Red Hat JBoss Operations Network 3.3

Development - Writing Custom Plug-ins

カスタムサーバーおよびエージェントリソースプラグインを作成するためのガイドライン

Last Updated: 2020-08-31

Red Hat JBoss Operations Network 3.3 Development - Writing Custom Plug-ins

カスタムサーバーおよびエージェントリソースプラグインを作成するためのガイドライン

Jared Morgan
jmorgan@redhat.com

Zach Rhoads
zach@redhat.com

Ella Deon Ballard
dlackey@redhat.com

法律上の通知

概要

本ガイドを参照して、サーバーとそのエージェントにリソースプラグインを書き込む方法を説明します。

目次

第1章 JBOSS ON プラグインの概要

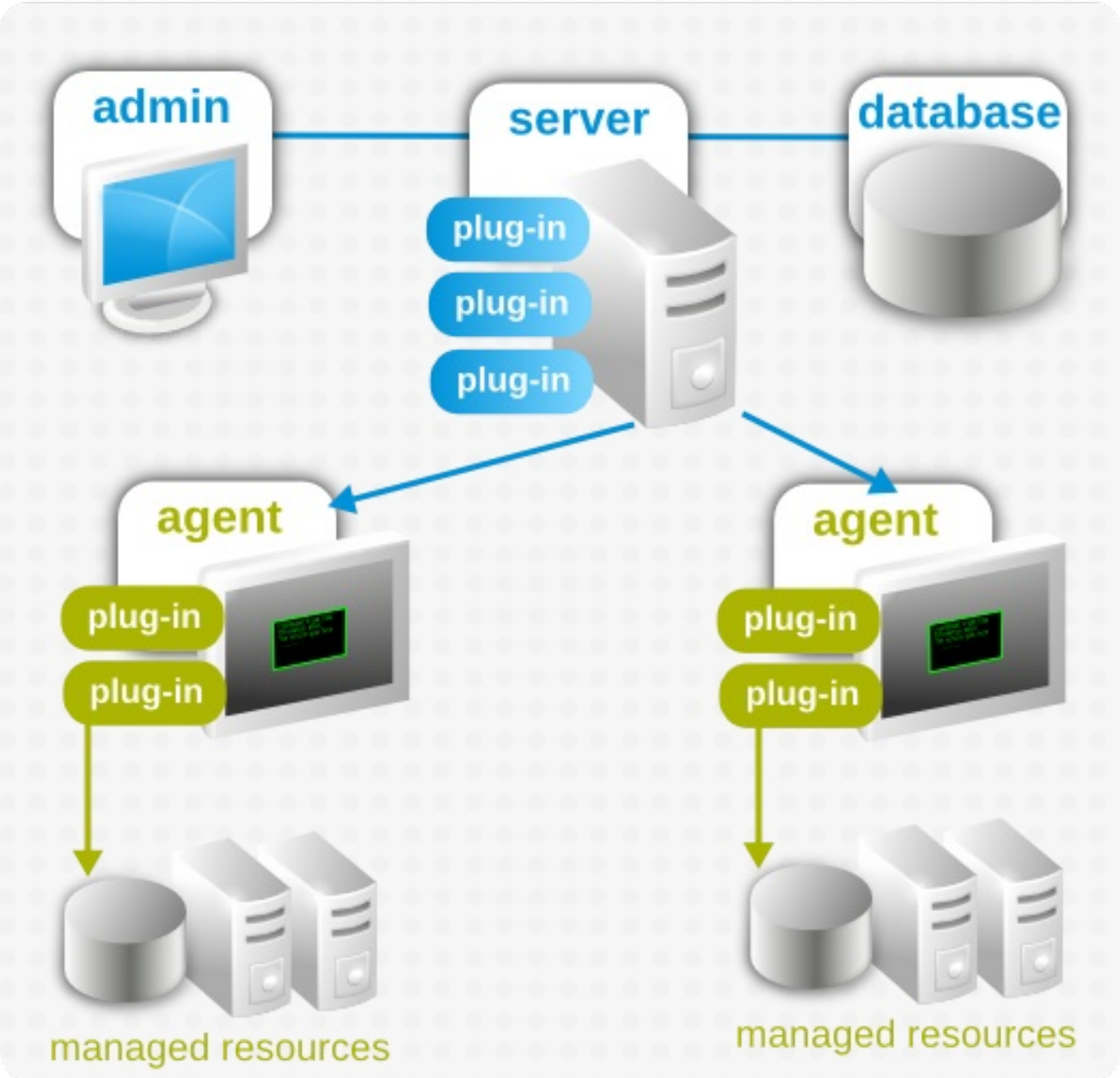
プラグインは、特定の方法でアプリケーションをより有用なものにします。これは、既存の機能に新しい機能またはオプションを提供する方法です。JBoss ON では、サーバー側のプラグインとエージェントプラグインなど、作成する必要のある機能に応じて 2 つのカテゴリーのプラグインがあります。JBoss ON には、新しいプラグインをデプロイするための非常にシンプルで密接に統合されたフレームワークがあるため、特定のカスタムタスクを実施できるように JBoss ON の拡張が比較的簡単です。JBoss ON のサブシステムまたは機能は、追加のプラグインを作成して拡張およびカスタマイズできます。本ガイドは、JBoss ON でプラグインを作成および実装する方法を紹介します。

バグを報告します。

1.1. JBOSS ON の拡張： プラグインの定義

JBoss ON は、サーバーとのハブおよびスポークアプローチに従います。エージェントはリソースにローカルでデプロイされ、リソースおよび JBoss ON サーバーと対話します。サーバー（またはサーバーのクラスター）は、エージェントから送信されるデータを処理します。データは、サーバーに接続されているデータベースに保存されます。ユーザーは、サーバー上の Web ベースの GUI を使用してデータを確認し、トリガー操作を行うことができます。

図1.1 JBoss ON のアーキテクチャー



プラグインは、使用する機能を定義してから、定義した機能または操作を実行するのに必要なコード（または API）が含まれます。JBoss ON では、プラグインはサーバーまたはエージェントのいずれかで動作することが意図されています。

サーバー側のプラグイン は、サーバーによって実行される操作またはタスクに関連するものです。これには、アラートおよび通知、コンテンツおよびパッケージの管理、GUI 外観および機能の設定、JBoss ON 情報の他のアプリケーションとの統合が含まれます。サーバー側のプラグインは、最初に関連付けられた server サブシステムによって識別され、その後の機能で識別されます。

エージェントプラグインは、主に リソースに関連するタスク、主にインベントリーを管理（リソースタイプの定義により）、監視を設定するために使用されます。その後、gent プラグインはリソースタイプによって完全に関連付けられます。

サーバー側とエージェントのプラグインにはいくつかの類似点があります。

- すべてのプラグインは JAR ファイルとしてパッケージ化されます。
- すべてのプラグインには必要な XML ファイル（ **プラグイン記述子** ）があり、すべてのプラグイン機能を定義します。

- すべてのプラグインにはコンパイルされた Java ファイルが含まれます。このファイルには、記述子に定義されたすべてのアクションを実行するのに必要なコードが含まれます。
- プラグインは、**プラグインと直接対話し、すべてのプラグインを起動および停止するエンティティーであるプラグインコンテナー** 内で実行されます。
- すべてのカスタムプラグインが JBoss ON サーバーにデプロイされます。サーバー側のプラグインは他のサーバーすべてに高可用性クラウド全体に伝播されますが、エージェントをダウンロードするためにエージェントプラグインがサーバーを介して利用可能になります。

バグを報告します。

1.2. JBOSS ON のプラグインの基本コンポーネント

JBoss ON ではプラグインを構成するいくつかの一般的な要素があります。これらの各要素については、サーバー側のセクションとエージェントプラグインのセクションで説明しますが、このセクションでは、これらの要素に関するより一般的なコンテキストを紹介し、サーバー側のプラグインおよびエージェントプラグインがこれらの要素を使用する方法の違いを比較します。

バグを報告します。

1.2.1. プラグインコンテナー

すべての JBoss ON プラグインは、プラグインコンテナー内で実行されます。このコンテナーは、すべてのプラグインの読み込み、開始、および停止を行います。エージェントやサーバーはプラグインと直接対話することはありません。エージェントとサーバーの両方が、ホストプラグインコンテナーと対話します。エージェントまたはサーバーはプラグインコンテナーと通信し、プラグインコンテナーがプラグインと通信します。

エージェントプラグインでは、プラグインコンテナーに関する関係はありません。すべてのエージェントプラグインは同じものを使用します。コンテナーは基本的にプラグインライターには表示されません。

ただし、サーバー側のプラグインはプラグインコンテナーと非常に異なる関係があります。サーバーは、特定のサブシステムまたは目的に指定されたプラグインコンテナーを複数実行します。プラグインコンテナー自体は、追加のスキーマ定義と特定の機能を提供してサーバー側のプラグインの設定を提供します。プラグインコンテナーは、サーバー側のプラグインのタイプを区別してサーバー側のプラグインを最初に識別するカテゴリーです。

プラグインコンテナーは、エージェントまたはサーバーとプラグインの関係を制御すると共に、プラグインとそのクラス間の関係を緩和します。プラグインコンテナーは、プラグインの依存関係（エージェントプラグイン用）、共有クラス、およびプラグインが必要とする外部ライブラリーを管理します。

バグを報告します。

1.2.2. プラグイン記述子

プラグイン記述子は、特定のプラグインの動作を定義するファイルです。このファイルは、プラグインがプラグインコンテナーおよびエクステンションによってサーバーまたはエージェントと対話することを可能にする必要な API クラスを読み込みます。これは、プラグインインスタンスの特定の設定を定義し、スケジュールまたは操作を設定し、プラグイン向けの機能を明示的に定義します。

プラグイン記述子は常に XML ファイルです。エージェントおよびサーバー側のプラグインは、プラグインの JAR ファイルにある **META-INF/** ディレクトリーにプラグイン記述子を配置する必要があります。サーバー側のプラグインの場合は、このファイルの名前は **rhq-serverplugin.xml**およびエージェン

トプラグイン用に指定する必要があり **rhq-plugin.xml**ます。

バグを報告します。

1.2.3. プラグインスキーマの定義

プラグイン記述子は XML ファイルであるため、ファイル内の要素および属性を設定するために使用するスキーマ定義が必要です。JBoss ON のすべてのプラグインは、エージェントプラグインで定義されたコアスキーマを使用し **rhq-configuration.xsd**ます。サーバー側のプラグインは、追加のスキーマ定義ファイルと共にそのスキーマを拡張してから **rhq-serverplugin.xsd**、サーバー側の各プラグインタイプのカスタムスキーマ定義を拡張します。

バグを報告します。

1.2.4. Java ファイル

プラグインの実際のコードは、プラグイン JAR パッケージ内の Java ファイルに含まれます。

エージェントプラグインには、プラグインごとに少なくとも 2 つの Java ファイルと、場合によっては複数の Java ファイルがあります。これには、以下のような理由があります。

- エージェントプラグインは、同じプラグインで親要素と子要素（プラットフォーム、サーバー、およびサービス）の両方を定義でき、各リソースタイプは独自のプラグインコードを使用します。
- エージェントプラグインには、2 つの別個の機能があります。ほぼすべてのエージェントプラグインには、プラグインで定義されたリソースタイプを特定してインベントリーする方法を決定する検出コンポーネント（検出 Java ファイル）が必要です。さらに、エージェントプラグインはリソースのイベント収集を有効にすることもできます。これには、個別のコンポーネント（イベントポーリング Java ファイル）がリソースログを追跡する必要があります。最後に、プラグイン機能を実装するコンポーネント（Java ファイル）が必要です。
- エージェントプラグインは依存関係を許可します。親プラグインは、子とクラスを共有できます。エージェントプラグインは、そのプラグインクラスを読み込むことができる他のエージェントプラグインに依存関係を設定できます。プラグインのパフォーマンスが向上し、関連するプラグインコードへのアクセスを容易にするために、エージェントプラグインが頻繁に小さな Java ファイルに侵入され、プラグインコードを再利用できます。

サーバー側のプラグインには、プラグインの動作を定義する単一の Java ファイルのみが含まれます。サーバー側のプラグインには依存関係がなく、他のサブシステム（検出やイベント監視など）と対話しないため、プラグインに関連するすべてのサブシステムを 1 つのファイルに定義できます。

バグを報告します。

1.2.5. 外部ライブラリー

プラグインが必要とするライブラリーまたはクラスは、独自の Java ファイルに含まれない必要があるライブラリーまたはクラスは **外部** ライブラリーです。

エージェントプラグインは、依存関係と共有クラスを使用して相互作用できます。エージェントプラグインの外部ライブラリーまたはクラスは、**別のエージェントプラグインで定義されたライブラリーまたはクラスを参照し** ます。これは、他のエージェントプラグインで利用できるプラグインのすべての JMX ライブラリーおよび EMS ライブラリーをすべて作成するため、エージェントプラグインで一般的に JMX プラグインの依存関係を必要とするためです。エージェントプラグインは、（複数のプラグインで同じライブラリーを一度に利用可能にすることで）ライブラリー管理を簡素化し、プラグインの作成を簡素化する子プラグインでクラスを共有することも可能です。

サーバープラグインは相互に対話しないため、同じプラグインコンテナーであってもサーバー側のプラグイン間で依存関係を確立したり、クラスを共有したりすることはできません。ただし、サーバー側のプラグインでは、プラグイン JAR ファイルに外部ライブラリーをパッケージ化でき、JAR ファイル内の **lib/** ディレクトリーにあるライブラリーにアクセスできます。

バグを報告します。

1.3. プラグインファイルのダウンロード

サンプルプラグインは RHQ ソースコードから入手できます。コードを確認するには、次のコマンドを実行します。

```
git clone http://git.fedorahosted.org/git/rhq/rhq.git
```

エージェントおよびサーバー側のプラグインの例は sourceRoot/**etc**/**samples**/ ディレクトリーに **あり**ます。これには、完全に開発されたサンプルとプラグインテンプレートの両方が含まれます。これらのテンプレートを使用して、新しいプラグインを作成できます。ソースコード全体をチェックアウトするのではなく、この URL でサンプルファイルを手動でダウンロードできます。

```
http://git.fedorahosted.org/git/?p=rhq/rhq.git;a=tree;f=etc/samples;hb=master
```

バグを報告します。

第2章 サーバー側のプラグインの作成： 背景

すべての JBoss ON プラグインには同様の設定とデプロイメントスタイルがありますが、プラグインがシステムにアクセスするのに必要な内容が若干異なります。**サーバー側のプラグイン** は、コア JBoss ON サーバーにアクセスしてアクションを実行するプラグインを参照します。基本的には、これらのプラグインは中央サーバーの動作に使用されるグローバルプラグインです。

バグを報告します。

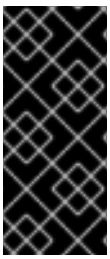
2.1. サーバー側のプラグインの概要

サーバー側のプラグインは JBoss ON サーバーの機能を拡張します。JBoss ON には、すでに複数のサーバー側のプラグインが同梱されています。

- リソースのアラート通知を送信するメソッドのアラート送信プラグイン。
- ファイルおよびアプリケーションをデプロイするためのバンドルプラグイン
- リソースまたはファイルシステムの設定およびファイルを監視するドリフトプラグイン
- リソース設定を管理するためのコンテンツプラグイン
- その他のすべての汎用プラグイン

サーバー側のプラグインはこれらの 3 つのカテゴリーに限定されません。サーバー側のプラグインフレームワークにより、サーバー自体に大量のアクセスが可能になります。サーバー側のプラグインは、イベントの監視に応答してリモートスクリプトを実行するか、またはカスタムワークフロー（JBoss ON サーバーのパービュー内のシステムをプロビジョニングする）を実行するために使用できます。

これは、より構造化された公式エージェントプラグインシステムよりも、プラグインを実装するための非常に簡単な方法です。これにより、プラグインの開発者が達成できるプラグインにより、よりレイテンシーが大きくなります。



重要

すべてのサーバー側のプラグインは、サーバーの状態レスセッション Bean(SLSB)に完全アクセスできます。これにより、サーバー側のプラグインの機能において、多くのレイティングおよび汎用性が可能になり、どの server サブシステムにもアクセスできるようになります。ただし、これによりサーバー側のプラグインが非常に強力になります。サーバー側のプラグインの作成およびデプロイには注意が必要です。

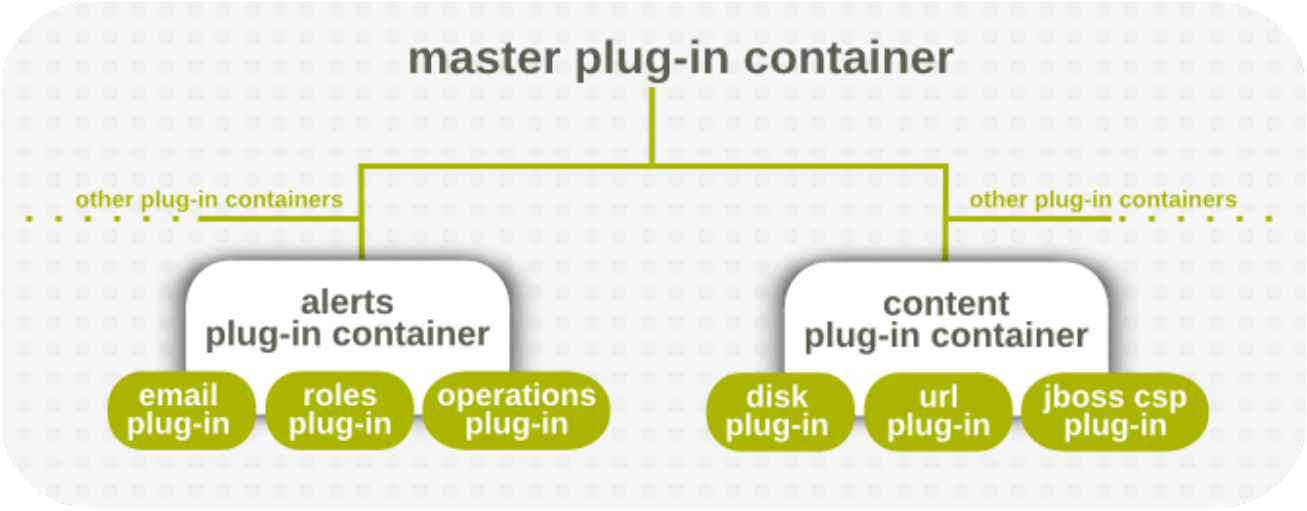
サーバー側のプラグインには、プラグインを作成し、デプロイするためのフレームワークが、エージェントプラグインのフレームワークとは異なります。以下は、サーバー側のプラグインの作成を開始する際に役に立つ一般的な情報の一部です。

- サーバー側のプラグインがビルドおよびデプロイされると、プラグインは **rhq-serverplugin.xml** プラグイン記述子が含まれる **META-INF/** ディレクトリーを持つ JAR ファイルです。
- 各プラグインは他のすべてのプラグインから独立しています。エージェントプラグインとは異なり、サーバー側のプラグインは相互に対話しません。サーバー側のプラグインにはプラグインの依存関係がありません。

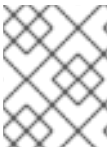
サーバー側のプラグインは **タイプ** 別に構成され、このタイプはプラグインが拡張するサブシステムまたは機能エリアに対応します。各タイプのプラグインは、定義された **プラグインコンテナーに** 含まれます。

サーバー側のプラグインは、プラグインの機能に関連するプラグインコンテナー内の JBoss ON サーバーで管理されます。プラグインコンテナーは、プラグインの開始や停止などの一般的なタスクを処理します。（すべてのプラグインコンテナーは、1 つのマスタープラグインコンテナーのメンバーです。）

図2.1 サーバー側のプラグインコンテナー



各プラグインには、各プラグインには 1 つのプラグインコンテナーしか存在しますが、各プラグインコンテナーにはサーバー側のプラグインの数に制限はありません。



注記

プラグインコンテナーは、プラグインの種類を定義します。1 つのタイプにしかできないため、プラグインは **1つ** のプラグインコンテナーにしか配置できません。

表2.1「利用可能なプラグインコンテナー」 利用できるプラグインコンテナーと JBoss ON の概要をまとめています。

表2.1 利用可能なプラグインコンテナー

プラグインの種類	description	コンテナー名
generic	すべてのカスタムプラグインの catch-all タイプ。このタイプのプラグインは、プラグインを開始および停止し、プラグインライブラリーを初期化およびシャットダウンするためにコンテナーのプラグインコンテナーとのみ対話します。	汎用プラグイン
アラートメソッド	アラート通知メソッドまたはアラートの送信方法を定義します。	AlertHandler

