのマッピングは、ActivationMapper インターフェースを実装することで、オーバーライドできます。また、グローバルまたは各 ESB デプロイメント設定内で宣言可能です。

ActivationMapper は、**jbossesb-properties.xml** ファイルを使うことでグローバルに指定されます。指定の JCA アダプターに使用するデフォルトのマッパーを定義します。設定するプロパティ名は **org.jboss.soa.esb.jca.activation.mapper."adapter name"** で、値は ActivationMapper のクラス名です。

以下のスニペットでは、JBoss JCA アダプター (**jms-ra.rar**) のアクティベーション仕様でプロパティをマッピングする際に使用するデフォルトの ActivationMapper の設定を示しています。

```
<properties name="jca">
  <property name="org.jboss.soa.esb.jca.activation.mapper.jms-ra.rar"
value="org.jboss.soa.esb.listeners.jca.JBossActivationMapper"/>
</properties>
```

デプロイメント内で ActivationMapper を指定すると、グローバル設定をオーバーライドします。マッパーをリスナー、バス、プロバイダー内で優先順位を保ちつつ指定可能です。

次のコード例は、プロバイダー設定内でマッパー設定を指定する例を示します。

```
<jms-listener name="listener" busidref="bus" maxThreads="100">
  <property name="jcaActivationMapper"
value="TestActivationMapper"/>
</jms-listener>
```

次のコード例は、プロバイダー設定内でマッパー設定を指定する例を示します。

-


```
<jms-bus busid="bus">
  <property name="jcaActivationMapper"
value="TestActivationMapper"/>
  <jms-message-filter dest-type="TOPIC" dest-name="DestName"/>
</jms-bus>
```

次のコード例は、プロバイダー設定内でマッパー設定を指定する例を示します。

```
<jms-jca-provider name="provider" connection-factory="ConnectionFactory">
  <property name="jcaActivationMapper"
value="TestActivationMapper"/>
  <jms-bus busid="bus">
    <jms-message-filter dest-type="TOPIC" dest-name="DestName"/>
  </jms-bus>
</jms-jca-provider>
```


付録A 付録 A: JAXB ANNOTATION INTRODUCTION 設定の記述

JAXB (Java Architecture for XML Binding) Annotation Introduction の設定を作成するのは非常に簡単です。AXB Annotation を理解できていれば、問題なく JAXB Annotation Introduction の設定を記述することができるでしょう。

設定の XSD

は、<http://anonsvn.jboss.org/repos/jboss/ws/projects/jaxbintros/tags/1.0.0.GA/src/main/resources/jaxb-intros.xsd> にてオンラインで提供されています。IDE にて <http://www.jboss.org/xsd/jaxb/intros> 名前空間に対してこの XSD を登録します。

現在サポートされているのは 3 つのアノテーションのみです。

1. @XmlType
(<https://jaxb.dev.java.net/nonav/2.1.3/docs/api/javax/xml/bind/annotation/XmlType.html>): “Class” 要素
2. @XmlElement
(<https://jaxb.dev.java.net/nonav/2.1.3/docs/api/javax/xml/bind/annotation/XmlElement.html>): “Field” と “Method” 要素
3. @XmlAttribute
(<https://jaxb.dev.java.net/nonav/2.1.3/docs/api/javax/xml/bind/annotation/XmlAttribute.html>): “Field” と “Method” 要素

設定ファイルの基本構成は、“Fields” と “Methods” が “Class” に含まれるというように Java クラスの基本構造に準拠しています。<Class>、<Field>、<Method> 要素はすべて “name” 属性を必要とします。この属性は、Class、Field、Method の名前を提供しています。この name 属性の値は正規表現に対応可能です。これにより、1 つの Class 内のフィールドや 1 パッケージ内のすべての Class に名前空間を設定するなど、単一の Annotation Introduction 設定で複数の Class、Field、Member を対象にすることができます。