プラグインの種類	description	コンテナー名
bundle	バンドルのタイプを定義し、処理します。このタイプのプラグインは、コアサーバーが Ant レシピやファイルベースのバンドルなどの特定のバンドルタイプのバンドルを処理および管理するために必要なタスクを実行します。各バンドルサーバープラグインは単一のバンドルタイプを認識し、処理できます。	バンドルプラグイン
drift	ドリフト操作および設定を処理します。このファイルは、ドリフト検出および修復のために管理されるコンテンツ（ファイル）を保存し、取得します。	drift JPA プラグイン
コンテンツ	リポジトリーまたはリポジトリーグループのメタデータが含まれます。	PackageSource
リポジトリー（パッケージ）	コンテンツリポジトリーを定義します。プラグインは単一のリポジトリーを定義できます。リポジトリーは、JBoss の ON 管理リソースのプロビジョニング、エンタイトルメント、および更新に使用されます。	ChannelSource

注記

プラグインコンテナーは JBoss ON のコアコードの一部であるため、JBoss ON を再構築せずに新しいプラグインコンテナーを作成することはできません。新しいプラグインタイプを定義するのではなく、サーバーの機能に完全にアクセスできるため、汎用プラグインコンテナーを使用します。

バグを報告します。

2.2. サーバー側のプラグイン設定の内訳

JBoss ON プラグインはパッケージ化された **.jar** ファイルです。これらの **.jar** ファイルによって使用されるディレクトリー構造、ライブラリー、およびクラスは、プラグインライターの判断と要件により完全に最大で、すべてのプラグインファイルにはプラグイン記述子 **.jar** ファイルが必要です **META-INF/rhq-serverplugin.xml**。

注記

プラグインを作成する際の主要なガイドラインは、**org.rhq.enterprise.server.plugin.pc.ServerPluginComponent** クラス。これは、コンテナー内のプラグインのライフサイクルを制御します。

サーバー側のプラグインは、3 種類のファイルを使用して JBoss ON サーバー内で定義されます。

- プラグインの記述子として機能する XML ファイル
- 記述子情報をプルし、プラグインのクラスを実装する Java ファイル。
- オプションのライブラリー依存関係。サードパーティーライブラリーは、プラグイン JAR ファイルの **lib/** ディレクトリーに保存する必要があります。

バグを報告します。

2.2.1. 記述子および設定

プラグイン記述子は、プラグインのデプロイ方法とプラグインの設定および動作に関する追加情報とともにプラグインコンテナーに指示するメカニズムです。プラグイン記述子は XML ファイルに含まれています。このファイルは、デフォルトまたはユーザー定義のタグと属性を使用してその設定を定義できます。

注記

サーバー側のプラグインの XML ファイルは、プラグインの JAR **rhq-serverplugin.xml** ファイルの **META-INF/** ディレクトリーのファイルで定義されます。（デフォルトのサーバー側のプラグインは同じ設定に従います。） このファイルは必須です。

プラグイン記述子の最も重要な設定は、プラグインのタイプ、その名前、およびそのバージョンを含むプラグインの基本的な定義です。すべてのプラグインには基本的な定義があります。バージョン番号がプラグイン記述子で手動で渡されていない場合は、**MANIFEST.MF** ファイルから自動的に取得されます。

サーバー側のプラグインの鍵は柔軟性です。サーバー機能への絶対的なアクセスがあり、監視、アラート、リモートアクション、プロビジョニング、リソース設定など、サーバーの既存の機能のいずれかを拡張できます。この柔軟性を維持するには、サーバー側のプラグインには少なくとも 3 つの一般的なセクションで高度な設定のオプションがあります。

- 定期的に、または cron スケジュールを使用したアクションのスケジュール設定
- 特定のプラグインタイプの全インスタンスに対するグローバルパラメーターの設定
- プラグインタイプでのローカルまたはインスタンス固有の設定の許可

バグを報告します。

2.2.1.1. 定義およびクラス

各サーバー側のプラグインにはルート要素があり、名前、表示名、パッケージ、バージョン、およびその他のプラグイン情報の属性が含まれます。これは、プラグイン設定に使用される XML スキーマ定義をインポートし、定義します（詳細はで説明します 「スキーマファイル」）。

例2.1 プラグイン記述子： 定義

```
<alert-plugin
        name="alert-email"
        displayName="Alert:Email"
        xmlns="urn:xmlns:rhq-serverplugin.alert"
        xmlns:c="urn:xmlns:rhq-configuration"
```

```
xmlns:serverplugin="urn:xmlns:rhq-serverplugin"
package="org.rhq.enterprise.server.plugins.alertEmail"
description="Alert sender plugin that sends alert notifications via email"
version="1.0"
>
```

プラグイン設定の 2 番目の部分は、プラグインで使用するコンポーネントまたはクラスを設定します。すべてのサーバー側のプラグインが以下を実装します。
org.rhq.enterprise.server.plugin.pc.ServerPluginComponent class: プラグインの簡単なライフサイクル管理を提供します。このコンポーネントは、プラグインを初期化、開始、停止、およびシャットダウンするコンテナーのフックを提供します。プラグインが初期化されると、プラグインのランタイム環境に関する情報を提供するサーバープラグインコンテキストが付与されます。

このコンポーネントはステートフルオブジェクトであり、プラグインが初期化されている限り存続します。このオブジェクトは正常ですが、プラグインはタスクを実行するか、必要な作業を行うために任意のメソッドを呼び出すことができます。

開発者は、プラグイン用のコンポーネントを以下のいずれかの方法で呼び出すオプションがあります。

- **<plugin-component>** タグを使用したクラスの指定（すべてのプラグインで利用可能）
- ユーザー定義タグを使用したクラスの特定（プラグインコンテナーで利用可能なスキーマに応じてサーバー側のプラグインの一部で利用可能）

プラグインに指定の呼び出しメソッドを使用する必要はありません。したがって、<plugin-component> などの使用は任意です。コンポーネントを呼び出す方法に関係なく、記述子に指定できるプラグインコンポーネントは 1 つだけです。

例2.2 プラグイン記述子： クラス情報

```
<serverplugin:plugin-component class="MyLifecycleListener" />
```

このプラグイン **<plugin-class>** 用に、コンテナー定義タグ（電子メールアラートサーバー側のプラグインなど）を作成できます。クラスを作成すると、オプションが導入され、コンポーネントで設定オプションやその他の情報が提供されます。

```
<plugin-class>RolesSender</plugin-class>
```

注記

の例は、特定の種類のサーバー側のプラグインに固有の 例2.2「プラグイン記述子： クラス情報」 ものです。すべてのサーバー側のプラグインが、その構造に対応しているわけではありません。

一部の記述子タグは、そのプラグインタイプに定義されたスキーマを使用して利用できます。これは、プラグインコンテナースキーマファイルで定義されるスキーマです。この例では、アラートセンダープラグインコンテナーは、JBoss ON のアラートメカニズムにフックするためのアラートセンダープラグインの **<plugin-class>** 要素を提供します。

コンテナー定義スキーマはアドホックではありません。これはプラグインファイルのみにドロップできないため、開発者は独自のスキーマ要素を定義することはできません。

バグを報告します。

2.2.1.2. コントロール操作

サーバープラグインのステートフルコンポーネントと直接対話する必要がある場合があります。この対話は任意の数のフォームを取ることができます。そのため、プラグイン自体をテストするためにエージェントの一覧やリソースの一覧を取得してプラグイン自体をテストします。

ユーザー定義のコントロールを許可するには、 **ServerPluginComponent** クラスはオプションで ControlFacet インターフェースを実装できます。これらの制御操作は、プラグイン設定領域にある JBoss ON Web インターフェースで直接呼び出すことができます。

コントロール操作は、要素の子である **<control>** 要素を使用してプラグイン記述子で設定され **<plugin-component>** ます。コントロールはオプションであるため、何も指定する必要はありません。また、複数のコントロールを指定することもできます。各コントロールには、ユーザーが制御操作に渡すオプションのパラメーター、および（オプション）結果プロパティーを指定することもできます。

例2.3 制御操作の設定

```
<serverplugin:plugin-component class="MyLifecycleListener">
  <serverplugin:control name="testControl" description="A test control operation">
    <serverplugin:parameters>
      <c:simple-property name="paramProp" required="true" description="Set to 'fail' to simulate an error"/>
    </serverplugin:parameters>
    <serverplugin:results>
      <c:simple-property name="resultProp" required="false"/>
    </serverplugin:results>
  </serverplugin:control>
</serverplugin:plugin-component>
```

制御操作は、サーバー側のプラグインタイプで使用できます。

バグを報告します。

2.2.1.3. ジョブのスケジュール設定

サーバー側のプラグインフレームワークの主な利点の 1 つは、プラグインのスケジュールされたジョブを定義する機能です。プラグインコンテナーは、それらのジョブの実際にスケジューリングと呼び出しを処理します。プラグイン記述子には、ジョブのトリガー時に呼び出されるクラスやメソッド、それらのジョブがトリガーされる頻度、およびジョブが呼び出されるときにジョブメソッドに渡す設定設定が簡単にプラグインコンテナーに指示されます。

このジョブは、プラグインのステートフルコンポーネントへのアクセスや、ジョブ自体についての情報が呼び出される際に実行する必要のあるすべての作業を実行できます。
ScheduledJobInvocationContext コンポーネント

ジョブ設定が完全に柔軟性があります。

- ジョブクラスはステートレス（各ジョブクラスが呼び出されるたびにインスタンス化される）か、プラグインコンポーネントインスタンスを呼び出すことでステートフルにすることができます。

注記

サーバー側のプラグインは、プラグインのライフサイクルリスナーとして機能するプラグインコンポーネントを定義できます。プラグインコンポーネントを使用することは非常に便利です。実際には、汎用サーバー側のプラグインがコアサーバーに接続する唯一のメカニズムです。

- ジョブは同時に実行できます。つまり、（同じサーバー上を含む）任意の数のサーバーで一度に複数の呼び出しを実行できます。ジョブが同時実行されない場合、1つのジョブ呼び出しはいつでも1つのジョブ呼び出しのみを実行できます。（ジョブが同時ではなく、クラスター化されていない場合 **は**、JBoss ON サーバークラウドのどこにでも1つのジョブ呼び出しのみを実行できます）。

- ジョブはクラスター化できます。つまり、ジョブは JBoss ON サーバークラウドの任意のサーバーから実行できます。ジョブがクラスター化されていない場合、ジョブは常にジョブがスケジュールされたマシンで実行されます。これは、concurrent 設定と併用して動作します。

- スケジュールは周期的（毎時実行など）か、パターン（たとえば、月曜日の 5pm 時など）で繰り返し発生します。

- 同じプラグインに複数のジョブをスケジュールし、それぞれはプラグインの **<scheduled-jobs>** エントリー **<map-property>** 下に置かれます。

それぞれのスケジュールされたジョブは、ジョブによって呼び出される名前、スケジュール、頻度、メソッド、またはすべてのコールバックデータを設定するマッピングエントリーです。

例2.4 プラグイン記述子： スケジュールされたジョブ

```
<serverplugin:scheduled-jobs>
        <!-- notice that we use the map name as the methodName -->
        <c:map-property name="myScheduledJobMethod1">
            <c:simple-property name="enabled" type="boolean" required="true" default="true"
summary="true" description="Whether or not the job should be scheduled"/>
            <c:simple-property name="scheduleType" type="string" required="true" default="cron"
summary="true" description="Indicates when the schedule triggers">
                <c:property-options>
                    <c:option value="periodic"/>
                    <c:option value="cron" default="true"/>
                </c:property-options>
            </c:simple-property>
            <c:simple-property name="scheduleTrigger" type="string" required="true" default="0 0/5 * *
* ?" summary="true" description="Based on the schedule type, this is either the period, in
milliseconds, or the cron expression"/>
            <c:simple-property name="concurrent" type="boolean" required="false" default="false"
summary="true" description="Whether or not the job can be run multiple times concurrently"/>
            <c:simple-property name="clustered" type="boolean" required="false" default="true"
summary="true" description="Whether or not the job can be run anywhere in the JBoss ON server
cluster, or if it must be run on the server where the job was schedule."/>
        </c:map-property>
</serverplugin:scheduled-jobs>
```

<scheduled-jobs> コンテナーエントリーは1つだけです。個々のジョブは、マッピング(**<map-property>**)エントリー内のこのコンテナー内にあります。

バグを報告します。

2.2.1.3.1. ジョブの状態

サーバー側のプラグインは、Stateless または stateful の 2 つのジョブのいずれかを実行できます。ステートフルジョブとステートレスジョブの唯一の違いは、ジョブがクラスを指定するかどうかのみです。プラグインがクラスを指定し **ない** 場合、プラグインジョブはプラグインコンポーネントを使用するため、ステートフルになります。ジョブがクラスを指定する場合、クラスは新規ジョブの開始ごとにインスタンス化されるため、ジョブはステートレスになります。

最も単純な場合、ステートレスジョブはクラスと、プラグインの開始時に呼び出すメソッドのみを必要とします。例：

例2.5 ステートレスジョブ設定

```
<c:map-property name="statelessJob1" description="invokes a stateless job class but given a job context">
    <c:simple-property name="class" type="string" required="true" readOnly="true" default="MyScheduledJob" summary="true" />
    <c:simple-property name="methodName" type="string" required="true" readOnly="true" default="executeWithContext" summary="true" />
</c:map-property>
```

ステートレスジョブに指定されたクラスを除き、ステートレスおよびステートフルジョブには同様の設定オプションがあります。ステートフルジョブとステートレスジョブの両方が、ジョブのスケジュールに役立つその他のオプションパラメーターを取ることができます。スケジュールされたジョブはプラグインの他のコンポーネントと同じ設定プロパティーを使用しますが、スケジュールされたジョブには、ジョブの作成に特別なプロパティーを定義する必要がある特殊なセマンティクスがあります。基本的に、これは各ジョブのプロパティーマップです。これらのプロパティーには以下が含まれます。

1. 呼び出すジョブのメソッド名。ステートフルジョブの場合、ターゲットメソッドはプラグインコンポーネントにあります。ステートレスジョブの場合、これは class プロパティーで指定されたクラスにあります。いずれの方法でも、メソッド名はサーバーに対して呼び出しを指示します。default メソッドはすでにプラグインコンポーネントで定義されており、特定のメソッド名プロパティーを使用せずにステートフルジョブで呼び出すことができます。

   ```
   <c:simple-property name="methodName" type="string" required="true" readOnly="true" default="executeWithContext" summary="true" />
   ```

   メソッドに引数がない場合、または型の引数が 1 つ必要です。
   ScheduledJobInvocationContext.

2. ジョブが有効であるかどうかを示す設定。

   ```
   <simple-property name="enabled" type="boolean" ... />
   ```

3. Periodic または cron ジョブであるかどうかを示すスケジュールタイプ。ジョブのタイプは、true に設定されるオプションで識別されます。例：

   ```
   <simple-property name="scheduleType" ... default="periodic" ... >
       <c:property-options>
           <c:option value="periodic" default="true"/>
   ```

```
            <c:option value="cron" />
        </c:property-options>
    </c:simple-property>
```

4. ジョブを実行するタイミング（「トリガー」）の実際のスケジュール。これは期間または cron スケジュールのいずれかです。定期的なジョブの場合、これは時間間隔（ミリ秒単位）を指定します。

```
<simple-property name="scheduleTrigger" type="string" required="true" default="60000" ... />
```

cron ジョブでは、デフォルトの引数に完全な cron 式が含まれます。

```
<simple-property name="scheduleTrigger" type="string" required="true" default="0 0/5 * * * ?"
... />
```

（cron スケジュール形式の完全な説明は http://www.quartz-scheduler.org/documentation/quartz-2.1.x/tutorials/tutorial-lesson-06.html にあり ます ）。

5. ジョブが同時実行であるかどうかの設定（つまり、このジョブを複数のサーバーで複数回実行できるか、または同時に実行できるか）。false の場合、ジョブの 1 つのインスタンスのみを一度に実行できるようにし、複数のサーバーがジョブの実行をスケジュールされていても、それらの 1 つのインスタンスでのみ実行されます。

```
<simple-property name="concurrent" type="boolean" ... />
```

6. ジョブは、JBoss ON サーバークラウドのどこにでも実行するか、またはジョブがスケジュールされたマシンで実行する必要があるかの設定を許可します。クラスターの値を true に設定すると、ジョブがクラスター化されるよう JBoss ON クラウドのサーバーからジョブを呼び出すことができます。この値は、ジョブをスケジュールされているすべてのマシンで実行する必要がある場合は false である必要があります。すべてのプラグインはすべてのサーバーに自動的に登録されるため、クラスター化されていないジョブは独立して各サーバー上で実行されます。

```
<simple-property name="clustered" type="boolean" default="true" ... />
```

7. ジョブには、任意でコールバックデータを受け入れるカスタム文字列を含めることができます。

```
<simple-property name="custom1" type="boolean" required="true" default="true"
summary="true" description="A custom boolean for callback data"/>
```

コールバックデータは、ブール値、文字列、long など、実行されるジョブに適するタイプです。

8. ステートレスジョブには、クラスのメソッド名を渡すプロパティーがあります。メソッド名はプラグインコンポーネントで呼び出されるクラスを識別するか、またはジョブが呼び出されるタイミングをインスタンス化するためにクラスを呼び出すこともできます。メソッドとクラスキーの両方が表示され 例2.5「ステートレスジョブ設定」 ます。ターゲットとして使用されるクラスには、メソッド名 simple プロパティーでメソッドが定義されている必要があります。

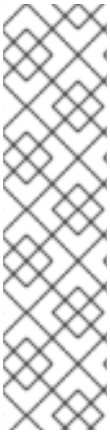
通常、ジョブはステートフルプラグインコンポーネントをターゲットとするため、クラスは指定されません。class プロパティーでは、ステートレスジョブの作成オプションが許可されます。

バグを報告します。

2.2.1.3.2. 同時およびクラスター化されたジョブ

スケジュールされたジョブ**が** 実行されると、スケジュールによって決定されます（**scheduleTrigger** 設定）ジョブの実行場所**と** 実行 **方法** は、concurrent と clustered の 2 つの設定で決定されます。

ジョブがスケジュールされた時間に到達すると、特定の JBoss ON サーバーが実行されるとは限りません。サーバーがタスクを実行するサーバーを決定する必要があり、これはクラスター化設定で提供されます。クラスター化設定を true に設定すると、JBoss ON サーバークラウドのすべてのサーバーがタスクを呼び出しできます。false に設定されている場合、タスクはスケジュールされているサーバーでのみ実行できます。



注記

クラスタリングの 1 つの点として、ジョブを JBoss ON サーバークラウドの任意のサーバーで実行 **できる** 間は、ジョブを実行するサーバーを予測したり、必要とできない点が挙げられます。一部のマシンはジョブを実行しない可能性があります。

一方、JBoss ON のサーバー側のプラグインはデプロイされると、クラウドのすべてのサーバーに自動的に伝播されます。クラスタリングがオフになっている場合（各ジョブはローカルサーバーからのみ実行されることを意味します）、すべての JBoss ON サーバーは、他のサーバーがジョブを実行したときに独立してこのジョブを実行します。最終的な結果は、すべての JBoss ON サーバーがこのジョブを一貫したスケジュールで実行し、同時に複数のジョブを実行することも保証されます。

JBoss ON サーバーがタスクを実行する場所を特定したら、タスクがすでに実行されているかどうかを確認する必要があります。同時設定が true の場合、タスクがすでに別のサーバー（または同じサーバーであっても）を実行している場合でも、ジョブはすべてのスケジュールがトリガーされます。同時実行が false に設定され、ジョブがすでに JBoss ON サーバークラウドのどこかで実行されている場合、ジョブの実行前にジョブの呼び出しが完了するまでサーバーが待機する必要があります。

クラスター化および同時接続設定は複数の方法で相互に再生できます。

ジョブがクラスター化されていて、JBoss ON サーバーがジョブを呼び出す前に、JBoss ON サーバークラウドの他の場所で実行しているかどうかを確認する必要があります。その場合、サーバーはジョブが完了するまで待機してから、新しいジョブを呼び出す必要があります。

ジョブがクラスター化されず、同時実行されていない場合、JBoss ON サーバーはローカルマシンを確認し、ジョブが実行されているかどうかを確認します。ジョブがローカルで実行されていない場合、JBoss ON サーバーはジョブがクラスター化されていないため、クラウドの別のサーバーで実行している場合でもジョブを呼び出すことができます。