Annotation Introduction 設定は Annotation の定義と完全に一致し、各アノテーションの要素と値のペアは、Annotation Introduction 設定上の属性によって表されます (XSD と IDE を使用して設定を編集してください)。

最後に例を挙げます。

```
<?xml version = "1.0" encoding = "UTF-8"?>
<jaxb-intros xmlns="http://www.jboss.org/xsd/jaxb/intros">

  <!--
    The type namespaces on the customerOrder are
    different from the rest of the message...
  -->

  <Class name="com.activebpel.ordermanagement.CustomerOrder">
    <XmlType propOrder="orderDate,name,address,items" />
    <Field name="orderDate">
      <XmlAttribute name="date" required="true" />
    </Field>
    <Method name="getXYZ">
      <XmlElement

namespace="http://org.jboss.esb.quickstarts/bpel/ABI_OrderManager"
nillable="true" />
```



```
        </Method>
    </Class>

    <!-- More general namespace config for the rest of the message... -->
    <Class name="com.activebpel.ordermanagement.*">
        <Method name="get.*">
            <XmlElement namespace="http://ordermanagement.activebpel.com/jaws"
/>
        </Method>
    </Class>

</jaxb-intros>
```


付録B 添付資料 B: サービス指向アーキテクチャーの概要

JBossESB はサービス指向アーキテクチャー (SOA) インフラストラクチャーです。SOA はアプリケーション開発で人気があるアーキテクチャーパラダイムです。SOA の原則は長年存在し、Web サービスの使用を必要としませんでした。SOA は Web サービスの使用によって広まりました。

Web サービスは業界標準のネットワークおよびアプリケーションのインターフェースとプロトコルを使用して他のアプリケーション (または他の Web サービス) を利用できる機能を実装します。SOA はソフトウェアコンポーネントが他のソフトウェアコンポーネントによって活用できるサービスとして機能を提供する手法に準拠します。コンポーネント (またはサービス) は再利用可能なソフトウェアのビルディングブロックを表します。

企業の意識がコスト削減にのみ集中する状態からようやくより安定的なコスト管理に移ってきたなかで、多くの企業は今まで経験したことがない状況に直面しています。現在の経済不況以前は、ほとんどの企業が IT 投資の選択肢を理解していました。多くの企業が主要なパッケージ実装 (Siebel、PeopleSoft など) を導入する一方で、他の企業は長年使用してきたレガシーシステムを元にしてシステムを構築していました。どちらの方法でもほとんどの企業が約束されたリターンを認識し、投資を行っていました。今現在はこのような大規模な投資への意欲は失われました。

ただし、企業は依然として前進し、競争に勝つ必要があります。SOA (およびこれらの原則の具体的な実装としての Web サービス) はこれを可能にします。結果として、ユーザー、アプリケーション、およびテクノロジーコンポーネント間のコラボレーションが大幅に改善され、どのようなビジネスにも大きな価値が生み出され、競争力が強化されます。

SAP、PeopleSoft などさまざまなベンダーのソフトウェアを使用している企業を考えてみてください。他社 (顧客、供給業者など) とビジネスを行うにはこれらのいくつかのソフトウェアパッケージが役に立つかもしれませんが、したがって、企業はこれらの既存のシステムをサービスとして公開することにより他社が利用できるようにします。このサービスはクライアント (他のソフトウェアコンポーネント) が呼び出すことができる安定した公開インターフェースを持つソフトウェアコンポーネントです。したがって、サービスを要求および実行するには、ある企業が所有するソフトウェアコンポーネントが他の企業が所有するコンポーネントと対話する必要があります (ビジネスツービジネス (B2B) トランザクション)。

このような組織間のやり取りに対して従来の分散システムインフラストラクチャー (ミドルウェア) は十分ではありません。

- ミドルウェアプラットフォームに関係する者の間で同意が必要になります。
- これらの関係者間には暗黙的な (場合によっては明示的な) 信頼の欠如が存在します。
- ビジネスデータは機密性が高く、想定された受信者のみによって参照されるべきです。
- 組織間の対話処理では従来のミドルウェアで前提としていたことの多くが無効です。たとえば、トランザクションは長時間 (数時間または数日) 継続するため、2 フェーズコミットなどの従来のトランザクションプロトコルは適用できません。

したがって、B2B 交換では、複数のミドルウェアプラットフォーム間での標準化が存在しないため、ポイントツーポイントを実現するのにコストがかかります。インターネットにより標準的な対話プロトコル (HTTP) とデータ形式 (XML) が提供されこれらのいくつかの問題は緩和されましたが、インターネットのこれらの標準自体はアプリケーション統合をサポートするのに十分ではなく、インターフェース定義言語、ネームおよびディレクトリサービス、トランザクションプロトコルなどを定義しません。Web サービスは Web が提供するものとアプリケーション統合が必要とするものの違いを解消しようとしません。

ただし、SOA の最終的な目標は企業間の対話処理であり、インターネットを使用してサービスにアクセスする必要はありません。サービスはローカルネットワークに存在するクライアントが簡単に利用できます。1 つの企業内で稼働する複数のシステムを統合するために Web サービスがこのように使用さ

れることは一般的です。

企業内および企業間で Web サービスがどのようにアプリケーション同士を接続するかを理解するために、スタンドアロンの在庫システムを考えてみてください。このシステムを他のシステムに接続しない場合、その本来の価値は限定されます。システムは在庫を追跡できますが、それ以外のことはできません。在庫情報は会計および顧客関係管理システムに別々に入力しなければなりません。在庫システムは自動的に供給業者に発注できません。このような在庫システムの利点は高いオーバーヘッドコストによって少なくなります。

しかし、在庫システムを XML を使用して会計システムに接続すると、興味深い結果が得られます。この場合、何かを購入または販売するたびに在庫の結果とキャッシュフローを 1 つのステップで追跡できるようになります。さらに倉庫管理システム、供給業者の発注システム、運送業者を XML を使用して接続するとすぐに、在庫管理システムは多数の価値を生み出すようになります。この場合、ビジネスのエンドツーエンド管理を行うことができ、影響を受ける各システムに対してではなく各トランザクションを 1 度だけ処理するだけですみます。作業が大幅に削減され、エラーの発生も大幅に減少します。これらの接続は Web サービスを使用して簡単に行えます。

ビジネスは以下のものを含む SOA の利点を理解しはじめています。

- パートナと簡単に接続することにより、新しいビジネスチャンスが生まれます。
- ソフトウェア開発の時間を削減し、他の企業によって作成されたサービスを使用することにより時間とお金を削減します。
- 独自のサービスを簡単に利用できるようにすることにより収益ストリームが増加します。

B.1. SOA とは

問題は一般的に過去の IT 投資の eProcurement、eSourcing、サプライチェーン管理、顧客関係管理 (CRM)、インターネットコンピューティングの領域に分類できます。これらのすべての投資は 1 つのサイロで行われました。短期の (戦術的な) 要件を満たすためにこれらシステムが増大し、これらの領域に対して下された決定によりアプリケーションとインフラストラクチャーの長期的な実現できなくなりました。