クラスター化設定は、同時実行チェックをどの程度厳格にするかを決定します。クラスター化が false の場合、同時実行チェックはジョブがスケジュールされたマシンでのみ実行されます。clustered が true の場合、同時実行チェックはクラスター内のすべてのマシンで実行されます。

表2.2 同時およびクラスター化された動作の比較

同時実行	clustered	スケジュールがトリガーされると、
true	true	...ジョブは常に呼び出されます。JBoss ON サーバークラウドの任意のサーバーで呼び出されることがあります。

同時実行	clustered	スケジュールがトリガーされると、
true	false	... ジョブは常に呼び出され、ジョブがスケジュールされているサーバー上で実行されます。
false	true	JBoss ON サーバーは、このジョブが JBoss ON サーバークラウドの他の場所で実行されているかをチェックします。その場合、新しいジョブは、開始するまで古いジョブが終了するまで待機する必要があります。JBoss ON サーバークラウドのどこでも、このジョブのインスタンスは 1 つしか実行できません。
false	false	... スケジューラーは、ジョブを呼び出す前にローカルでジョブが実行されているかどうかを確認します。ジョブ呼び出しは一度に 1 つのみサーバー上で実行することができますが、クラウドの複数のサーバーはジョブを同時に実行している可能性があります。

注記

ジョブが一貫したスケジュールの **すべて** のサーバーで実行されるようにするには、**クラスター化** を false に設定します。**concurrent** は、古いジョブがマシンで依然として実行されている間、マシンで新規ジョブを開始できるかどうかを判断します。

指定された JBoss ON サーバーでジョブをどこかで実行するには、**クラスター化** を true に設定します。**concurrent** は、いつでも複数のジョブを実行することが許可されるかどうかを判断します。

バグを報告します。

2.2.1.4. プラグイン設定（グローバルおよびローカルの両方）

グローバルな構成設定は、サーバー側のプラグインの各インスタンスにデフォルト値またはグローバル設定に設定できます。グローバル設定パラメーターはすべて **<plugin-configuration>** エントリー内（標準の JBoss ON スキーマで定義）に含まれ、各パラメーターが **<simple-property>** 項目と識別されます。グローバル設定は、同じメールや SNMP アカウントを使用するアラートなど、単一のアイデンティティーにアクセスするプラグインに役立ちます。

例2.6 プラグイン記述子： グローバル設定

```
<serverplugin:plugin-configuration>
        <c:simple-property name="user" type="string" required="false"/>
        <c:simple-property name="password" type="password" required="false"/>
```

```
</serverplugin:plugin-configuration>
```

同じサーバー側のプラグインを複数のインスタンスで作成できます。これらの異なるインスタンスは、異なる機能を満たすために、若干異なる設定を必要とします。たとえば、電子メールアラート送信者の異なるインスタンスは、sys admins の異なるグループに通知を送信する必要があります。

これらのインスタンス固有の構成設定は、サーバー側のプラグインに固有のスキーマ（や **<alert-configuration> <simple-property>** 項目など）を使用して設定エントリーを使用して、プラグイン記述子で識別されます。

例2.7 プラグイン記述子： インスタンス固有の設定(Alert)

```
<alert-configuration>
        <c:simple-property name="emailAddress" displayName="Receiver Email Address(es)"
type="longString"
                description="Email addresses (separated by comma) used for notifications."/>

h5. </alert-configuration>
```

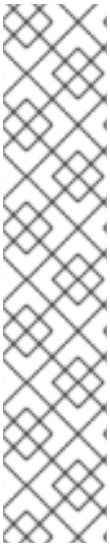
各プラグインコンテナータイプは、そのタイプのプラグインに関連する独自のスキーマセットを定義します。たとえば、GUI やパースペクティブには、異なるタイプの UI 要素に対して個別の明示的なスキーマ要素があります。

例2.8 プラグイン記述子： インスタンス固有の設定(Perspectives)

```
<perspectivePlugin
  description="The Core Perspective defining Core UI Elements"
  displayName="Core Perspective"
  name="CorePerspective"
  package="org.rhq.perspective.core"
  xmlns="urn:xmlns:rhq-serverplugin.perspective"
  xmlns:serverplugin="urn:xmlns:rhq-serverplugin"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">


<!-- Menu -->
<menuItem name="logo" displayName="" url="/" iconUrl="/images/JBossLogo_small.png">
  <position placement="firstChild" />
</menuItem>
```

sourceRoot **ディレクトリー/modules/enterprise/server/xml-schemas/src/main/resources** のプラグインコンテナースキーマをチェックして、特定のタイプのプラグインで利用可能な要素を確認します。すべてのプラグインタイプがローカル設定を許可する訳ではありません。たとえば、一般的なプラグインはグローバルプラグインの設定のみを許可します。



注記

コンテナースキーマは、JBoss ON パッケージではなく、RHQ ソースコードに含まれています。コードを確認するには、次のコマンドを実行します。

```
git clone http://git.fedorahosted.org/git/rhq/rhq.git
```

プラグインの例は sourceRoot/**etc/samples/custom-serverplugin/** ディレクトリー **で**、新規プラグインを作成するためのテンプレートとして使用できます。ソースコード全体をチェックアウトするのではなく、この URL で custom-serverplugin ファイルを手動でダウンロードできます。

```
http://git.fedorahosted.org/git/?p=rhq/rhq.git;a=tree;f=etc/samples/custom-serverplugin;hb=master
```

バグを報告します。

2.2.2. スキーマファイル

サーバー側のプラグインは、その XML プラグイン記述子ファイルのメタデータおよび設定を使用して定義されます。記述子で **使用できる** 設定要素は、プラグインコンテナータイプの XML スキーマ定義 (XSD)ファイルで定義されます。

記述子ファイルは、プラグインタイプスキーム内の要素に準拠する必要があります。記述子が必須要素がない、またはプラグインコンテナーのスキーマに定義されていない要素を使用しようとすると、プラグインの読み込みに失敗します。

すべてのプラグイン（エージェントプラグインおよびサーバー側のプラグイン）は、この **rhq-configuration.xsd** ファイルを使用します。このファイルは、プラグインで利用可能な基本的な設定オプションを定義します。

rhq-configuration.xsd ファイルのスキーマはにより拡張され **rhq-serverplugin.xsd**ます。このファイルは、サーバー側のプラグインの機能に固有の追加の XML 要素を提供します。このファイルは、すべてのサーバー側のプラグインによって参照されます。

サーバー側のプラグインが使用する最後の XSD ファイルは、プラグインコンテナーに固有のものです。プラグインコンテナースキーマファイルは、そのタイプのプラグインに必要な要素を定義するか（アラートセンダープラグインの場合と同様に）、特定のスキーマ要素（汎用プラグインの場合と同様）がありません。

特定のサーバー側のプラグインスキーマファイルは sourceRoot/**modules/enterprise/server/xml-schemas/src/main/resources** ディレクトリーに **あり**ます。

このセクションでは、各 XSD に関連付けられる設定要素および属性の概要を取り、一般的に XSD について十分に理解し、特に JBoss ON を使ってサーバー側のプラグインを作成し、必要に応じてスキーマを拡張します。

各 XSD ファイルの詳細は、XSD ファイル自体のコメント（**<xs:annotation>** アイテム内）で確認できます。XSD ファイルおよび XML スキーマの詳細は、http://www.w3.org/TR/xmlschema-0/ のように XML および XSD のリファレンスガイドを参照して ください。

注記

JBoss ON XSD ファイルには、ファイル内の各設定エリアの説明が含まれるアノテーションが付けられます。

バグを報告します。

2.2.2.1. プラグインコンテナースキーマファイルの解析

特定のタイプのサーバー側のプラグインに使用でき、必要なスキーマ要素はすべて、そのタイプのプラグインコンテナーのスキーマで定義されます。XSD ファイルで 2 つの関連する要素が設定されます。

- 要素
- attributes

注記

XML スキーマの詳細は、XML および XSD のリファレンスガイドを参照してください。
例 : http://www.w3.org/TR/xmlschema-0/

要素 はプラグインの XML ファイルで使用できるタグに変換されます。例：

```
<xs:element name="alert-plugin">
```

プラグインの XML ファイルで、その要素がタグを定義します。

```
<alert-plugin>
Stuff
</alert-plugin>
```

属性 は、XML ファイルのタグで使用できるフラグです。例：

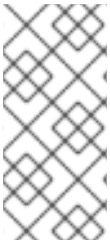
```
<xs:attribute name="name">
```

この属性は XML ファイルの以下のようになります。

```
<alert-plugin name="myAlertPlugin">
Stuff
</alert-plugin>
```

要素と属性は、XSD ファイルに階層的に編成されます。プラグインファイルの container 要素は、XSD の上部で定義されます。子要素は親要素のタイプを参照し、親の定義内にサブ要素として含まれます。同様に、要素で使用できる属性は要素の定義に含まれます。

サーバー側プラグインのタイプに定義されたタグと属性を検索する最も簡単な方法の 1 つは、sourceRoot **ディレクトリー/modules/enterprise/server/xml-schemas/src/main/resources** のプラグインコンテナースキーマを確認し **<xs:element name="">**、**<xs:attribute name="">** エントリーを検索することです。



注記

コンテナースキーマは、JBoss ON パッケージではなく、RHQ ソースコードに含まれています。コードを確認するには、次のコマンドを実行します。

```
git clone http://git.fedorahosted.org/git/rhq/rhq.git
```

バグを報告します。

2.2.2.2. rhq-configuration.xsd ファイル

この **rhq-configuration.xsd** ファイルは、すべての JBoss ON プラグインで使用できるスキーマを提供します。これは、エージェントとサーバー側のプラグインにより使用されます。

rhq-configuration.xsd ファイルは **ソース**にあり**/modules/core/client-api/src/main/resources**ます。

注記

コンテナースキーマは、JBoss ON パッケージではなく、RHQ ソースコードに含まれています。コードを確認するには、次のコマンドを実行します。

```
git clone http://git.fedorahosted.org/git/rhq/rhq.git
```

プラグインの例は sourceRoot/**etc/samples/custom-serverplugin/** ディレクトリー **で**、新規プラグインを作成するためのテンプレートとして使用できます。ソースコード全体をチェックアウトするのではなく、この URL で custom-serverplugin ファイルを手動でダウンロードできます。

```
http://git.fedorahosted.org/git/?p=rhq/rhq.git;a=tree;f=etc/samples/custom-serverplugin;hb=master
```

rhq-configuration スキーマで定義される最も一般的な要素は、やのようなプラグインの設定値の設定に関連 **<simple-property>** し **<map-property>**ます。

表2.3 rhq-configuration.xsd スキーマ要素

要素	description
configuration-property	ユーザー定義の設定のプラグインに設定属性を追加する場合。
simple-property	デフォルト設定値を設定する場合。
オプション	プロパティーの値が列挙された一覧(false)から取得されるか、またはユーザーが定義する値(true)であるかを設定します。

rhq-configuration.xsd file また、必須属性や **name** オプション **displayName** 属性など、プラグイン記述子に使用できる最も一般的なフラグも定義します。

表2.4 rhq-configuration.xsd スキーマ要素属性

attribute	description
Name	**必須。**プラグインの一意の名前を指定します。

attribute	description
displayName	GUI のプラグインに使用する名前を指定します。これを指定しない場合には、name の値が使用されます。
description	プラグインの簡単な説明を提供します。

rhq-configuration.xsd ファイルに多くの要素および属性が設定されます。それぞれの項目の説明は、項目の **<xs:annotation>** タグのテキストで説明されています。

バグを報告します。

2.2.2.3. rhq-serverplugin.xsd ファイル

rhq-serverplugin.xsd は中央のサーバー側のプラグインスキーマファイルです。

rhq-serverplugin.xsd ファイルは、サーバー側のプラグインごとに重要なスキーマ要素を提供します。おそらく、最も重要な要素は **<server-plugin>**（プラグインのルート要素用）と **<scheduled-jobs>**（リソースまたはサーバーでジョブを実行する場合）です。

rhq-serverplugin.xsd ファイルは **ソース**にあり**/modules/enterprise/server/xml-schemas/src/main/resources**ます。

rhq-serverplugin.xsd ファイルの最も一般的な要素はに記載されてい 表2.5「rhq-serverplugin.xsd スキーマ要素」 ます。

表2.5 rhq-serverplugin.xsd スキーマ要素

要素	description
server-plugin	プラグイン記述子のルート要素が含まれます。
help	には、プラグインを他のアプリケーションと統合するのに役立つ追加の使用情報またはその他のヒントが含まれます。
plugin-component	プラグインが停止したり、開始したときに通知されるクラスを特定します。これはステートフルオブジェクトで、スケジュールされたステートフルジョブのターゲットです。
scheduled-jobs	指定されたタスクを実行するプラグインのスケジュールを定義します。

に定義さ **rhq-serverplugin.xsd** れている属性のほとんどは、プラグイン記述子のルート要素で使用される include フラグです。これらは、サーバー側のプラグインのリリースおよび更新を制御する管理属性を追加します。

表2.6 rhq-serverplugin.xsd スキーマ要素属性

attribute	description
パッケージ	プラグインパッケージ名を設定する場合。
version	プラグインのバージョンを設定する場合。バージョンが記述子に設定されていない場合、プラグイン JAR ファイルは **Implementation-Version** 設定でバージョン番号を定義 **META-INF/MANIFEST.MF**する必要があります。
apiVersion	プラグインの作成に使用する API のバージョンを設定する場合。

rhq-serverplugin.xsd ファイルに多くの要素および属性が設定されます。それぞれの項目の説明は、項目の **<xs:annotation>** タグのテキストで説明されています。

バグを報告します。

2.2.3. Java クラスファイル

などの要素を実装するためにプラグインによって使用される Java クラスファイルは、プラグインの JAR ファイルで利用可能である **ServerPluginComponent ControlFacet** 必要があります。

バグを報告します。

2.3. ALERT SENDER SERVER-SIDE PLUG-INS のATOMY

アラート通知送信者は、アラートの送信に使用されるメソッドです。各送信者は、アラート送信者のプラグインを使用して実装されます。同じタイプのプラグインの複数インスタンスは、異なる設定で設定できます。すべてのプラグインが提供する機能は、その方法でアラートを送信する機能です。

アラート送信者は、サーバー側のプラグインとして実装されます（で説明しているものと同じ一般的な設定の概念があります） 「サーバー側のプラグイン設定の内訳」。サーバー側のプラグインフレームワークにより、アラート送信者はカスタムプラグインを介して簡単に拡張でき、デフォルトのサーバー側のプラグインの設定を編集することもできます。

このセクションでは、デフォルトのサーバー側のプラグイン（電子メールアラート送信者）のいずれかの要素を定義して、アラート送信者の作成プロセスを明確にし、簡単にします。

バグを報告します。

2.3.1. デフォルトのアラート送信者

JBoss ON は、アラート送信の最も一般的な方法の一部に対応するデフォルトのインストールを含む複数のアラートセンダープラグインを提供します。

表2.7 デフォルトのアラート送信者

Alert メソッド	description	プラグイン名

Alert メソッド	description	プラグイン名
email	アラート情報を使用して、ユーザーまたはユーザー一覧に電子メールを送信します。	alert-email
roles	内部メッセージを JBoss ON ユーザーロールに送信します。	alert-roles
SNMP	SNMP トラップに通知を送信します。	alert-snmp
operations	ターゲットリソースで JBoss ON がサポートするタスクを開始しました。	alert-operations
subject	JBoss ON のユーザーに通知を送信します。	alert-subject

プラグインの開発者および管理者は、カスタムアラート送信者プラグインを作成してデプロイし、追加のインスタントメッセージングシステムなどの組織固有の他のシナリオや形式に対応できます。

バグを報告します。

2.3.2. 実際のアラート送信プラグインの内訳

の説明にあるように 「サーバー側のプラグイン設定の内訳」、サーバー側のプラグインは設定に 3 種類のファイルを使用します。

- 所定の XML スキーマファイル(XSD)に準拠する XML プラグイン記述子
- Java ファイル

XML プラグイン記述子と Java ファイルは、すべてのプラグインに固有です。ただし、すべてのデフォルトのアラート送信者は、同じ 3 つのスキーマファイルを使用して記述子の属性を提供します。

「サーバーサイドプラグインのデプロイ」 プラグインの構築およびデプロイプロセスについて説明します。このセクションでは、デフォルトのアラート送信者(alert-email)を定義するために使用される各設定ファイルの要素に、アラートプラグインの作成方法の例としてアノテーションを付けます。

バグを報告します。

2.3.2.1. 記述子

すべてのプラグイン記述子は、そのプラグインの **src/main/resources/META-INF/** ファイル **rhq-serverplugin.xml** で呼び出されるファイルです。

注記

デフォルトのアラートスキーマは、アラートプラグインバリデーターが機能し、アラートがモニタリングシステムに正常に接続できるようにするためにプラグイン記述子と共に使用する必要があります。

プラグイン記述子のヘッダーは、プラグインで使用するスキーマファイルをプルし、プラグインのパッケージ情報（クラス、説明、バージョン番号）を定義します。**displayName** フラグには、インストール済み **のサーバー側のプラグイン** 一覧のプラグインに付与する名前が含まれます。

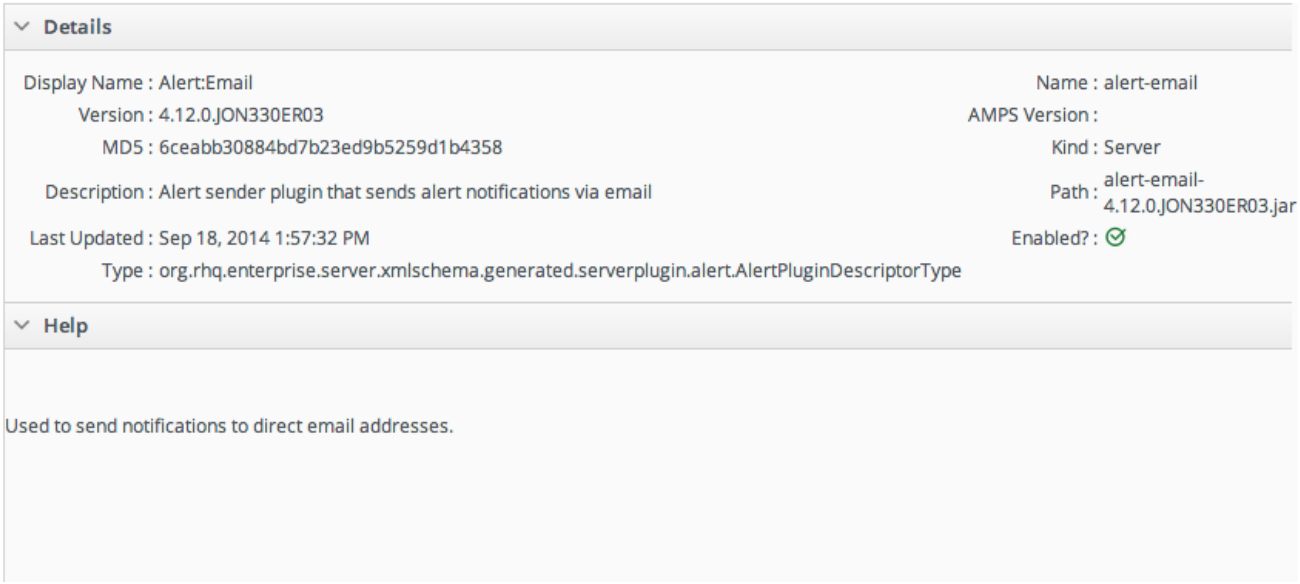
```
<alert-plugin
    name="alert-email"
    displayName="Alert:Email"
    xmlns="urn:xmlns:rhq-serverplugin.alert"
    xmlns:c="urn:xmlns:rhq-configuration"
    xmlns:serverplugin="urn:xmlns:rhq-serverplugin"
    package="org.rhq.enterprise.server.plugins.alertEmail"
    description="Alert sender plug-in that sends alert notifications via email"
>
```

次のセクションでは、アラートのヘルプテキストを提供します。

```
<serverplugin:help>
    Used to send notifications to direct email addresses.
</serverplugin:help>
```

ヘルプテキストは、UI のヘルプの説明セクションに表示されます。

図2.2 アラートヘルプテキスト



記述子の次のセクションは、alert-email プラグインについてより適しています。

```
<!-- startup & tear down listener, + scheduled jobs
<serverplugin:plugin-component />
-->
```