SOA 手法を実装する 3 つの主要な理由は以下のとおりです。

1. コスト削減: サービス同士が対話する方法によって達成されます。直接的なコスト効果は、向上した処理の生産性、効果的なソースオプション、現在のコストを可変モデルにシフトする大幅に拡張された機能をによってもたらされます。
2. 標準に基づいた手法をとることにより、組織は以前よりも非常に迅速かつ簡単に情報/ビジネスプロセスを接続および共有できるようになります。標準的なフレームワークとインターフェースを提供することにより開発者の役割が単純化され IT 導入の生産性が大幅に向上します。個別機能の統合負荷を緩和し、環境内で高速な導入テクニックを適用することにより導入時間は大幅に短縮されます。
3. Web サービスを使用すると、新しいビジネスモデルを実現でき新しいビジネスチャンスが生み出されます。Web サービスは従来の機能と利益に基づく方法とは大きく異なり価値と離散的なリターンを測定できる機能を提供します。通常の TCO (総所有コスト) モデルでは過去の投資から生まれた生涯の価値が考慮されません。コスト中心のこの観点により、これらの過去の投資を活用する多くのチャンスが失われ、結果的にほとんどの企業が必要からではなく予見されたニーズのためにアーキテクチャーで冗長性を構築することになってしまいます。これらの同じ組織はインフラストラクチャーのオーバーヘッドによって調整されたアプリケーションのポートフォリオに IT 投資の価値を見出します。Web サービスに基づいた方法では、過去の IT 投資の生涯貢献度が考慮され、計画的なシステムの置き換えではなくこれらの投資の発展が促進されます。

SOA/Web サービスはエンタープライズソフトウェアの開発およびデプロイ方法を根本的に変えます。SOA は進化し、新しいアプリケーションはモノリシックな方法で開発されず、従来の方法により引き起こされた現在の経済的および技術的なボトルネックを解消する仮想オンデマンド実行モデルになります。

作業を効率化し、所有コストを削減し、市場で競争力を持ち、差異化を図りたい先進的な企業にとってサービスとしてのソフトウェアは一般的なモデルです。Web サービスを使用すると、企業はソフトウェアの取得から大きな利益を生み出し、急速に変化する市場に対応し、ビジネスパートナーといつでも取引を行えるようになります。疎結合の規格をベースにしたアーキテクチャーはネットワークで利用可能なソフトウェアリソースを活用できる分散コンピューティングの1つの方法です。ビジネスプロセス、プレゼンテーションルール、ビジネスルール、データアクセスを独立した疎結合レイヤに分離することは、より良いソフトウェアを構築するだけでなく今後起こる変更に対応するためにも役に立ちます。

SOA により、既存の機能と新しい開発成果を組み合わせることで複合アプリケーションを作成できます。このように既存の機能を再利用することにより、プロジェクトのリスクの低下、導入時間の短縮、ソフトウェアの全体の品質の向上が実現されます。

疎結合により、モノリシックな方法を使用したコストがかかる移行に関連するリスクを負わずに部品を独自のペースで変更できるようになります。SOA により、ビジネスユーザーは技術的な制約を気にせずに現在直面しているビジネス上の問題に集中できます。ソリューションを開発する個人にとって SOA は以下の点で役に立ちます。

- ビジネスアナリストはビジネスドメインの知識を増やしつつ開発ライフサイクルの高位の仕事に集中できます。
- 複数のチームが取り組むことができるコンポーネントベースのサービスに機能を分けることによって並列開発が可能になります。
- 品質保証や単体テストが効率的になります。エラーは開発ライフサイクルの初期に検出できます。
- 開発チームはリスクを増やさずに初期の要件から内容を変更することもできます。
- アーキテクチャー内のコンポーネントは再利用可能な資産でありその部品を再び作成する必要はありません。
- サービスの機能分解やビジネスプロセス関連の基盤コンポーネントで、柔軟性、今後の管理や統合のしやすさを保ちます。
- セキュリティルールはサービスレベルで実装されるため、企業内の多くのセキュリティ懸念事項を解決できます。

B.2. SOA の基本

従来の分散コンピューティング環境は密接に結合され、変化する環境に十分に対応できません。たとえば、アプリケーションが他のアプリケーションと対話する場合にいずれかのシステムのデータ型が変更されると両方のアプリケーションがどのようにデータ型やデータエンコーディングを処理するかが問題となります。互換性がないデータ型はどのように処理されるのでしょうか？

サービス指向アーキテクチャー (SOA) はリクエスター、プロバイダー、ブローカーの3つの役割から構成されます。

サービスプロバイダ

サービスプロバイダはサービスへのアクセスを許可し、サービスの説明を作成し、サービスブローカーに公開します。

サービスリクエスター

サービスリクエスターはサービスブローカーによって提供されたサービスの説明を検索することによってサービスを探します。また、リクエストはサービスプロバイダーによって提供されるサービスにバインドします。

サービスブローカ

サービスブローカーはサービスの説明のレジストリをホストし、リクエスターをサービスプロバイダに関連付けます。

B.3. SOA の利点

SOA は分散エンタープライズシステムに複数の大きな利点を提供します。最も注目すべき利点には相互有用性、効率性、標準化などが含まれます。これらについてはこのセクションで簡単に説明します。

B.3.1. 相互運用性

相互運用性は、データと機能を共有することによってさまざまなシステムのソフトウェア同士の通信を可能にします。SOA と Web サービスは Web とインターネットスケールコンピューティングと同様に相互運用性に大きく基づきます。ほとんどの企業はその存続期間において数多くのビジネスパートナーと取引を行います。SOAP などの Web サービステクノロジーを使用すると、新しいパートナーを得るたびに 1 つのインターフェースを作成するだけですみます。したがって、パートナーは UDDI を使用してサービスを動的に見つけ、SOAP を使用してバインドできます。また、自社のネットワーク内に Web サービスを導入することによりシステムの相互運用性を拡張することもできます。システムに Web サービスを追加すると、統合コストを削減し、コミュニケーションと顧客ベースを増大させることができます。

重要なのは業界が Web Services Interoperability Organization を創設したことです。

「Web Services Interoperability Organization はプラットフォーム、アプリケーション、およびプログラミング言語間の Web サービスの相互運用性を促進するために設立されたオープンな組織です。この組織は相互運用可能な Web サービスを開発するためのガイダンスや推奨される慣習を提供したり、リソースを提供したりすることにより多様な Web サービスのリーダーが顧客のニーズに応えることができるよう支援します。」 (www.ws-i.org)

WS-I は実際には Web サービスが業界標準と同様に WS-I 標準に準拠するかどうかを判断します。整合性や信頼性を確立するために企業は WS-I 標準に準拠して Web サービスを構築しようとします。

B.3.2. 効率性

SOA を使用すると、既存のアプリケーションを再利用できます。完全に新しいアプリケーションを作成する代わりに、既存のアプリケーションによって公開されたさまざまな組み合わせのサービスを使用してアプリケーションを作成できます。開発者は業界標準のテクノロジーを学ぶことに集中できるため作業は効率的になり、新しいテクノロジーが現れるたびに時間をかけて学ぶ必要がありません。マネージャにとってこれは新しいソフトウェアを購入し、新しいスキルを持った開発者を雇用するコストが削減されることを意味します。この手法により、開発者は変化するビジネス要件に対応し、プロジェクトの開発サイクル期間を短縮できるようになります。一般的に SOA を使用するとアプリケーションの再利用、開発者の学習時間の短縮、全体的な開発プロセスの短縮が可能になり効率が向上します。

B.3.3. 標準化

標準化を実際に図るには、その標準が業界の多数によって認められ使用される必要があります。1 つのベンダやベンダの小規模なグループがテクノロジーや仕様の進化を支配することがあってはなりません。業界リーダーの全員とは言わないまでもそのほとんどが Web サービス仕様の開発に参加します。ほと

んどの企業は何らかの形でインターネットと World Wide Web を使用します。WWW の基礎となるプロトコルは当然 HTTP です。Web サービスの基礎は HTTP と XML に基づいて構築されます。SOA では特定の実装フレームワークを必要としませんが、相互運用性が重要であり、SOAP は優れたすべての SOA 実装がともに使用する複数のプロトコルの 1 つです。