他のタイプのサーバー側のプラグインでは、このエリアには要素にスケジューリング情報が含まれることや、**<scheduled-jobs>** 要素に Java クラスを実装することもでき **<plugin-component>** ます。プラグインはタスクを実行しないため、アラート送信者を持つジョブをスケジュールする理由はなく、イベントの検出時にサーバーからメッセージを送信する方法を提供します。

グローバル優先度は、アラートの単一インスタンスに適用されるパラメーターを定義します。つまり、アラート送信者を使用するように設定されているすべての通知に適用されます。これらのグローバル設定パラメーターは XML ファイルで設定できますが、の説明に従って JBoss ON GUI を使用して編集す

ることもでき 「プラグイン設定プロパティーの設定」 ます。

alert-email プラグインには、通知に使用する送信者メールアドレス、メールサーバー、およびログインクレデンシャルが含まれます。

```
<!-- Global preferences for all email alerts -->

<serverplugin:plugin-configuration>
    <c:simple-property name="mailserver" displayName="Mail server address" type="longString"
        description="Address of the mail server to use (if not the default JBoss ON one )"
        required="false"/>
    <c:simple-property name="senderEmail" displayName="Email of sender" type="string"
        description="Email of the account from which alert emails should come from"
        required="false"/>
    <c:simple-property name="needsLogin" displayName="Needs credentials?"
        description="Mark this field if the server needs credentials to send email and give them below"
type="boolean"
        default="false"/>
    <c:simple-property name="user" type="string" required="false"/>
    <c:simple-property name="password" type="password" required="false"/>
</serverplugin:plugin-configuration>
```

デフォルトが設定されている場合は、設定自体にデフォルト値が指定されています。ただし、alert-email プラグインにはパラメーターのデフォルト値が設定されていないため、プラグイン設定の値はプラグインの設定ページで追加する必要があります。

<short-name> 要素はすべてのアラート送信元プラグインに必要です。これにより、アラート定義の通知領域のアラート送信者タイプに使用される名前を指定します。

```
<!-- How does this sender show up in drop downs etc -->

<short-name>Email</short-name>
```

この **<short-name>** 値はドロップダウンメニューおよびその他のユーザー指向のエリアで使用されるため、この値は displayName の値よりも人間が分かりやすくなります。

次のセクションでは、アラート通知の送信に使用するプラグインクラスを指定します。サーバー側のプラグインのコンポーネントは通常、 **org.rhq.enterprise.server.plugins.**pluginName。記述子の **package** 要素の **<plugin>** 要素から取得されます。alert-email プラグインの場合、完全なパッケージ名はです。 **org.rhq.enterprise.server.plugins.alertEmail**。 **EmailSender.java** クラス。

```
<!-- Class that does the actual sending -->

<plugin-class>EmailSender</plugin-class>
```

alert-email 記述子の最後のセクションは、通信設定の残りの半分を提供します。グローバルパラメーターは、JBoss ON サーバーがメール通知の送信に使用したメールサーバーなど、すべての通知に適用されるものを設定します。**<alert-configuration>** エントリーは、そのアラート送信者タイプを使用するすべての通知インスタンスに対して個別に設定される情報を提供します。これは **alert-email**、メールで送信された通知を受信するメールアドレスの一覧を可能にするフィールドです。

```
<!-- What can a user configure when defining an alert -->

<alert-configuration>
    <c:simple-property name="emailAddress" displayName="Receiver Email Address(es)"
```

```
type="longString"
        description="Email addresses (separated by comma) used for notifications."/>
    </alert-configuration>
```

バグを報告します。

2.3.2.2. Java リソース

Java ファイルの最初の部分はパッケージ名を識別し、そのタイプの送信者に必要なプロパティーをインポートします。メール送信者の Java ファイルには、アラート送信プラグインコンテナー、通知テンプレート、アラートを定義するその他のクラスでプルする設定が含まれます。

```
package org.rhq.enterprise.server.plugins.alertEmail;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import org.rhq.core.domain.alert.Alert;
import org.rhq.core.domain.alert.notification.SenderResult;
import org.rhq.enterprise.server.plugin.pc.alert.AlertSender;
import org.rhq.enterprise.server.util.LookupUtil;
```

EmailSender.java ファイルの残りのファイルは、通知設定およびプラグインのグローバル設定からデータをプルします。

開くには、送信側を設定します。

```
public class EmailSender extends AlertSender {

    @Override
    public SenderResult send(Alert alert) {
        String emailAddressString = alertParameters.getSimpleValue("emailAddress", null);
        if (emailAddressString == null) {
            return SenderResult.getSimpleFailure("No email address given");
        }
```

次の行はメールアドレスでプルされ、通知設定およびメールサーバーから通知を受信し、通知と送信者のメールアカウントをグローバル設定から送信します。

```
        List<String> emails = AlertSender.unfence(emailAddressString, String.class, ",");
        try {
            Set<String> uniqueEmails = new HashSet<String>(emails);
            Collection<String> badEmails = LookupUtil.getAlertManager()
                .sendAlertNotificationEmails(alert, uniqueEmails);

            List<String> goodEmails = new ArrayList<String>(uniqueEmails);
            goodEmails.removeAll(badEmails);

            SenderResult result = new SenderResult();
            result.setSummary("Target addresses were: " + uniqueEmails);
            if (goodEmails.size() > 0) {
                result.addSuccessMessage("Successfully sent to: " + goodEmails);
```

```
            }
            if (badEmails.size() > 0) {
                result.addFailureMessage("Failed to send to: " + badEmails);
            }
            return result;
        } catch (Throwable t) {
            return SenderResult.getSimpleFailure("Error sending email notifications to " + emails + ",
cause: "
                + t.getMessage());
        }

    }

    @Override
    public String previewConfiguration() {
        String emailAddressString = alertParameters.getSimpleValue("emailAddress", null);
        if (emailAddressString == null || emailAddressString.trim().length() == 0) {
            return "<empty>";
        }
        return emailAddressString;
    }
}
```

最後の部分は、電子メールアラートプラグインの応答、簡単な失敗または成功を設定します。

```
      catch (Exception e) {
        log.warn("Sending of email failed: " + e);
        return SenderResult.getSimpleFailure("Sending failed :" + e.getMessage());

      }
      return SenderResult.getSimpleSuccess("Send notification to " + txt + ", msg-id: " + status.getId());
    }
}
```

バグを報告します。

2.3.2.3. スキーマ要素

alert-email プラグインは（デフォルトのアラート送信者プラグインすべてとして）3 つのスキーマファイルを使用します。

- **rhq-configuration.xsd**すべての JBoss ON プラグインによって使用される
- **rhq-serverplugin.xsd**サーバー側のすべてのプラグインによって使用される
- **rhq-serverplugin-alert.xsd**: アラートプラグインによって使用されます。

これらのファイルのスキーマは、ビルドして拡張します。

この **rhq-serverplugin-alert.xsd** ファイルは、アラート送信元プラグインに必要です。追加のスキーマファイルを追加して他の要素を含めることができますが、アラートスキーマには、アラートセンダープラグインに非常に有用なスキーマ要素が複数含まれています。

表2.8 便利なアラートスキーマ要素

スキーマ要素	description	親タグ
alert-plugin	単一アラートプラグイン定義のルート要素。	なし。
short-name	UI で使用されるプラグインの表示名。	alert-plugin
plugin-class	プラグインの機能を実装するクラス。	alert-plugin
alert-configuration	アラートインスタンスの設定時に UI に表示する（デフォルト）設定要素。これには、ユーザー名、パスワード、URL、サーバー名、またはポートなどの一般的なデータが含まれます。	alert-plugin

バグを報告します。

第3章 サーバー側のプラグインの作成： 手順

3.1. ヒント： XSD アノテーションの確認

サーバー側のプラグインに要素を提供する XSD ファイルや **rhq-configuration.xsd rhq-serverplugin.xsd**、のようなタイプ固有のファイルが多数あります **rhq-serverplugin-alert.xsd**。

これらのスキーマファイルでは、異なるプロパティーと属性が定義されます。既存のデフォルトスキーマが使用できるようにアノテーションが付けられます。

たとえば、**<control>** 要素の場合は以下のようになります。

```
 <xs:element name="control" type="serverplugin:ControlType" minOccurs="0"
maxOccurs="unbounded">
            <xs:annotation>
               <xs:documentation>
                  Defines operations a user can invoke on the plugin component.
                  Typically, a user interface will allow a user to invoke these operations to
                  control the server plugin component during runtime.
               </xs:documentation>
            </xs:annotation>
 </xs:element>
```

XSD ファイルのアノテーションを使用して読み取り、プラグインの動作を計画し、設定にカスタムスキーマが必要であるかどうかを判断します。

バグを報告します。

3.2. サーバー側のプラグインの作成

サーバー側のプラグインは柔軟性があり、サーバー側のプラグイン（alerts、コンテンツ、GUI）の動作を構成するカテゴリーがありますが、一般的なプラグインは、ほぼすべてのサーバー機能に対応できます。

サーバー側のプラグインの作成の概要は次のとおりです。

1. **オプション**。RHQ Project GitHub ページからテンプレートとして使用するサンプルプラグインを手動でダウンロードします。
2. プラグインのタイプを特定します。各サーバー側のプラグインは、プラグインのタイプまたは機能に関連付ける高レベルのプラグインコンテナーによって管理されます。
3. **オプション**。プラグイン設定にカスタムスキーマを作成します。
4. sourceRoot ディレクトリーにカスタムプラグイン用のディレクトリーを作成し**/modules/enterprise/server/plugins** ます。例：

   ```
   mkdir myPlugin
   cd myPlugin/
   mkdir -p src/main/java/org/rhq/enterprise/server/plugins/myPlugin
   mkdir -p src/main/resources/META-INF
   ```

5. Maven ビルドに使用する同様の既存プラグインから **pom.xml** ファイルをコピーし、新しいプラグインをパッケージ化します。例：

```
cp ../alert-email/pom.xml .
```

6. プロパティーが新しいプラグインを反映するように、**pom.xml** ファイルを編集します。

注記

サーバー側のプラグインが使用する親リポジトリーの場所を必ず含めてください **https://repository.jboss.org/nexus/content/groups/public/org/rhq/rhq-enterprise-server-plugins-parent/**。例：

```
 <repositories>
  <repository>
   <snapshots>
    <enabled>true</enabled>
   </snapshots>
   <id>jboss</id>
   <name>JBoss Repository</name>
   <url>https://repository.jboss.org/nexus/content/groups/public/org/rhq/rhq-
enterprise-server-plugins-parent/</url>
  </repository>
...
  </repositories>
```

7. 特定のプラグインインスタンスを定義するプラグイン記述子を作成します。プラグイン記述子は、プラグインクラスからスケジュールされたジョブまですべて定義します。プラグイン記述子要素については、で説明されてい 「プラグイン設定（グローバルおよびローカルの両方）」 ます。

8. プラグインの Java クラスを実装します。

9. プラグインを構築します。Maven ビルドプロセス中に、プラグインファイルを検証できます。

```
mvn install
```

10. にあるように、プラグインをデプロイし 「サーバーサイドプラグインのデプロイ」 ます。サーバー側のプラグインが 1 台のサーバーにデプロイされると、クラウドの他のすべての JBoss ON サーバーに自動的に伝播されます。

バグを報告します。

3.3. SERVER-SIDE プラグインの検証

JBoss ON サーバーには、Maven ビルドプロセスの一部としてサーバー側のプラグインを検証する特別なクラスが含まれています。

検証 とは、ビルドプロセスでサーバー側のプラグイン記述子が許容可能で、完了したことを確認することを意味します。すべてのサーバー側のプラグインで、以下のような処理がないかチェックされます。

- XML は適切に作成され、設定されたサーバープラグイン XML スキーマを使用して検証されます。
- プラグインコンポーネントが指定されている場合、そのクラスはプラグイン JAR で見つかり、インスタンス化できます。

- スケジュールされたジョブがすべて適切に設定されている。
- プラグインには有効なバージョンがあります。
- プラグイン設定が正しく宣言されている。

注記

プラグインの検証は、JAR ファイルの作成時に Maven ビルドプロセスで実行されます。プラグイン JAR が別のシステムまたは別のマシンを使用して構築されている場合は、検証は実行されません。

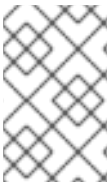
Maven を使用してビルド時にプラグインを自動的に検証するには、検証するプラグインをバリデーターの **pom.xml** 設定ファイルに追加する必要があります。

1. sourceRoot **ディレクトリー/modules/enterprise/server/plugins/validate-all-serverplugins/** で **pom.xml** ファイルを開きます。
2. カスタムサーバー側の JAR ファイルを参照するファイルに **<pathelement>** 行を追加します。例：

```
<pathelement location="../myPlugin/target/myPlugin.jar" />
```

3. プラグインを構築します。

```
mvn install
```



ヒント

RHQ ソースコードには、カスタムプラグインインフラストラクチャーで使用できるバリデーターユーティリティーが含まれます。これは、以下に含まれます。**org.rhq.enterprise.server.plugin.pc.ServerPluginValidatorUtil** クラス。

バグを報告します。

3.4. サーバーサイドプラグインのデプロイ

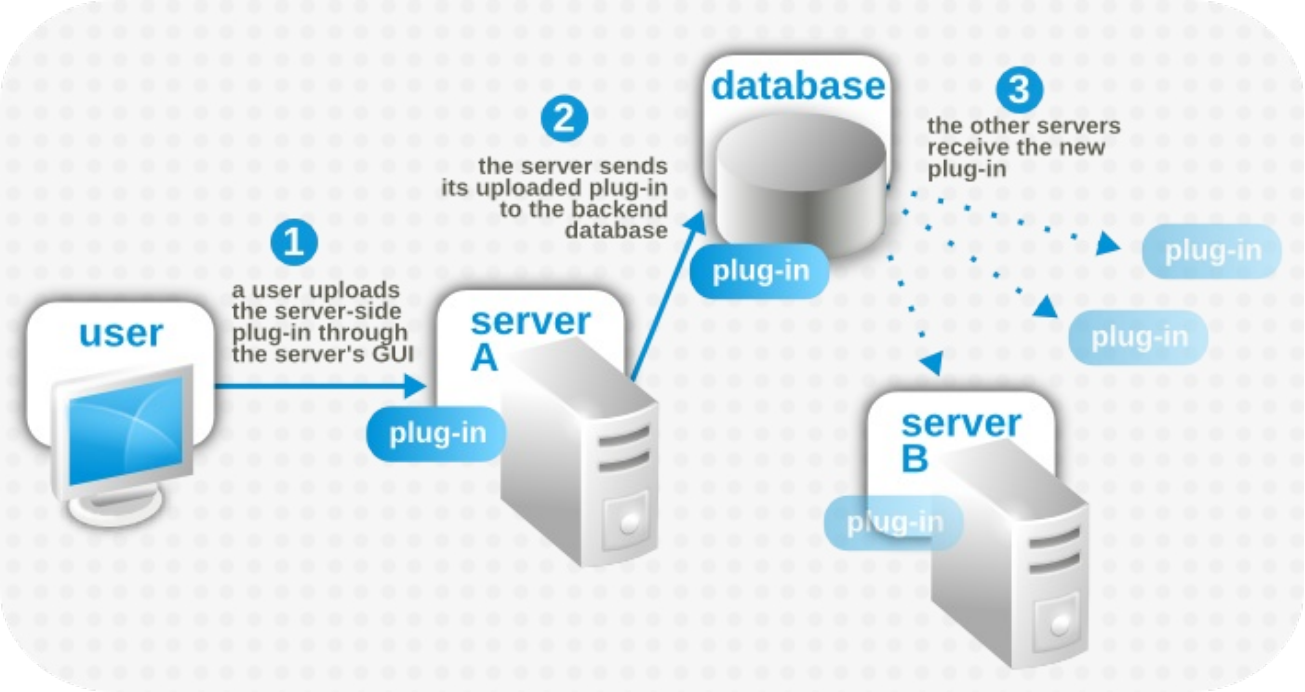
サーバー側は、以下の 2 つのいずれかの方法でデプロイされます。

- プラグイン JAR ファイルをサーバーのルートディレクトリー（ローカル）の sourceRoot **フォルダー/plugins/** にコピーします。
- Web インターフェース（リモート）でプラグイン JAR ファイルをアップロードします。

サーバー側のプラグインはホットデプロイされるため、JBoss ON サーバーを再起動せずにデプロイされるとすぐにアクティブになります。すべてのサーバー側のプラグインはグローバルにデプロイされ、サーバークラウド間で自動的に伝播されます。各サーバーの設定は定期的にポーリングされます（サーバープロパティーファイルに定義される間隔）。

デフォルトでは、プラグイン記述子の設定が明示的に有効にされない限り、すべてのサーバー側のプラグインが自動的に有効（そのためアクティブ）されます。プラグインがデプロイおよび有効化されると、インフラストラクチャーの他の JBoss ON サーバーに自動的に伝播されます。

図3.1 サーバー側のプラグイン伝搬



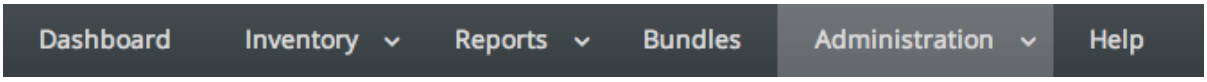
サーバー側のプラグインには、以下の 2 つの状態があります。

- デプロイ済みおよび有効化
- デプロイ済みおよび無効

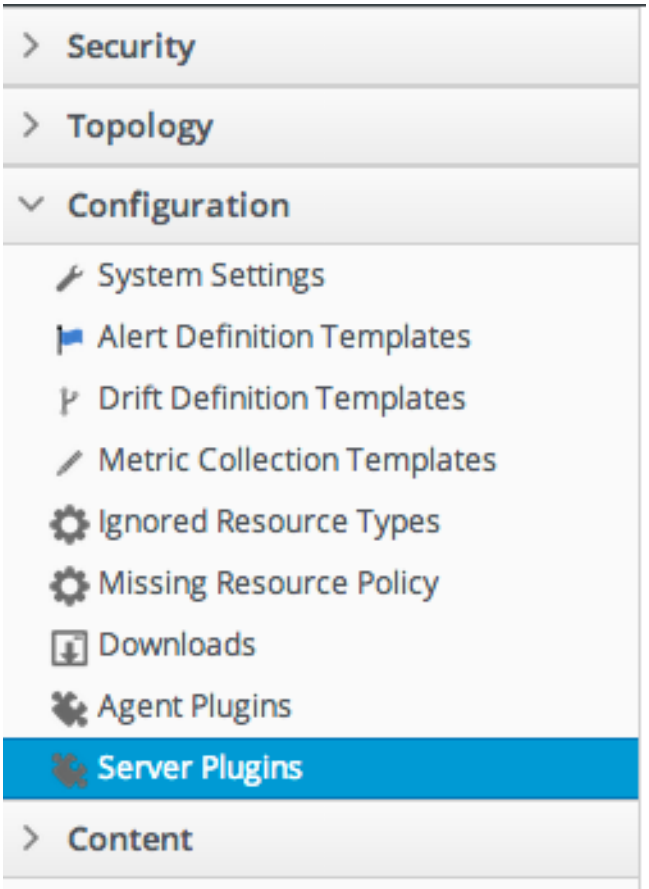
バグを報告します。

3.4.1. サーバー側のプラグインをリモートでデプロイ

1. トップメニューで、**Administration** タブをクリックします。



2. 左側のナビゲーションバーの **Configuration** ボックスで、**Server Plugins** リンクをクリックします。



3. ページの下部にある **Upload Plugin** セクションまでスクロールします。
4. **Browse...** ボタンをクリックして、プラグイン JAR ファイルの場所を参照します。
5. プラグインをデプロイするには、**Upload** ボタンをクリックします。

あるサーバーにアップロードしたプラグインは自動的にデプロイされ、数分間以内に他のすべての JBoss ON サーバーにデプロイおよび登録されます。

バグを報告します。

3.4.2. サーバー側のプラグインをローカルでデプロイ

各サーバーインストールにはトップレベルの plugins/ ディレクトリーがあります。サーバーは定期的にこのディレクトリーをポーリングします。新規または更新された JAR ファイルはすべてサーバー設定の適切なディレクトリーにコピーされ、元の JAR ファイルは plugins/ ディレクトリーから削除されます。

JAR ファイルが JBoss ON サーバーと同じホストマシンにある場合、JAR ファイルはその sourceRoot/**plugins**/ ディレクトリーにコピーでき 、サーバーによってデプロイされます。

バグを報告します。

3.5. SERVER-SIDE プラグインの更新

更新されたプラグイン JAR ファイルをデプロイするとサーバー側のプラグインを更新できます。プラグイン記述子には、プラグインパッケージのバージョン番号を含めることができます。サーバーはこのバージョン番号（JAR ファイルの **META-INF/MANIFEST.MF** ファイルにある **Implementation-Version** 設定）を使用して、プラグインの後続のバージョンを特定し、クラウドの JBoss ON サーバーのプラグインを更新します。

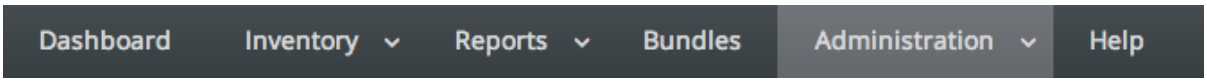
バグを報告します。

3.6. サーバー側のプラグインの無効化

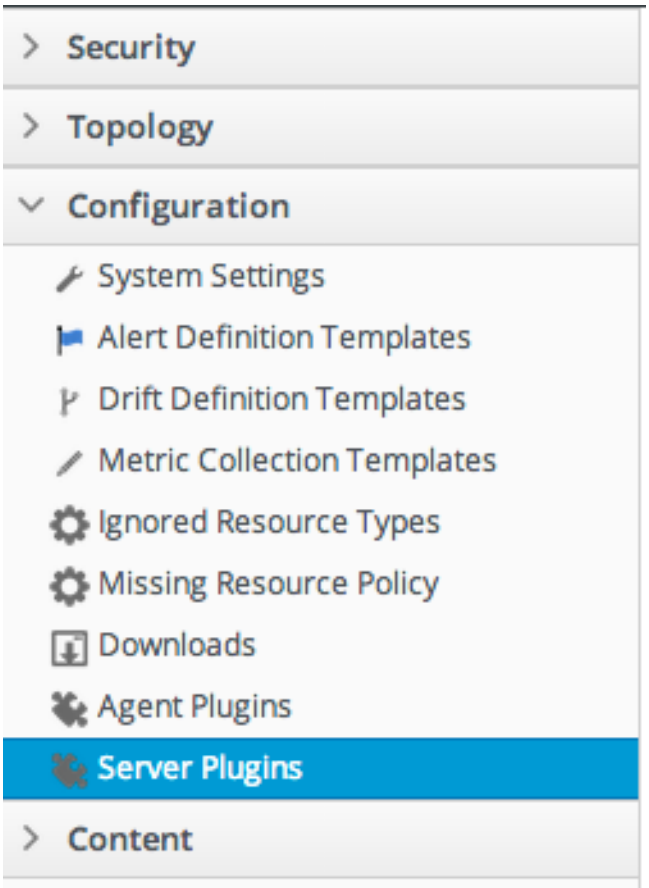
指定がない場合は、すべてのプラグインがデプロイ時に有効になります。プラグインが無効になっていると、クラウドのすべての JBoss ON サーバーの設定にリストされているままとなり、サーバーでロードまたは起動できなくなります。

プラグインを無効にするには、以下を実行します。

1. トップメニューで、**Administration** タブをクリックします。

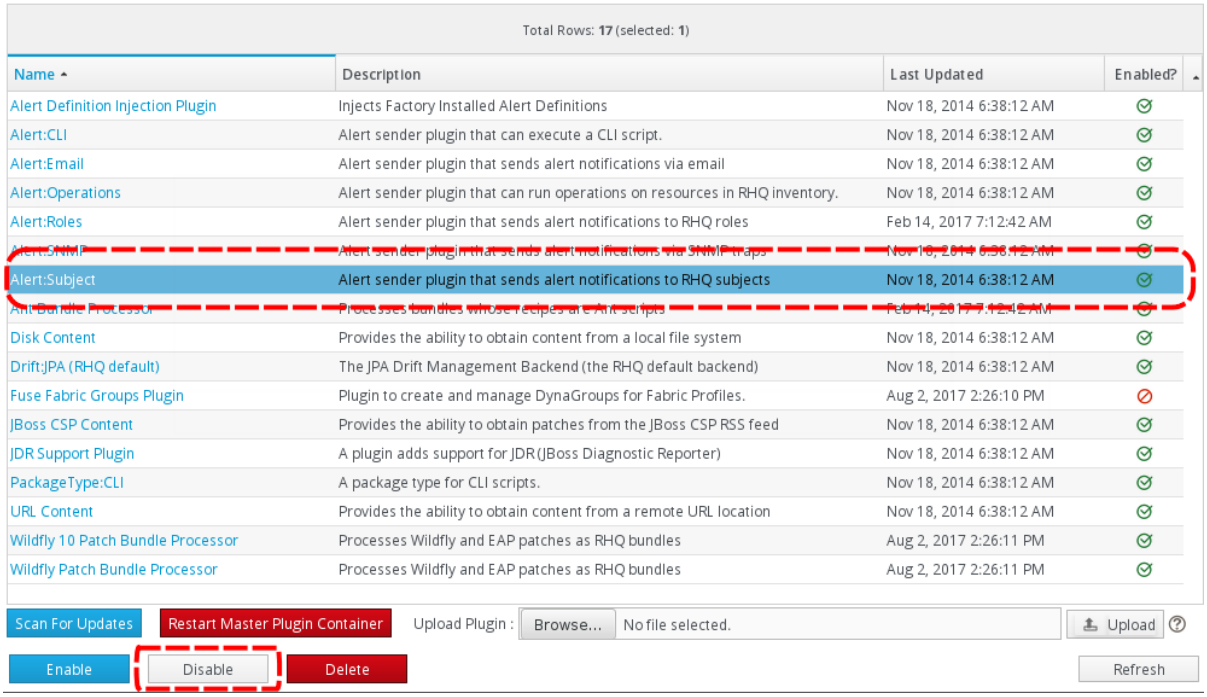


2. 左側のナビゲーションバーの **Configuration** ボックスで、**Server Plugins** リンクをクリックします。



3. 無効にするサーバー側のプラグインを選択します。

4. **DISABLE** ボタンをクリックします。



サーバープラグイン管理ページの **'Enabled?'** フィールドは、プラグインが有効かどうかを示します。

無効にしたプラグインは後で再度有効にするには、そのプラグインを選択し、**ENABLE** ボタンをクリックします。

バグを報告します。

3.7. サーバー側のプラグインコンテナーの再起動

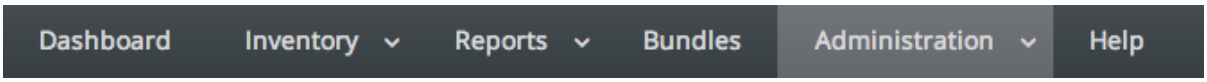
サーバー側のプラグインの各タイプは、対応するプラグインコンテナーによって制御されます。各プラグインコンテナーは、マスタープラグインコンテナーによって制御されます。プラグインコンテナーは、プラグインを読み込み、起動、および停止します。

開発者は新しいサーバー側のプラグインをホットデプロイするため、プラグインのパフォーマンスを確認するためにプラグインコンテナーを再起動すると便利です。これは、マスタープラグインコンテナーを再起動して行います。

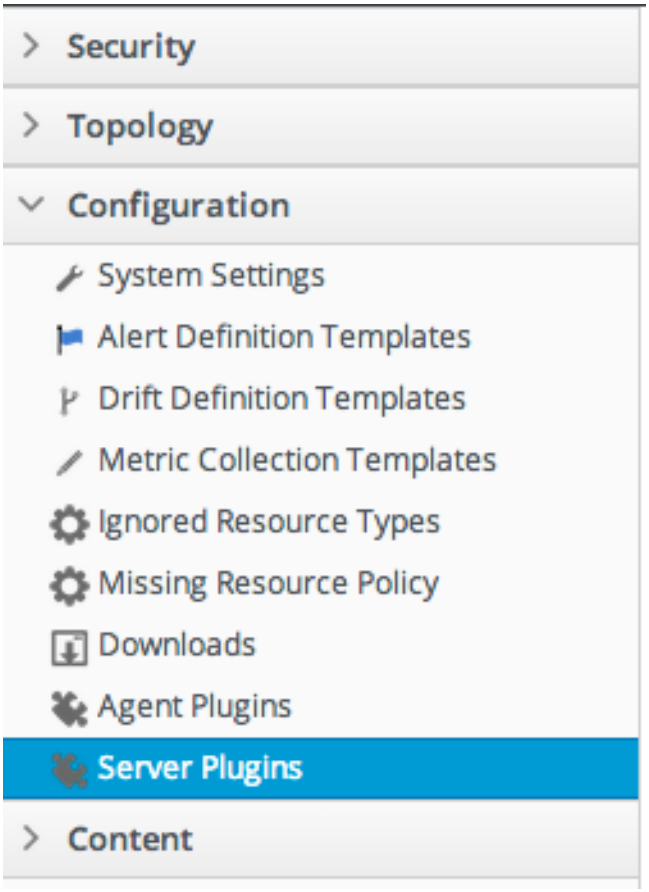
注記

サーバー側のプラグインに関するその他のすべてのアクションは **クラウドで行われ** ます。新しいプラグインが追加されると、クラウド全体に追加されます。ただし、プラグインコンテナーは、ローカルでタスクを実行します。プラグインコンテナーを再起動し、JBoss ON サーバーが Web インターフェース（コマンドのローカルであるサーバー）をホストしているマスタープラグインコンテナーを再起動し、そのマスタープラグインコンテナーのみを再起動します。

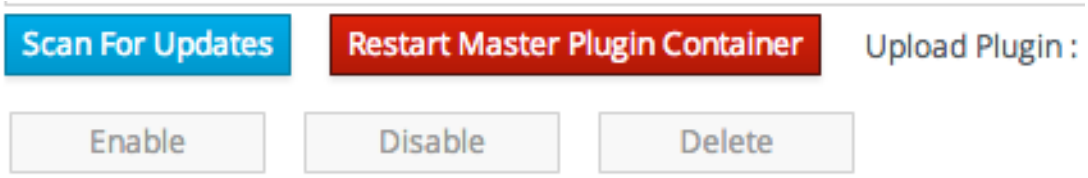
1. トップメニューで、**Administration** タブをクリックします。



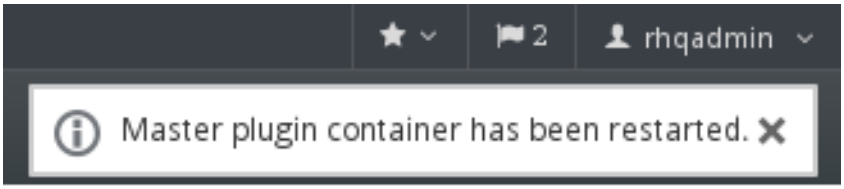
2. 左側のナビゲーションバーの **Configuration** ボックスで、**Server Plugins** リンクをクリックします。



3. テーブルの下部までスクロールし、**RESTART MASTER PLUGIN CONTAINER** ボタンをクリックします。



4. 再起動プロセスが実行されると（問題が発生しないことを前提として）、右上隅に成功メッセージが表示されます。



バグを報告します。

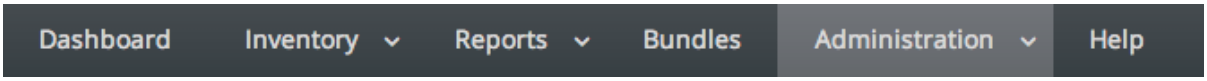
3.8. プラグイン設定プロパティーの設定

デフォルトのサーバー側のプラグインやカスタムプラグインによっては、管理者はプラグインインスタンスの特定の設定プロパティーを定義することができます。使用できるプロパティーはプラグインの **rhq-plugin.xml** ファイルで定義され、値は JBoss ON UI に指定されます。

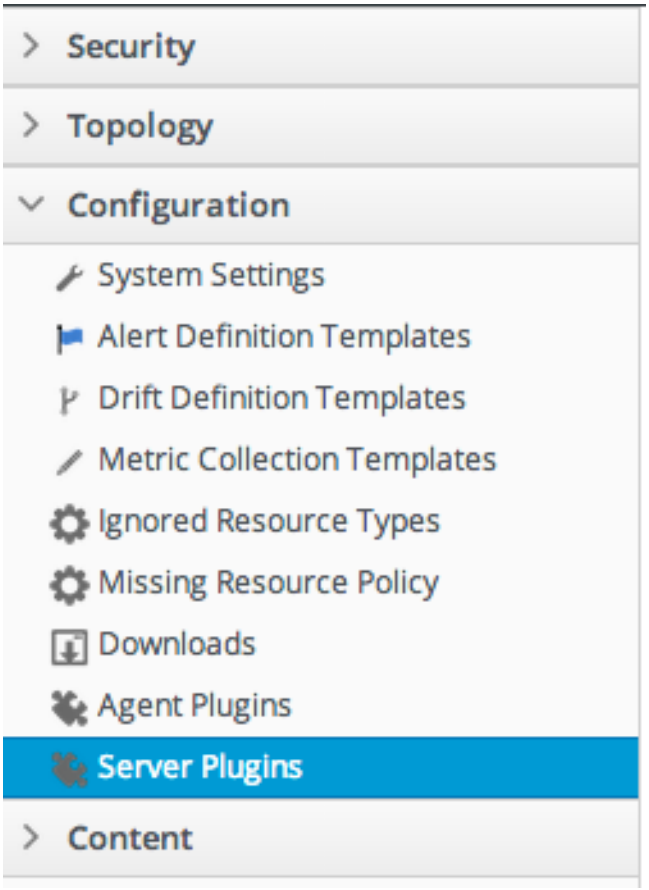
記述子ファイルは、そのプラグインのすべてのインスタンスに適用される特定の設定パラメーターを定義できます（記述子パラメーターは、サーバー側 「記述子および設定」 のプラグインおよびエージェントプラグインについてで説明されています 「記述子および設定」 ）。記述子はデフォルト値を設定

しても、これらのフィールドを空白のままにすることもできます。いずれの方法でも、JBoss ON の Web UI でグローバルプラグイン設定パラメーターを設定または変更できます。

1. トップメニューで、**Administration** タブをクリックします。



2. 左側のナビゲーションバーの **Configuration** ボックスで、**Server Plugins** リンクをクリックします。

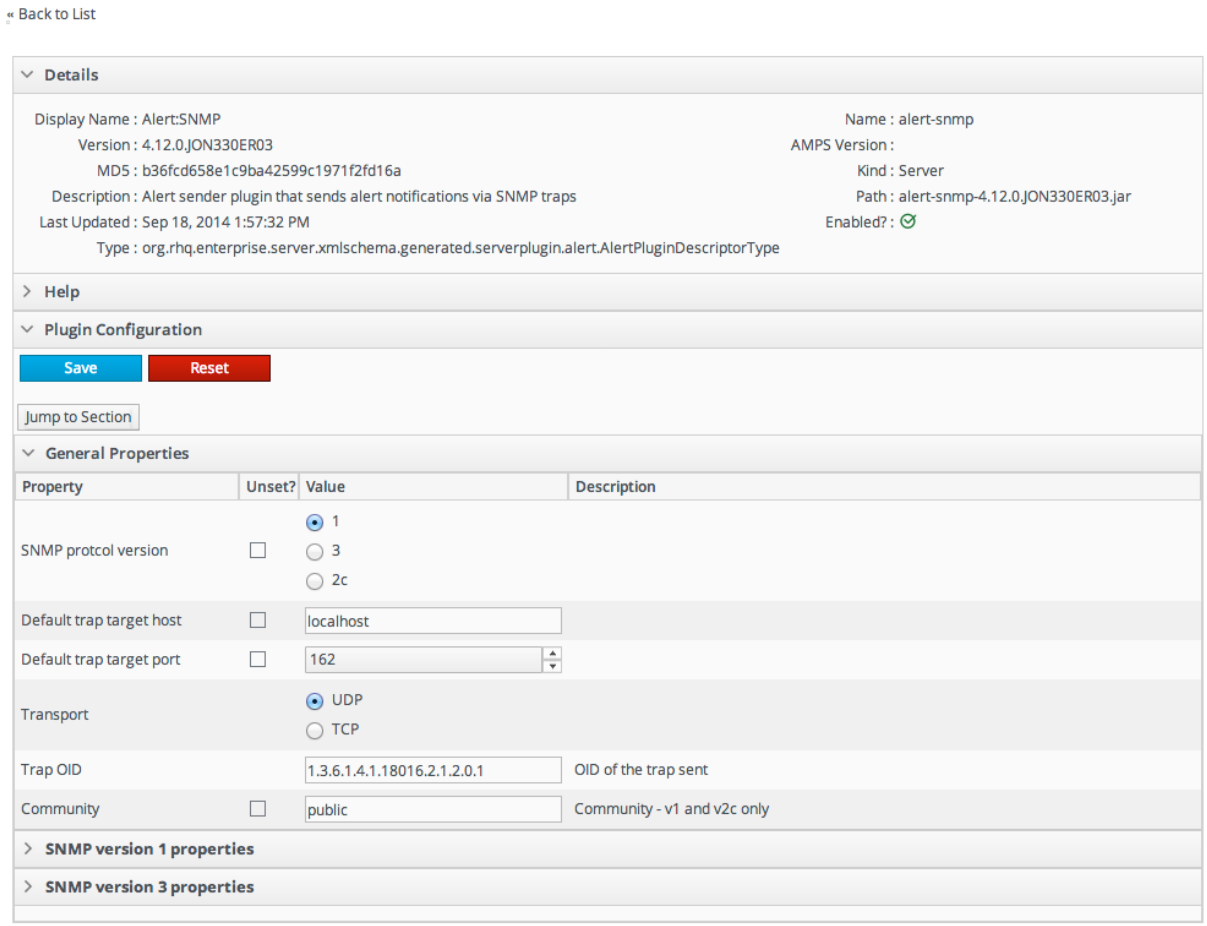


3. テーブルでサーバー側のプラグインの名前をクリックします。

4. プラグインの詳細ページの間に **Plugin Configuration** セクションを展開し、設定プロパティーにアクセスします。

注記

必要に応じて、チェックボックスの選択を **Unset** 解除して、編集するフィールドをアクティベートします。



5. 設定セクション上部の **SAVE** ボタンをクリックします。

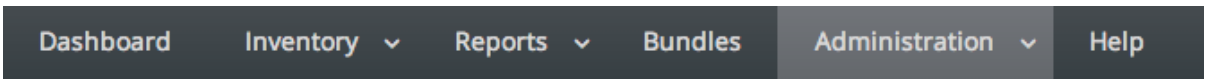
バグを報告します。

3.9. サーバー側のプラグインの削除

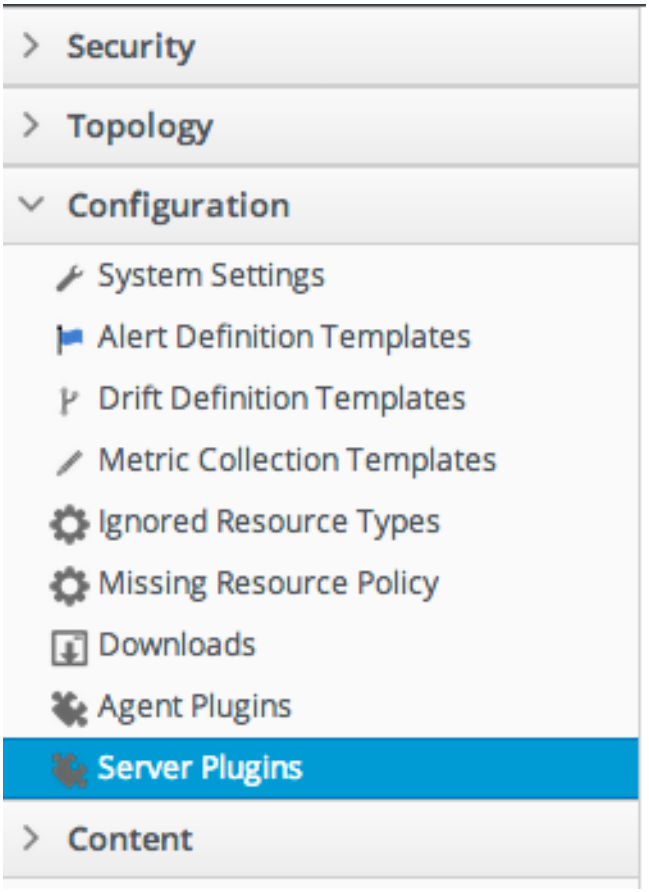
警告

プラグインを削除すると、プラグインに関連付けられたリソースタイプおよびリソースがすべて削除されます。この操作は元に戻すことはできません。

1. トップメニューで、**Administration** タブをクリックします。

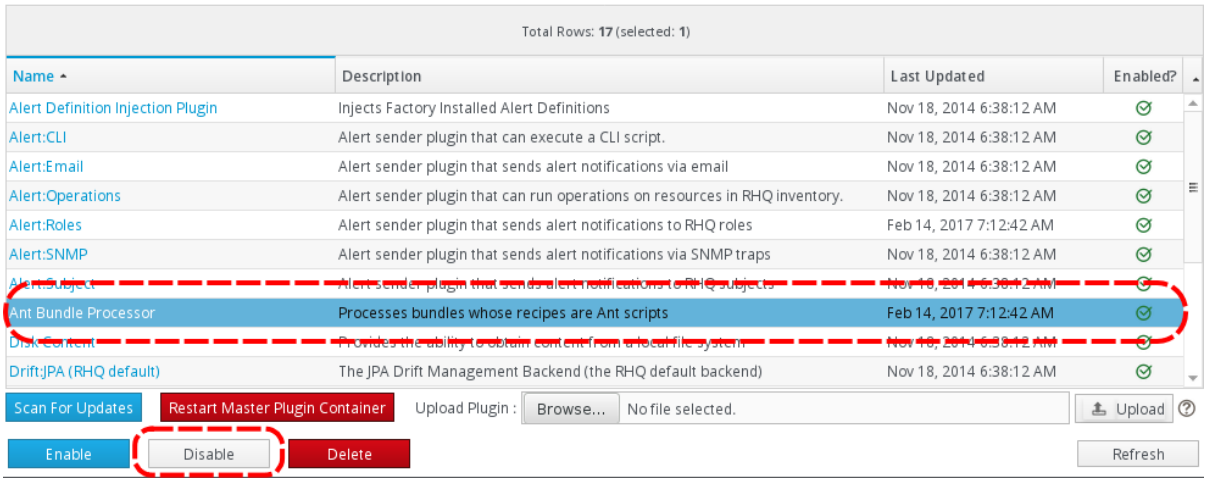


2. 左側のナビゲーションバーの **Configuration** ボックスで、**Server Plugins** リンクをクリックします。



3. 削除するプラグインを選択します。

4. **Delete** ボタンをクリックします。



バグを報告します。

第4章 エージェントプラグインの作成： 背景

エージェントプラグインは、エージェントがリソースと対話する際に持つ制御機能を拡張します。これには、リソースに対する監視、操作、設定の追加が含まれます。

バグを報告します。

4.1. エージェントプラグインの ADVANCED MANAGEMENT PLUG-IN SYSTEM(AMPS)について

JBoss ON のエージェントリソースプラグインには、記述方法に関する特定のパターンまたはシステムがあります。これは **Advanced Management Plug-in System** (AMPS)と呼ばれます。AMP は JBoss ON のコア API で定義されます。