B.3.4. ステートフルおよびステートレスサービス

Web サービスのほとんどの提唱者は、アーキテクチャが Web のようにスケーラブルかつ柔軟であることが重要であると認識しています。結果として、Web サービスの現在の対話パターンは粒度が粗いサービスやコンポーネントに基づきます。アーキテクチャは意図的にサービスエンドポイントの背後で起こることについて規定していません。Web サービスは究極的には関係者間の構造化データの転送とこのような転送を(メッセージの暗号化やデジタル署名などによって)保護するメタレベルの情報にのみ関係します。これにより、実装の柔軟性が提供され、ユーザーに直接影響を与えずにシステムが要件やテクノロジーの変更に適応できるようになります。また、ほとんどの企業はさまざまな理由からバックエンド実装の決定や方針をユーザーに公開したくありません。

CORBA、J2EE、DCOM などの分散システムでは、通常対話はコンテナ内に存在するステートフルオブジェクト間で行われます。これらのアーキテクチャーでは、オブジェクトは個別に参照されるエンティティとして公開され、特定のコンテナ、したがって多くの場合は特定のマシンに関連付けられます。ほとんどの Web サービスアプリケーションはオブジェクト指向言語を使用して作成されているため、そのアーキテクチャを Web サービスに拡張することを考えるのは自然です。したがって、サービスは、特定の状態を表す Web サービスリソースを公開します。結果として、このようなアーキテクチャーによりクライアントとサービスの結合は密接になり、World Wide Web と同等のスケーラビリティを実現することは難しくなります。

現在、Web サービス定義に参加する企業によって定義されるセッションコンセプトには、WS-Addressing EndpointReferences (ReferenceProperties/ReferenceParameters とともに使用) と WS-Context の明示的なコンテキスト構造の 2 つの主要なモデルが存在します。これら両方のモデルは JBossESB 内でサポートされます。WS-Addressing セッションモデルは Web サービスエンドポイント情報とセッションデータ間の結合を提供し、分散オブジェクトシステムのオブジェクト参照に類似します。

WS-Context は HTTP サーバー、トランザクション、MOM システムで存在するセッションモデルが進化したセッションを提供します。その一方で、WS-Context を使用するとサービスクライアントがサービスとの関係を動的および一時的にバインドできるようになります。サービスに対するクライアントの通信チャネルは特定のセッション関係によって影響されません。

これは、Web サービスを内部デプロイメントからインターネットで提供されている一般的なサービスにスケーリングするときに重要になります。Web サービスの現在の対話パターンは粒度が粗いサービスまたはコンポーネントに基づいています。アーキテクチャは意図的にサービスエンドポイントの背後で起こることについて規定していません。Web サービスは究極的には関係者間の構造化データの転送とこのような転送を(メッセージの暗号化やデジタル署名などによって)保護するメタレベルの情報にのみ関係します。これにより、実装の柔軟性が提供され、ユーザーに直接影響を与えずにシステムが要件やテクノロジーの変更に適応できるようになります。また、サービスがユーザーや(一時的にバインドされる)その対話を代表してステータスを保持するかどうかなどの問題は実装によって異なり、通常はユーザーに公開されません。

ステートフルサービスと対話するときに WS-Addressing に基づくセッション形式のモデルを使用する場合は、ステータスとサービス間の密接な結合によりクライアントが影響を受けます。このモデルが使用されている他の分散環境 (CORBA や J2EE など) と同様に、クライアントがサービスエンドポイントに対して持つリモート参照 (アドレス) は以降の起動のためにクライアントによって記憶されなければなりません。クライアントアプリケーションが同じ論理セッション内の複数のサービスと対話する場合、サービスは他のサービスの関連付けられたステータスとともに使用する場合のみクライアントに影響します。つまり、クライアントは各サービス参照を記憶し、特定の対話と何らかの方法で関連付ける必要があります。したがって、複数の対話の結果として、さまざまな参照セットを組み合わせることで各セッションを表すことができるようになります。

たとえば、同じアプリケーションセッション内で N 個のサービスが使用され、各サービスが m 個の異なるステータスを保持する場合、クライアントアプリケーションは $N*m$ 個の参照エンドポイントを保持する必要があります。初期サービスエンドポイント参照が UDDI など一部のブートストラッププロセスから取得されることがよくあることに注意してください。ただし、このモデルではこれらの参照はステートレスであり、アプリケーション対話を開始する以外は役に立ちません。これ以降に特定のステータスへのアクセスが必要なこれらのサイトにアクセスする場合は、WS-Addressing モデルで異なる参照を使用する必要があります。

これにより、当然 Web のサイズ的环境にスケーリングされませんが、代わりに WS-Context を使用して Web サービスの疎結合の性質を引き続き受け継ぐことができます。これまでに示したように、一連のサービスとの各対話はセッションとしてモデル化でき、セッションは関連付けられたコンテキストを持つ WS-Context アクティビティとしてモデル化できます。クライアントアプリケーションが同じセッション内の一連のサービスと対話するときは、コンテキストがサービスに必ず伝播され、このコンテキストはクライアントとの対話に必要なステータスにマッピングされます。

このマッピングがどのように行われるかは実装によって異なり、クライアントに公開する必要はありません。また、特定のセッション内の各サービスは同じコンテキストを取得し、これらのサービスに後で再びアクセスしたときに同じコンテキストが再び提供されるため、クライアントアプリケーションは必ず正しいステータスセットに戻ることができます。したがって、前述した例の N 個のサービスと m 個のステータスの場合、クライアントは N 個のエンドポイント参照のみを必要とし、これまでに述べたように通常これらのエンドポイント参照はブートストラッププロセスから取得されます。この結果、このモデルのスケーリングは大幅に向上します。

付録C 参考資料

JBoss Enterprise SOA Platform チームは、SOA アプリケーションの構築の指針として、以下の参考資料を推奨しています。

推奨資料

Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Gregor Hohpe 、 Bobby Woolf. 0321200683. Addison-Wesley Professional .

Enterprise Service Oriented Architectures. James McGovern, Oliver Sims, Ashish Jain, 、 Mark Little. 140203704X. Springer .

Open Source SOA. Jeff Davis. 1933988541. Manning Publications Co .

SOA Design Patterns. Thomas Erl. 0136135161. Prentice Hall PTR .

付録D GNU GENERAL PUBLIC LICENSE 2.0

GNU GENERAL PUBLIC LICENSE Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.,
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
- b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
- c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but

does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange;
or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that

system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED

TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along

with this program; if not, write to the Free Software Foundation, Inc.,

51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author

Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.

This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program
`Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into
proprietary programs. If your program is a subroutine library, you may
consider it more useful to permit linking proprietary applications with
the
library. If this is what you want to do, use the GNU Lesser General
Public License instead of this License.

付録E 改訂履歴

改訂 5.2.0-0.1.400 Rebuild with publican 4.0.0	2013-10-31	Rüdiger Landmann
改訂 5.2.0-0.1 翻訳ファイルとXML ソース 5.2.0-0を同期	Tue Feb 5 2013	Credit
改訂 5.2.0-0 SOA 5.2 向けに更新	Tue Jun 14 2011	David Le Sage
改訂 5.1.0-0 SOA 5.1 向けに更新 コミュニティ文書の再構築に合わせて本書も大幅に整理 セクション 4.1.4 注釈付きアクションクラスの追加 セクション 9.10 Camel Gateway の追加 セクション 12.5.3 MappedHeaderList の追加	Mon Mar 7 2011	David Le Sage
改訂 5.0.2-0 SOA-2072 - さらに詳細にまとめたプロトコルのサンプル。セクション 11.7.7 SOA 5.0.2 向けに更新	Fri Jun 11 2010	David Le Sage
改訂 5.0.1-0 SOA 5.0.1 向けに更新	Tue Apr 20 2010	David Le Sage
改訂 5.0.0-0 Created	Tue Feb 23 2010	David Le Sage