エージェントプラグインは、5 つの部分から構成されます（まとめて AMPS システムです）。

- **エージェントのプラグインコンテナー。**プラグインには JBoss ON エージェント内で実行され、デプロイされたすべてのリソースプラグインのマネージャーが提供されます。

  プラグインコンテナーは、実際にリソースプラグインのライフサイクルを管理するものです。エージェントはプラグインコンテナーを開始し、プラグインコンテナーはリソースプラグインを起動します。プラグインコンテナーは、リソースプラグインに対してすべてのクラスロード、スレッド、および実行も処理します。

  プラグイン開発者は、プラグインコンテナーと対話する必要がありません。プラグインが適切なコンポーネントおよび有効なプラグイン記述子で記述されている限り、エージェントはこのリソースを管理できます。

- **ドメインオブジェクト**。これは、プラグインの個別のオブジェクト（特にリソース、リソースタイプ、および設定）を定義します。AMPS の他のすべての要素は、ドメインオブジェクトを使用してリソース要素を定義します。

  ドメインオブジェクト内で最大規模の API セットの 1 つが **configuration** です。設定 API は、プラグイン設定からリソースへの接続から操作引数への接続まで、設定プロパティーのセットが必要な場所に使用されます。

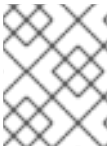
- **プラグインコンポーネント。**これらのコンポーネントは、エージェントプラグインによって使用される実際のコンポーネントインターフェースと、プラグインがサポートする **ファセット** を定義します。

  プラグインコンポーネントはパブリック API です。

  AMPS 内のこの要素は、プラグインライターが使用する部分です。これには、プラグイン作成者がリソースプラグインに実装するインターフェースが含まれます。

- **ネイティブシステム**オペレーティングシステムの情報からリソースを監視または管理するのに必要な情報が多くあります。ネイティブシステムは、オペレーティングシステム情報への JNI またはネイティブアクセスを提供し、プロセステーブルから情報をプルしたり、外部プログラムを実行したり、システムメトリックを収集することもできます。

- **リソースプラグイン。**JBoss ON には、すでにリソースプラグインのセットが定義されています。各リソースプラグインは、特定の製品（アプリケーションとサーバー、サービス、またはプラットフォーム）を管理します。これらのプラグインはエージェントのプラグインコンテナーにロードされ、API で定義されるプラグインコンポーネントを実装します。



注記

エージェントプラグインは JBoss ON ドメインおよびネイティブシステム API を使用して、オブジェクトと通信層をそれぞれ定義できます。

バグを報告します。

4.2. エージェントプラグイン設定の内訳

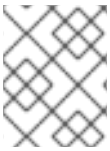
エージェントには固定機能がありません。監視機能からインベントリー可能なリソースまで、それらの機能はプラグインにより定義されます。

そのコアでは、エージェントプラグインは単一の JAR ファイルと XML プラグイン記述子ファイル（**META-INF/** ディレクトリー**rhq-plugin.xml** 内）で構成されます。

エージェントプラグイン記述子と併せて、JAR ファイルパッケージで定義された各プラグインの Java ファイルには最大 3 つの異なるタイプの Java ファイルがあります。

- プラグイン機能のすべてのコードを含むプラグインコンポーネントファイル
- プラグインで定義されたリソースの検出プロセスを設定する ***Discovery.java** ファイル
- リソースで収集できるイベントを ***EventPoller.java** 定義する

Java ファイルの定義は、プラグイン **rhq-plugin.xml** 記述子のプラグインの設定を密に追跡します。



ヒント

プラグインジェネレーターでエージェントプラグインテンプレートを生成すると、プラグインの作成に役立つ TODO マーカーでファイルが作成されます。

バグを報告します。

4.2.1. スキーマファイル

エージェントプラグインは、その XML プラグイン記述子ファイルのメタデータおよび設定を使用して定義されます。記述子で **使用できる** 設定要素は、エージェントプラグインの XML スキーマ定義(XSD)ファイルで定義されます。

すべての JBoss ON プラグイン（エージェントプラグインおよびサーバー側のプラグイン）は、この **rhq-configuration.xsd** ファイルを使用して利用可能な基本的な設定オプションを定義します。

エージェントもファイルを使用します。この **rhq-plugin.xsd** ファイルは **rhq-configuration.xsd** スキーマを拡張し、リソース関連のプラグイン専用の追加要素を追加します。

注記

両方の XSD ファイル内の特定の要素の詳細は、XSD ファイル自体のコメント（**<xs:annotation>** 項目）で確認できます。

この **rhq-configuration.xsd** ファイルは、すべての JBoss ON プラグインで使用できるスキーマを提供します。 **rhq-configuration.xsd** ファイルは **ソース**にあり**/modules/core/client-api/src/main/resources**ます。

rhq-configuration スキーマで定義される最も一般的な要素は、やのようなプラグインの設定値の設定に関連 **<simple-property>** し **<map-property>**ます。

表4.1 rhq-configuration.xsd スキーマ要素

要素	description
configuration-property	ユーザー定義の設定のプラグインに設定属性を追加する場合。
simple-property	デフォルト設定値を設定する場合。
オプション	プロパティーの値が列挙された一覧(false)から取得されるか、またはユーザーが定義する値(true)であるかを設定します。

rhq-configuration.xsd file また、必須属性や **name** オプション **displayName** 属性など、プラグイン記述子に使用できる最も一般的なフラグも定義します。

表4.2 rhq-configuration.xsd スキーマ属性

attribute	description
Name	**必須。** プラグインの一意の名前を指定します。
displayName	GUI のプラグインに使用する名前を指定します。これを指定しない場合には、name の値が使用されます。
description	プラグインの簡単な説明を提供します。

は **rhq-plugin.xsd**、エージェントプラグイン専用のすべてのスキーマ要素を提供します。 **rhq-plugin.xsd** ファイルは **ソース/modules/core/client-api/src/main/resources** ディレクトリーにあります。

rhq-plugin.xsd ファイルの最も一般的な要素はに記載されてい 表4.3「rhq-plugin.xsd スキーマ要素」ます。

表4.3 rhq-plugin.xsd スキーマ要素

要素	description
plugin	プラグイン記述子のルート要素が含まれます。
Dependencies	このプラグインが必要または拡張する他のプラグインを特定します。

要素	description
プラットフォーム、サーバー、サービス	エージェントプラグイン内で定義されるリソースのタイプを特定します。**<platforms>** はトップレベルの要素ですが、プラットフォームやその他のサーバー **<servers>** およびサービスリソースの子として **<services>** 追加されます。
メトリクス	そのリソースタイプに対して収集できるメトリクスを定義するプラットフォーム、サーバー、またはサービス内の要素。 このリソース要素の子要素および属性は **rhq-plugin.xsd** ファイルにリストされます。 値の配列など、大きなデータ構造の一部を形成する値は、監視の前に個別の値にデコンストラクトする必要があります。
イベント	リソースがイベントをサポートするかどうかを定義するプラットフォーム、サーバー、またはサービス内の要素。イベントが含まれる他の設定プロパティーはありません。イベント自体はリソースのログファイルから累積されます。
bundle-target	リソースにバンドルをデプロイするかどうかおよび設定します。 このリソース要素の子要素および属性は **rhq-plugin.xsd** ファイルにリストされます。
drift-definition	リソースに対してドリフトの監視を実行するかどうかおよび方法を設定します。 このリソース要素の子要素および属性は **rhq-plugin.xsd** ファイルにリストされます。
resource-configuration	リソースタイプの設定プロパティーを定義します。 このリソース要素の子要素および属性は **rhq-plugin.xsd** ファイルにリストされます。
operation	そのリソースタイプで実行できる操作を定義します。 このリソース要素の子要素および属性は **rhq-plugin.xsd** ファイルにリストされます。
コンテンツ	リソースタイプにアップロードまたはデプロイできるパッケージのタイプを設定します。 このリソース要素の子要素および属性は **rhq-plugin.xsd** ファイルにリストされます。

に定義さ **rhq-plugin.xsd** れている属性のほとんどは、プラグイン記述子のルート要素で使用される include フラグです。これらは、エージェントプラグインのリリースや更新を制御する管理属性を追加します。

表4.4 rhq-plugin.xsd スキーマ属性

attribute	description
パッケージ	プラグインパッケージ名を設定する場合。
version	プラグインのバージョンを設定する場合。これは、OSGi と互換性のある形式である必要があります。
ampsVersion	このプラグインが必要とするエージェントプラグインシステムバージョンの場合。これは、OSGi と互換性のある形式である必要があります。
pluginLifecycleListener	プラグインを初期化およびシャットダウンするリスナー。
検出	検出スキャンでリソースタイプを検出するかどうかを設定します。このフラグは、親リソースで検出される子リソースには不要です。

rhq-plugin.xsd ファイルに多くの要素および属性が設定されます。それぞれの項目の説明は、項目の **<xs:annotation>** タグのテキストで説明されています。

バグを報告します。

4.2.2. 記述子および設定

プラグイン記述子には、プラグインおよび設定したリソースに関するすべてを記述するメタデータが含まれます。記述子は、エージェントプラグインが定義するリソースタイプのタイプと対話を記述します。

プラグイン記述子には、リソースの名前、サポートされるリソースバージョン、リソース階層の合計（リソースの階層、リソースの子）、エージェントがリソースへの接続に使用する設定プロパティー、およびエージェントが管理できるリソースに関連するすべての監視メトリクス、操作、およびイベントに関する情報が含まれます。

プラグイン記述子には、プラグイン自体に関する情報も含まれます。

プラグイン記述子のリソース定義は、プラットフォーム、サーバー、またはサービスです。複数のリソースを単一のプラグイン記述子に定義できます。1 つのリソースはルート(parent)要素であり、残りのリソースはその子です。

プラグイン記述子は XML ファイルであるため、明確で構造化されたスキーマ定義に従います。スキーマ定義は、プラグイン記述子がリソースの管理インターフェースを JBoss ON に公開できるようにするものです。

プラグイン記述子は、最低でもリソースタイプを定義します。これまでは、JBoss ON が管理することのできるリソースのさまざまな側面が定義されます。

- プラグインがサポートするリソースタイプ（サーバーおよびサービス）の名前
- エージェントのプラグインコンポーネントがリソースへの接続に使用するすべての設定

- リソースの監視に使用するメトリクス（セキュリティー定義）は、リソース自体が発行するデータの種類によって異なります。
- リソースで起動できる操作のセット。これは通常、開始および停止操作ですが、スクリプトの実行などのアプリケーション固有の操作や他のアクションを含めることができます。
- リソースの実際の設定で編集できるリソース設定値。

  プラグイン設定は、コンポーネントに対し、リソースへの接続方法を指示します。一方、リソース設定は、外部で編集できるリソース自体の設定です。
- リソース階層の一部である子リソース。たとえば、JBoss サーバーにはデータソースサービスが実行されているため、データソースサービスは JBoss サーバーの子リソースプラグインで定義されます。

バグを報告します。

4.2.2.1. リソースの種類、メタデータ、およびプラグインの設定

エージェントプラグインの top 要素は **<plugin>** 要素です。

```
<plugin name="JMX"
        displayName="Generic JMX"
        package="org.rhq.plugins.jmx"
        description="Supports management of JMX MBean Servers via various remoting systems."
        ampsVersion="2.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="urn:xmlns:rhq-plugin"
        xmlns:c="urn:xmlns:rhq-configuration">
```

この要素のいくつかの属性は重要です。

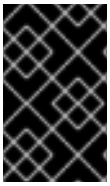
- **name** また、プラグインの内部および GUI 名 **displayName** を付与します。
- **ampsVersion** プラグインのバージョン番号を指定します。
- **package** プラグインのコンポーネントによって使用されるクラスの名前を指定します。

プラグイン記述子の次の要素は、プラグインによって定義されるルートリソースを定義します。これは **<platform>**、**<server>**、またはになり **<service>**ます。

1 つまたは複数のリソースをプラグイン記述子に定義できます。プラグイン記述子は、これらのリソースタイプを定義するだけでなく、これらのタイプを親子階層で編成します。たとえば、JBoss EJB サービスは JBoss サーバー内でのみ実行されるため、EJB サービスリソースタイプは JBoss サーバーリソースタイプの子タイプを論理的にする必要があります。

階層は、リソース定義 **<server>** （または他の **<service>**） **<service>** リソース定義内でネストすることで定義されます。

```
<server name="JMX Server" discovery="JMXDiscoveryComponent" class="JMXServerComponent"
        description="Generic JMX Server"
 supportsManualAdd="true" createDeletePolicy="neither">
```



重要

リソースプラグインの編集時に**は、リソースタイプの名前を変更しないでください。**これにより、古いバージョンのプラグインを使用していたリソースとの後方互換性が保たれます。

属性の 2 つは、プラグインに関連付けられた Java リソースファイルに関連するものです。

- **discovery** リソースタイプの特定に使用される検出コンポーネントを特定します。
- **class** プラグインの実際のコードが含まれるプラグインコンポーネントを特定します。

属性の 2 つは、そのタイプのリソースをインベントリーに追加する方法を定義します。

- **supportsManualAdd** 管理者は、リソースをインベントリーに追加できます。
- **createDeletePolicy** インベントリーから子を手作業で追加または削除できるかどうかを設定します。

プラグインの設定に関連する最後の部分は（オプション）プラグインの設定プロパティーです。それらは柔軟であり、許可される値やテンプレートにデフォルト設定を定義する場合でも、プラグインのリソースタイプに一致するリソースの特定またはセットアップに必要な情報をすべて定義できます。

```
<plugin-configuration>
  <c:list-property name="Servers">
    <c:map-property name="OneServer">
      <c:simple-property name="host"/>
      <c:simple-property name="port">
        <c:integer-constraint
          minimum="0"
          maximum="65535"/>
      </c:simple-property>
      <c:simple-property name="protocol">
        <c:property-options>
          <c:option value="http" default="true"/>
          <c:option value="https"/>
        </c:property-options>
      </c:simple-property>
    </c:map-property>
  </c:list-property>
</plugin-configuration>
```

ポートには制約があるため、GUI は 0 から 65535 までの入力を検証できます。プロトコルは、ドロップダウンメニューの一覧からデフォルト値の HTTP で選択できます。

プロパティーには 3 つのタイプがあります。

- **<simple-property>**1 つのキーと値のペアを定義する
- **<map-property>**。以下に示すように、単一のエンティティーに関連する複数のキーと値のペアを定義します。 **java.util.Map** 概念
- **<list-property>**プロパティーのリストが含まれる

<simple-property> 要素のクルーフの両方 **<map-property>** を **<list-property>** 定義して定義します。また、これらのプロパティーは **<group>** 要素の下で正式にグループ化できます。**<group>** 要素を使用すると、UI に折りたたみ可能な設定が作成されます。

テンプレートは、設定プロパティーの一部を事前に設定することができます。

```
        <c:template name="JDK 5" description="Connect to JDK 5">
          <c:simple-property name="type"
default="org.mc4j.ems.connection.support.metadata.J2SE5ConnectionTypeDescriptor"/>
          <c:simple-property name="connectorAddress"
default="service:jmx:rmi:///jndi/rmi://localhost:8999/jmxrmi"/>
        </c:template>
```

バグを報告します。

4.2.2.2. 検出およびプロセススキャン

JBoss ON の主な機能はインベントリーです。各リソースはそのインベントリーに存在する必要があるため、プラグイン記述子は、リソースの検出方法とインベントリーへの追加方法を定義する必要があります。これは、discovery コンポーネントで行われます。

<plugin> 要素には、リソースプラグインの検出 Java ファイルを特定する **discovery** 属性があります。（プラグインに複数のリソースが定義されている場合は、複数の検出コンポーネントがあります。）

エージェントプラグインがプラグインジェネレーターを使用して生成されると、検出要件を追加するための適切なテンプレートが作成されます。Discovery コンポーネントには、リソースインスタンスを特定し、そのリソースに固有の識別子を割り当てる情報が必要です。

```
/**
 * Discovery class
 */
public class testDiscovery implements ResourceDiscoveryComponent
,ManualAddFacet
{

   private final Log log = LogFactory.getLog(this.getClass());

    /**
     * Do the manual add of this one resource
     */
    public DiscoveredResourceDetails discoverResource(Configuration pluginConfiguration,
ResourceDiscoveryContext context) throws InvalidPluginConfigurationException {

        // TODO implement this
        DiscoveredResourceDetails detail = null; // new DiscoveredResourceDetails(
//              context.getResourceType(), // ResourceType
//          );

        return detail;
    }
}
```

リソースインスタンスの特定は、インベントリーを維持する上で重要です。リソースは一意で、検出スキャン間で一貫して特定する必要があります。エージェントは、一意の **リソースキーを作成してリソース** を特定します。リソースキーはリソースの種類によって異なります。キーに関係なく、リソースに侵

入し、エージェントを検出できる必要があります。キーはグローバルに一意である必要はありませんが、親リソースの下で一意である必要があります。JMX サーバーでは、リソースキーはインスタンスに固有のコネクターアドレスになります。

```
    public DiscoveredResourceDetails discoverResource(Configuration pluginConfig,
                                                      ResourceDiscoveryContext discoveryContext)
            throws InvalidPluginConfigurationException {
        // TODO: Connect to the remote JVM to verify the user-specified conn props are valid, and if
connecting
        // fails, throw an exception.
        String resourceKey =
pluginConfig.getSimpleValue(CONNECTOR_ADDRESS_CONFIG_PROPERTY, null);
        String connectionType = pluginConfig.getSimpleValue(CONNECTION_TYPE, null);

        // TODO (ips, 09/04/09): We should connect to the remote JVM in order to obtain its version.
        String version = null;

        DiscoveredResourceDetails resourceDetails = new
DiscoveredResourceDetails(discoveryContext.getResourceType(),
                resourceKey, "Java VM", version, connectionType + " [" + resourceKey + "]", pluginConfig,
null);
        return resourceDetails;
    }
```

オプションの依存関係は検出に影響する可能性があります。埋め込みプラグインは、子タイプをソースプラグインからコピーし、埋め込みプラグインの検出コンポーネントを使用して検出を実行します。インジェクトされたプラグインは、親リソースを取得し、すべての子リソースタイプを通じてサイクルを行い、各タイプの検出を実行し、親コンポーネントを各子タイプの検出方法に挿入します。

多くの場合、リソースはローカルマシンで実行しているプロセスです。JBoss ON エージェントはプロセステーブルに対してクエリーを実行してこれらのローカルプロセスを検出できます。これは最初に **<process-scan>** 要素を使用してプラグイン記述子に定義され、その後 Discovery コンポーネントに実装されます。

プラグイン記述子に定義された各リソースタイプには **<process-scan>** 子要素を持つことができます。**<process-scan>** 要素自体は空ですが、必要な属性が 2 つあり **name**、. は特定 **query name** のスキャンメソッドを特定します **query**。は何かを行う属性です。**query** は、PIQL(Process Info Query Language)で記述された文字列です。この値は、プロセスの検索に使用されます。

sourceforgePIQL API ドキュメントは、 PIQL クエリーの構文に関する多くの情報を提供します。

PIQL プロセスのスキャンクエリーの基本的な形式は、プロセスを検索する 3 つの用語であり、マーカーの種類を特定してから、一致する値です。

```
process|attribute|match=value,arg|attribute|match=value
```

検出のプロセススキャンでは、スキャンは通常プロセス名または PID ファイルを探します。

名前でプロセスを検索するには、特定タイプのリソースや特定インスタンスへの検索を絞り込むために、追加の属性が必要になる場合があります。たとえば、JBoss AS インスタンスには、**java** で始まるプロセス名があり、**org.jboss.Main** の値を持つ引数があります。**ps** 情報には、これらの属性の両方が含まれます。

```
jsmith    2035   0.0 -1.5   724712  30616  p7  S+    9:49PM   0:01.61 java
 -Dprogram.name=run.sh -Xms128m -Xmx512m -Dsun.rmi.dgc.client.gcInterval=3600000
```

```
-Dsun.rmi.dgc.server.gcInterval=3600000 -Djboss.platform.mbeanserver
-Djava.endorsed.dirs=/devel/jboss-4.0.5.GA/lib/endorsed
-classpath /devel/jboss-4.0.5.GA/bin/run.jar:/lib/tools.jar
org.jboss.Main -c minimal
```

PIQL クエリーは、query 属性に定義された一致する引数を指定した **basename** クエリー属性を使用して、プロセス名の一致を試行し **arg** ます。

```
process|attribute|match=value,arg|attribute|match=value
  |     |     |____    |_  |____    |______
  |     |        |     |     |        |
  |     |        |     |     |        |
  |     |        |     |     |        |
  |     |        |     |     |        |
process|basename|match=^java.*,arg|org.jboss.Main|match=.*
```

プロセススキャンクエリーは pid ファイルと一致することができます。これは、プロセスを識別する簡単な方法です。PID ベースの PIQL クエリーには、若干単純な形式があります。

```
process|attribute|match=value
```

例：

```
process|pidfile|match=/etc/product/lock.pid
```

rhq-plugin.xml 記述子ファイルにスキャンクエリーを定義したら、スキャンおよびプロセス結果を実装するために検出コンポーネントを記述する必要があります。

例4.1 検出コンポーネントのプロセススキャンメソッド

```
List<ProcessScanResult< autoDiscoveryResults =
  context.getAutoDiscoveredProcesses();
for (ProcessScanResult result : autoDiscoveryResults) {
   ProcessInfo procInfo = result.getProcessInfo();
   ....
   // as before
   DiscoveredResourceDetails detail =
     new DiscoveredResourceDetails(
        resourceType, key, name, null,
        description, childConfig, procInfo
     );
   result.add(detail);
}
```

エージェントは、メトリクスを収集する（監視用）およびスケジュールされた操作（またはタスク）をリソース上で起動してリソースを管理します。これら 2 つの領域はプラグインで同じように設定されます。

モニタリングエリアは、プラグイン記述子の **<metric>** 要素で設定されます。

```
<metric displayName="Bytes Sent"
        description="Shows the rate that data bytes are sent by the Web service."
        property="Bytes Sent/sec"
```

```
            defaultOn="true"
            displayType="summary"
            measurementType="trendsup"
            units="bytes"/>
```

その属性の最も関連する属性は監視プロパティーに関連しています（これは部分的にリソース自体によって定義されます）。

- **property** リソースモニタリングプロパティーを特定します。
- **measurementType** 収集されるデータタイプを設定します。
- **units** モニタリングする単位を設定します。

プラグイン Java コンポーネントでは、をプルしてメトリクスを最初に設定し **MeasurementFacet** ます。

```
public class testComponent implements ResourceComponent
, MeasurementFacet
, OperationFacet
```

次に、各タイプのモニタリングデータの **MeasurementScheduleRequest** エンティティーを **MeasurementReport**持つモニタリング用のエントリーが存在します。

```
    public void getValues(MeasurementReport report, Set<MeasurementScheduleRequest> metrics)
throws Exception {

        String propertyBase = "\\Web Service(_Total)\\";
        Pdh pdh = new Pdh();

        for (MeasurementScheduleRequest request : metrics) {
            double value = pdh.getRawValue(propertyBase + request.getName());
            report.addData(new MeasurementDataNumeric(request, value));
        }
}
```

重要

メトリクスを定義する際、値の配列など、大きなデータ構造の一部を形成する値は、個別の値を監視する必要があります。

同様に、操作は **<operation>** 要素のプラグイン記述子で設定され、、を使用してプラグイン Java コンポーネントに実装され **OperationFacet**、**OperationResult** メソッドで呼び出されます。

バグを報告します。

4.2.2.3. イベント

リソースのイベントは基本的にエージェントによって認識されるログメッセージのタイプです。

プラグイン記述子では、イベントはロギング領域を名前で識別する単純な **<event>** 要素（子なし）によって設定されます。

```
<event name="errorLogEntry" description="an entry in the error log file"/>
```

イベント処理は、 **EventPoller** コンポーネントこれは大きなプラグイン Java コンポーネントにすることができますが、通常は別の ***EventPoller.java** コンポーネントに分割されます。イベントポーリングの実装方法は、リソースとロギングの性質によって異なります。最も簡単な方法の 1 つは、**EventPoller()**次に、イベントタイプを定義し、イベントのポーリング方法を設定します。

```
    public PerfTestEventPoller(ResourceContext resourceContext) {
        this.resourceContext = resourceContext;
    }

    public String getEventType() {
        return PERFTEST_EVENT_TYPE;
    }

    public Set<Event> poll() {
        int count = Integer.parseInt(System.getProperty(SYSPROP_EVENTS_COUNT, "1"));
        String severityString = System.getProperty(SYSPROP_EVENTS_SEVERITY,
EventSeverity.INFO.name());
        EventSeverity severity = EventSeverity.valueOf(severityString);
        Set<Event> events = new HashSet<Event>(count);
        for (int i = 0; i < count; i++) {
            Event event = new Event(PERFTEST_EVENT_TYPE, "source.loc",
System.currentTimeMillis(), severity, "event #"
                + i);
            events.add(event);
        }
        return events;
    }
```

バグを報告します。

4.2.2.4. リソース設定

リソースには、エージェントによって管理される JBoss ON GUI でパラメーターまたは設定を変更できます。編集できるこれらのプロパティーは、**<resource-configuration>** 要素のプラグイン記述子で定義されます。これらの設定要素は、**<plugin-configuration>** 要素と同じ規則に従います。プロパティーは **<simple-property>** 要素として定義され、UI で折りたたみ可能なグループに一覧表示したり、マップしたり、整理したりできます。

```
        <resource-configuration>
          <c:group name="Attributes">
            <c:simple-property
              name="appBase"
              required="true"
              readOnly="true"
              description="The Application Base directory for this virtual host." />
            <c:simple-property
              name="autoDeploy"
              type="boolean"
              description="Does this host deploy new applications dropped in appBase at runtime?" />
            <c:simple-property
              name="deployOnStartup"
              type="boolean"
              description="Does this host deploy applications in appBase at startup?" />
            <c:simple-property
              name="deployXML"
```

```
            displayName="Deploy XML"
            type="boolean"
            description="deploy Context XML config files?" />
          <c:simple-property
            name="unpackWARs"
            displayName="Unpack WARs"
            type="boolean"
            description="Does this Host automatically unpack deployed WAR files?" />
          <c:simple-property
            name="aliases"
            required="false"
            type="longString"
            description="Aliases assigned to the Host. When editing, each alias must be on a new line.
Aliases are automatically lowercased." />
        </c:group>
      </resource-configuration>
```

プラグイン Java コンポーネントで最初に定義されているものは、現在の設定をロードできることです。

```
public Configuration loadResourceConfiguration() {
    Configuration configuration = super.loadResourceConfiguration();
    try {
        resetConfig(CONFIG_ALIASES, configuration);
    } catch (Exception e) {
        log.error("Failed to reset role property value", e);
    }

    return configuration;
}
```

プラグインコンポーネントの 2 番目の部分では、エージェントが設定プロパティーを変更することができます。

```
public void updateResourceConfiguration(ConfigurationUpdateReport report) {
    Configuration reportConfiguration = report.getConfiguration();
    // reserve the new alias settings
    PropertySimple newAliases = reportConfiguration.getSimple(CONFIG_ALIASES);
    // get the current alias settings
    resetConfig(CONFIG_ALIASES, reportConfiguration);
    PropertySimple currentAliases = reportConfiguration.getSimple(CONFIG_ALIASES);
    // remove the aliases config from the report so they are ignored by the mbean config processing
    reportConfiguration.remove(CONFIG_ALIASES);

    // perform standard processing on remaining config
    super.updateResourceConfiguration(report);

    // add back the aliases config so the report is complete
    reportConfiguration.put(newAliases);

    // if the mbean update failed, return now
    if (ConfigurationUpdateStatus.SUCCESS != report.getStatus()) {
        return;
    }
```

```
        // try updating the alias settings
        try {
            consolidateSettings(newAliases, currentAliases, "addAlias", "removeAlias", "alias");
        } catch (Exception e) {
            newAliases.setErrorMessage(ThrowableUtil.getStackAsString(e));
            report.setErrorMessage("Failed setting resource configuration - see property error messages
for details");
            log.info("Failure setting Tomcat VHost aliases configuration value", e);
        }

        // If all went well, persist the changes to the Tomcat server.xml
        try {
            storeConfig();
        } catch (Exception e) {
            report
                .setErrorMessage("Failed to persist configuration change.  Changes will not survive Tomcat
restart unless a successful Store Configuration operation is performed.");
        }
    }
```

バグを報告します。

4.2.3. ライフサイクルリスナー

一部のプラグインは、すぐに初期化を実行する必要があります。一部のプラグインは、読み込み時および一部のクリーンアップがアンロード時に初期化を行う必要があります。プラグインのグローバル初期化とシャットダウンは、**ライフサイクルリスナー** によって実行されます。

The **org.rhq.core.pluginapi.plugin.PluginLifecycleListener** クラスはプラグインコンポーネントで必要なグローバルリソースを割り当て、これらのリソースをクリーンアップします。

各プラグインは、トップレベルの **<plugin>** 要素に ***pluginLifecycleListener*** 属性を指定して 1 つのライフサイクルリスナーをオプションで定義できます。

```
<plugin name="Apache"
  displayName="Apache HTTP Server"
  description="Management of Apache web servers"
  package="org.rhq.plugins.apache"
  pluginLifecycleListener="ApachePluginLifecycleListener"
  ...
```

バグを報告します。

4.2.4. プラグイン依存関係： プラグイン間の関係の定義

エージェントプラグインは、他のエージェントプラグインと関係を持つことができます。これらの関係は、プラグイン間での依存関係が作成されます。プラグインは他のプラグインが読み込まれる場合にのみ操作でき、プラグインがクラスを共有できるようにするか、または既存リソース定義に追加して追加の親または子リソースを追加してリソース階層を拡張することができます。

親プラグインは、それに応じて別のプラグインを持つものです。**子プラグイン** は、別のプラグインに依存するプラグインです。

エージェントプラグインには、依存関係を定義する 3 つの方法があります。

1. 必要な依存関係は **<depends>** 要素を使用して設定されます。使用するだけで必要なプラグインが読み込まれないと、他のプラグインが読み込まれないことを **<depends>** 意味します。**useClasses** 属性を追加すると、親プラグインのクラスおよび JAR ファイルを子プラグインで使用できるようになります。

2. インジェクションプラグインの依存関係は、ルートレベルのリソースが別のリソースタイプ内で実行され、親リソースが親プラグインとして定義されることを意味します。これにより、既存のリソースタイプに新しい子が追加されます。

3. 埋め込みプラグインの依存関係は、既存の子に新しい親リソースタイプを追加することを意味します。これにより、（両方のプラグインの設定に応じて）新たな親のクラスローダーを共有するか、単に検出を拡張するだけです。

重要

埋め込みおよびインジェクションプラグインの依存関係は相互排他的です。これは、同じプラグイン定義で使用することはできません。

これらすべてのプラグイン依存関係モデルには、親プラグインから依存するプラグインへの 1 つの方向のみのメタデータとクラス定義のフローがあります。情報は、別の方向ではフローできません。

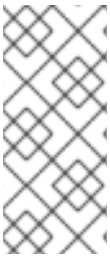
バグを報告します。

4.2.4.1. 必要なプラグイン依存関係

<plug-in> **<depends>** 要素の下にある要素は、プラグインが依存し、ロードする必要がある親プラグインを定義します。**<depends>** 要素は、必要な依存関係を指定するものです。すべてのプラグインが正常にデプロイされない限り、**<depends>** プラグインは正常にデプロイされません。

<depends> 要素は **useClasses** 属性を指定して、親プラグインから JAR でプルできます。この **useClasses** オプションは、1 つのプラグイン記述子で必要な依存関係の 1 つに対してのみ設定できます。**<depends>** 要素に **useClasses** 属性がない場合、デフォルトでプラグイン記述子に指定された最後の **<depends>** 要素は useClasses 属性を true に設定しています。

この **<depends>** 要素は、プラグインが別のプラグインのクラスにアクセスする必要がある場合や、プラグインが別のプラグインがデプロイされた場合のみデプロイする必要がある場合に使用されます。



ヒント

組み込みおよびインジェクションプラグインの依存関係は **オプション** の依存関係です。指定のプラグインがロードされていない場合は、これらの依存関係を読み込むことなくプラグインが実行されます。埋め込みプラグインまたはインジェクションプラグインを **必要** な依存関係にするには、埋め込みプラグインまたはインジェクションプラグインを **<depends>** 要素および他の設定を使用して必要なプラグインとして設定します。

バグを報告します。

4.2.4.2. 埋め込みプラグインの依存関係

埋め込みプラグインの依存関係は、既存の子リソースの新しい親リソースを追加します。依存するプラグイン（新しい親リソース用）は、子によって異なります。

埋め込みプラグインの依存関係を使用すると、サーバーまたはサービス定義は、別のプラグインにあるソースリソースタイプのコピーになります。このコピーは、リソース要素に **sourcePlugin** および

sourceType 属性を設定してプラグイン記述子に定義されます。プラグインソースを指定すると、サーバーまたはサービスはソースリソースタイプからコピーされます。これは、埋め込みサーバーまたはサービスが検出とリソースクラスを上書きできる例外で、ソースと同じメタデータを持ちます。

埋め込みプラグインは、オプションの依存関係です。

バグを報告します。

4.2.4.3. インジェクションプラグインの依存関係

エージェントプラグインのルートレベルのリソースタイプは、の内部で実行できる親リソースタイプを定義できます。これは基本的に、リソースを別の既存リソースに子タイプとして挿入します。これはインジェクションプラグイン **の** 依存関係です。

インジェクションプラグインの依存関係は、子リソースタイプがその親リソースタイプを認識しているが、子を認識しないことを示しています。プラグインの知識はスケールダウンせず、フローアップしません。親プラグインのタイプ情報は子プラグインで知られていますが、親プラグインはそれに依存する子プラグインについて何も認識しません。

インジェクション依存関係は、**<runs-inside>** 要素内で許可されたポリシーのリストです。各親はオプションの依存関係です。

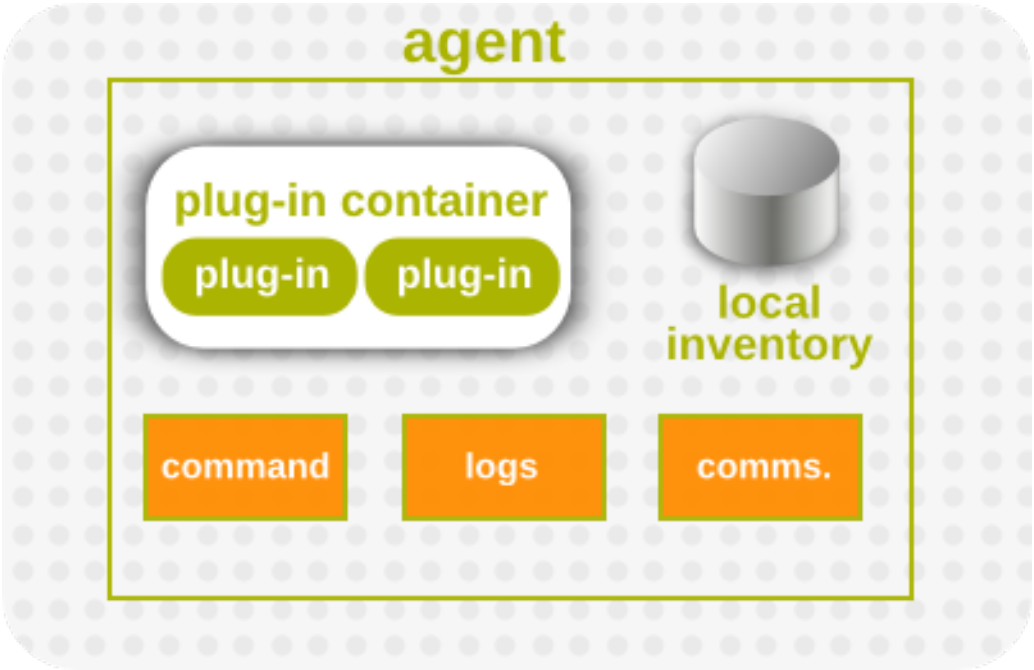
```
<runs-inside>
   <parent-resource-type name="JMX Server" plugin="JMX" />
   <parent-resource-type name="JBoss Server" plugin="JBoss AS" />
</runs-inside>
```

バグを報告します。

4.2.5. プラグイン間のクラス共有

すべてのエージェントプラグインには、プラグインコンテナーの実行中に独自のクラスローダーがあります。インベントリーの各リソースには、プラグインクラスローダーと同じクラスローダーが割り当てられます。

図4.1エージェントコンポーネント、使用方法



<depends useClasses=""> 属性が true に設定されていない限り、プラグインクラスローダーはすべて相互に分離されます。プラグインが別のプラグインに直接依存し、その依存関係が <depends useClasses="true"> で定義されている場合、その親プラグインの JAR クラス（およびすべての親 JAR）が依存プラグインのクラスローダーで利用可能になります。

この種類の依存関係を作成し、プラグイン間で JAR とクラスを共有する最も一般的な理由は、あるプラグインで定義された 1 つのリソースが別のプラグインで定義される別のリソースでデプロイされ、実行されるためです。子プラグインは、親リソースに接続して、子リソースの検出および管理を行う方法が必要になります。デフォルトでは、すべての管理リソースには共有されるリソースクラスローダーが割り当てられます。このクラスローダーは、プラグインまたは親リソースに属することができます。

リソースコンポーネントが実際に管理されたリソースへの接続を作成する必要がある場合は、そのリソースのクラスローダーは、通常の共有プラグインクラスローダーとは異なる必要があります。通常、リソースには、独自のクラスローダー内で管理されたリソースに接続するために必要なクライアント JAR が必要です。これらのクライアント JAR は通常、管理されているリソースのバージョンに非常に固有のものです。接続リソースは接続方法を認識しているので、接続の作成を担当する必要があります。リソースに独自の接続クラスローダーが必要な場合は、リソースタイプの属性を指定 **classLoader="instance"** し、リソースタイプの検出コンポーネントが実装されていることを確認します。 **ClassLoaderFacet** したがって、管理されている特定バージョンの特定のバージョンの接続クラスを追加で検索できる場合は、プラグインコンテナーに指示します。

に 例4.2「プラグイン Z のクラスローダー」、Z1.server の **classLoader** オプションが shared に設定されます。つまり、Z1.server リソースは親リソースとクラスローダーを共有し、そのクラスローダーはリソースクラスローダーまたはプラグインクラスローダーになります。すべての Z1.server リソースは同じクラスローダーを使用します。

例4.2 プラグイン Z のクラスローダー

```
<plug-in name="Z">

<depends plugin="A" />

<server name="Z1.server" classLoader="shared">
  <runs-inside>
    <parent-resource-type name="B1.server" plugin="B"/>
    <parent-resource-type name="C1.server" plugin="C"/>
  </runs-inside>
</server>

<server name="Z2.server" sourcePlugin="D" sourceType="D1" classLoader="instance">
</server>

<server name="Z3.server" classLoader="instance">
</server>

</plugin>
```

通常、**classLoader** オプションを instance に設定すると、各リソースが独自のリソースプラグインを使用することを意味します。ただし、Z2.server の場合は、Z2.server プラグインはプラグイン D の値を組み込むことで拡張されるため、Z2.server リソースはクラスローダーを親プラグインと共有します。

z3.server は、**classLoader** オプションが instance に設定され、インジェクトまたは埋め込み依存関係のないため、単に独自のリソースクラスローダーを使用します。**classLoader** オプションが instance に設定されている場合、 **ResourceDiscoveryComponent** 実装は任意で、 **ClassLoaderFacet** リソー

スのクラスローダーに配置する必要のある追加の JAR を **List<URL>** 参照するメソッド (**getAdditionalClasspathUrls**)を使用します。プラグインコンテナーがリソースのクラスローダーを作成する必要がある場合は、リソースの検出コンポーネントがこのファセットを実装しているかどうかを確認します。そうであれば、追加のクラスパス URL を取得し、リソースの作成時にこれをリソースクラスローダーに追加します。

リソースタイプがインジェクションまたは埋め込み依存関係のいずれかで定義されている場合、クラスローダーはその **classLoader** 属性値とその親の属性値の両方に依存し **classLoader** ます。

リソースクラスローダー	親クラスローダー	classloader Description
shared	shared	リソースがクラスと親クラスの両方にアクセスできるように、**useClasses** 値を true に設定する必要があります。
インスタンス	shared	リソースには主に独自のクラスが必要ですが、親クラスを使用できるように **useclasses** するため、リソースを true に設定することは有益です。
shared	インスタンス	リソースは独自のクラスローダーのみを使用します。
インスタンス	インスタンス	リソースは独自のクラスローダーのみを使用します。

バグを報告します。

4.3. 拡張例： リソースのコンテンツタイプ

JBoss ON でリソースを管理する方法は、エージェントのリソースプラグインでどのように記述されるかによって異なります。これには、収集できるすべてのモニタリングメトリクス、設定できるすべての設定、および実行できるすべての操作が含まれます。これには、そのリソースにデプロイできるコンテンツと、どのようなコンテンツが想定されるかも含まれます。

パッケージ は、コンテンツの一部を参照します。パッケージは通常、JBoss AS JAR ファイルのような種類のファイルです。ただし、リソースのパッケージタイプはリソースプラグインで定義されているので、リソースプラグインがこれを使用するように設定されている限り、パッケージは任意のものになります。

リソースに関連する他の要素と同様に、パッケージタイプはリソースタイプのプラグイン記述子で定義されます。各パッケージタイプは、特定の属性（でリスト パッケージの属性）を定義できますが、すべてのパッケージは名前とそのタイプを定義する必要があります（カテゴリーは 4 つあります）。

プラグイン記述子では、パッケージタイプは **<content>** 要素によって識別されます。必要なプロパティーはメイン **<content>** 要素でフラグとして設定されます。設定可能なプロパティーは、リソースに新しいパッケージをアップロードする際にユーザーが設定する設定可能なプロパティーは **<c:simple-property>** 子要素に指定されます。たとえば、Platform Resource Plug-in のこの content 要素は、Windows プラットフォームの deployable（カテゴリー）パッケージタイプを特定します。

```
<content name="InstalledSoftware" displayName="Installed Software" category="deployable"
```

```
description="Installed Windows Software">
        <configuration>
          <c:simple-property name="Publisher"/>
          <c:simple-property name="Comments"/>
          <c:simple-property name="Contact"/>
          <c:simple-property name="HelpLink"/>
          <c:simple-property name="HelpTelephone"/>
          <c:simple-property name="InstallLocation"/>
          <c:simple-property name="InstallSource"/>
          <c:simple-property name="EstimatedSize" units="kilobytes"/>
        </configuration>
      </content>
```

パッケージはリソースに手動で追加できますが、エージェントは新規コンテンツをアクティブに **チェック** し、検出されたコンテンツをそのインベントリーに追加できます。パッケージは、再帰的な **パッケージ検出スキャン** を介して JBoss ON では同梱されています。この検出が発生する間隔は、プラグインの記述子にあるパッケージの定義に明示的に設定することも、プラグインスキーマファイルで指定したデフォルト値を使用できます。

パッケージの属性

表示名（オプション）

ユーザーインターフェースのパッケージタイプの名前。

説明（オプション）

は、このタイプのパッケージに含まれるコンテンツの種類を説明します。

カテゴリー（必須）

4 つの列挙オプションの 1 つ。

- 実行可能なスクリプト（編集可能な可能性があります）
- 実行可能なバイナリー
- 設定ファイル（リソースの設定ファイル）
- deployable

検出間隔（オプション）

このタイプのパッケージ検出スキャンの間隔を定義します。異なるパッケージタイプを間隔で設定して、パッケージインベントリーの変更の可能性を表すことができます。

作成タイプのフラグ（オプション）

true に設定すると、エンクロージングリソースタイプのリソースを作成する際にこのタイプのパッケージが使用されます。この状況の例は、Java EAR ファイルです。JBoss ON のエンタープライズアプリケーションを表す EAR リソースタイプがあります。そのリソースタイプの下には、EAR ファイル自体を表すために定義されたパッケージタイプがあります。このパッケージタイプには、作成タイプとしてフラグが付けられます。新規の EAR リソースを作成する場合は、EAR ファイルを同時に作成する必要があります。パッケージは通常新しいリソースの作成を表していないため、この属性のデフォルトは false です。

設定（オプション）

configuration 要素を使用すると、プラグインはパッケージタイプに関するオープンな属性のセット

を定義できます。これらの値はパッケージの検出中に設定され、読み取り専用としてマークされていない場合は、アーティファクトの作成時にユーザーに指定できます。この設定要素のプロパティーの例として、EAR ファイルが展開形式または zipped としてデプロイされるかどうかを記述するブール値があります。EAR ファイルが検出されると、このフラグにデータが設定され、パッケージタイプに指定された情報が伝送されます。さらに、JBoss ON で新しい EAR ファイルをデプロイする場合、このフラグを設定して、パッケージを AS インスタンスにデプロイする方法を指定できます。

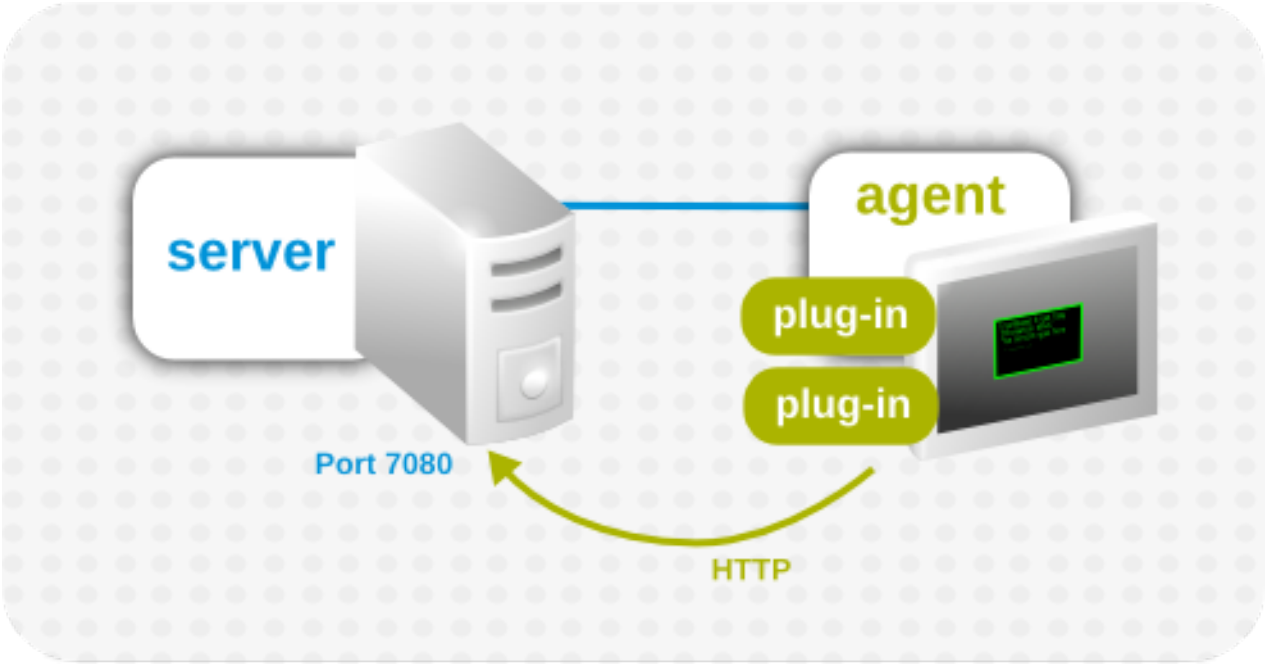
バグを報告します。

4.4. 拡張例： HTTP メトリクス

このサンプルプラグインは、エージェントが HTTP サーバーに接続して、Web サーバー自体がパフォーマンスまたは可用性を監視するために書き込まれます。このプラグインは 2 つのタスクを実行します。

- GET または HEAD リクエストを指定のサーバーのベース URL に発行します。
- HTTP の戻りコードと応答時間をリソース特性として収集します。

図4.2 基本的なエージェントプラグインシナリオ



注記

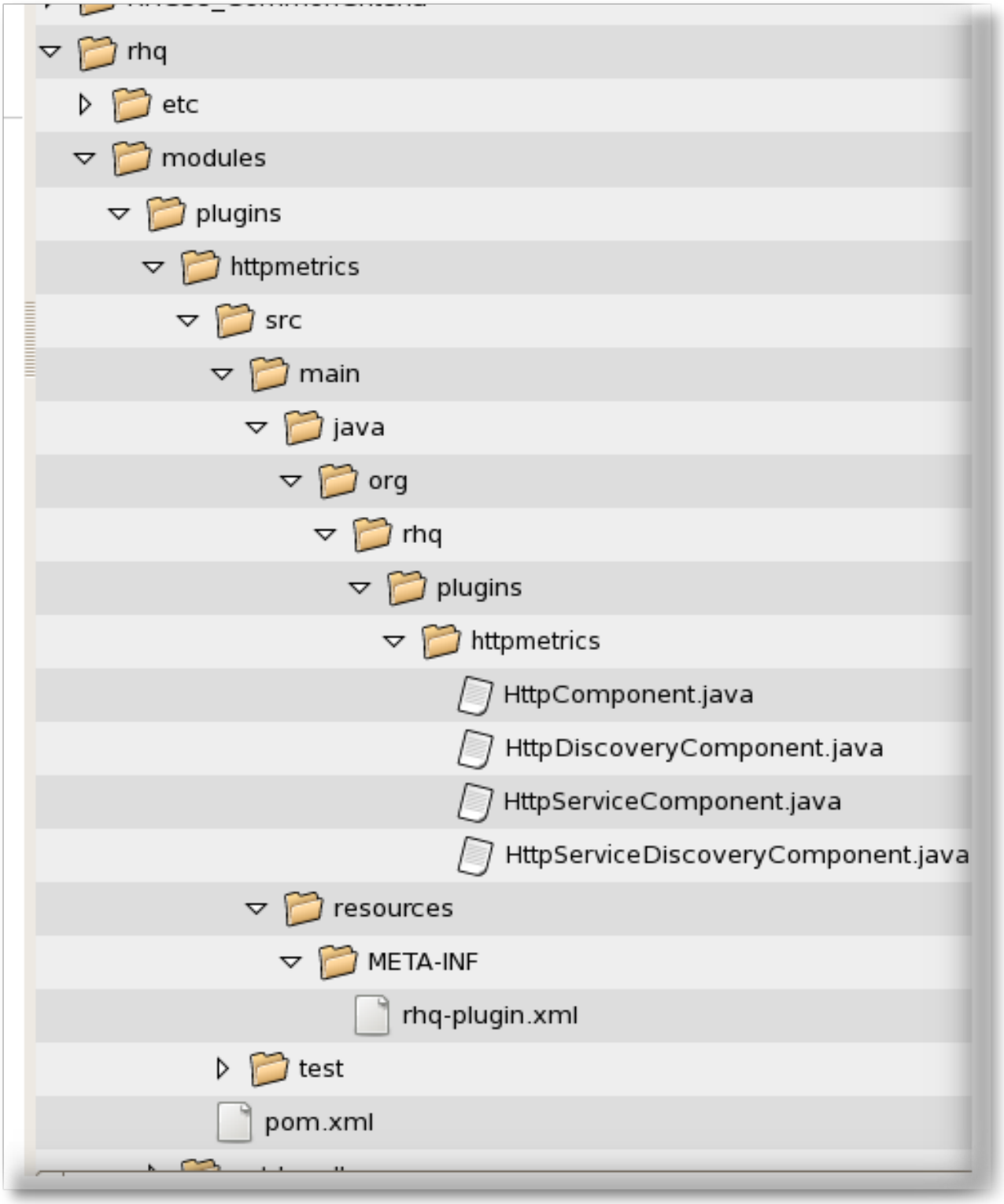
分かりやすくするため、このプラグインは JBoss ON サーバーと同じマシンで実行されているエージェントに対して書き込まれます。

HTTP メトリクスプラグインは、サーバーと子サービスを実行して 2 つのモニタリングメトリクスを収集するように設計されています。これには、プラグインの動作を定義する Java ファイルと、プラグインが URL を見つけるために必要な検出コンポーネントの動作が 2 つ必要です。

すべてのエージェントプラグインと同様に、この例では **rhq-plugin.xml** ファイルをプラグイン記述子として持ちます。これは、JBoss ON がプラグイン設定を認識するために必要なものです。プラグインは Maven プロジェクトとして構築されるため、適切に設定された JAR **pom.xml** ファイルをエージェ

ントプラグインとしてデプロイできるため、必須ではありませんが、ファイルには必須ではありません。

図4.3 エージェントプラグインプロジェクトのディレクトリーレイアウト



バグを報告します。

4.4.1. プラグイン記述子(rhq-plugin.xml)の確認

プラグイン記述子は、すべてが連携する場所です。プラグイン記述子の最初の部分は、名前、バージョン番号、およびスキーマなどのプラグインの基本情報を定義します。

例4.3 基本的なプラグイン情報

```
<?xml version="1.0" encoding="UTF-8" ?>
<plugin name="HttpTest"
        displayName="HttpTest plugin"
        package="org.rhq.plugins.httptest"
        version="2.0"
        description="Monitoring of http servers"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="urn:xmlns:rhq-plugin"
        xmlns:c="urn:xmlns:rhq-configuration">
```

この **package** 属性は、記述子のプラグイン設定で参照される Java クラス名の Java パッケージを特定します。

HTTP メトリクスプラグインは、子リソースを実行するサーバーとして定義されます。サービスは独自に実行できないため、最初にサーバーとして HTTP メトリクスリソースを定義すると、複数のサービスがそこから起動できるようになります。

例4.4 サーバー定義

```
<server name="HttpCheck"
  description="Httpserver pinging"
  discovery="HttpDiscoveryComponent"
  class="HttpComponent">
```

次の手順では、サービス要素をサーバーの子として追加します。サーバーとサービスの動作を区別するために、サービスには独自の検出コンポーネントとプラグインコンポーネントがあります。これは、独自の **.java** ファイルとクラスを指します。この **supportsManualAdd** オプションは、UI を使用して HTTP サービスを手動で追加できることを JBoss ON に指定します。これは管理に重要になります。

例4.5 サービス定義

```
<service name="HttpServiceCheck"
        discovery="HttpServiceDiscoveryComponent"
        class="HttpServiceComponent"
        description="One remote Http Server"
        supportsManualAdd="true"
```

<service> 要素の途中では、UI で設定されるプラグインプロパティーを定義します。これは、simple（URL の単純な文字列の設定）にすることができます。

例4.6 簡易設定プロパティー

```
<plugin-configuration>
  <c:simple-property name="url"
    type="string"
    required="true" />
</plugin-configuration>
```

プロパティーを使用すると、より高度な設定が可能になり、どの程度の情報を設定するかに応じて、プロトコル、ポート、ホスト名、または IP アドレスに特定の値が許可されます。

例4.7 複雑な設定プロパティー

```
<plugin-configuration>
  <c:list-property name="Servers">
    <c:map-property name="OneServer">
      <c:simple-property name="host"/>
      <c:simple-property name="port">
        <c:integer-constraint
          minimum="0"
          maximum="65535"/>
      </c:simple-property>
      <c:simple-property name="protocol">
        <c:property-options>
          <c:option value="http" default="true"/>
          <c:option value="https"/>
        </c:property-options>
      </c:simple-property>
    </c:map-property>
  </c:list-property>
</plugin-configuration>
```

<service> 要素の最後の部分には、HTTP メトリクスプラグインに設定されたメトリクスが含まれます。応答時間の最初のメトリクスは、数値のデータタイプを収集します。ステータスメトリックは、特性のデータタイプを収集します。（JBoss ON は、領域を節約するための変更特性のみを保管するのに十分なインテリジェントです。）

例4.8 定義されたメトリクス

```
    <metric property="responseTime"
            displayName="Response Time"
            measurementType="dynamic"
            units="milliseconds"
            displayType="summary"/>

    <metric property="status"
            displayName="Status Code"
            data type="trait"
            displayType="summary"/>
     </service>
  </server>
</plugin>
```

メトリックの定義に使用できる属性は、エージェントプラグインの XML スキーマで定義されます。この例で使用される属性はに記載されてい 表4.5「メトリクスの属性」 ます。

メトリクスを定義する際、値の配列など、大きなデータ構造の一部を形成する値は、個別の値を監視する必要があります。

表4.5 メトリクスの属性

attribute	description
property	このメトリクスの一意の名前を指定します。を使ってコードで名前を取得することもできます。**getName()** コール。
description	メトリックの人間が判読できる説明を提供します。
displayName	JBoss ON UI に表示される名前を指定します。
データタイプ	数値や特性などのメトリックのタイプを設定します。
units	数値データタイプに使用する測定単位。
displayType	値がに設定されていると **summary**、メトリクスはインジケータチャートに表示され、デフォルトで収集されます。
defaultOn	デフォルトで収集されるメトリクスを設定します。
measurementType	数値の値の特性を設定します。オプションは、傾向、傾向、または動的です。両方の傾向のメトリックの場合、システムは毎分のメトリックを自動で作成します。 値の配列など、大きなデータ構造の一部を形成する値は、監視の前に個別の値にデコンストラクトする必要があります。

バグを報告します。

4.4.2. Discovery コンポーネント（HttpDiscoveryComponent.java および HttpServiceDiscoveryComponent.java）の確認

2 つの Java ファイルは、HTTP メトリクスサーバーと監視に定義された URL を検出する方法を定義します。

最初の Java ファイルは **HttpDiscoveryComponent.java**、HTTP メトリクスサーバーを検出します。検出コンポーネントは、リソースを検出するためにエージェントの InventoryManager によって呼び出されます。これは、プロセステーブルのスキャン、MBeanServer のクエリー、またはその他の方法で実行できます。いずれの方法でも最も重要なことは、検出コンポーネントが同じリソースに対して同じ一意の鍵を返すことです。DiscoveryComponent を実装する必要があり **org.rhq.core.pluginapi.inventory.ResourceDiscoveryComponent**、実装する必要があり **discoverResources()**ます。

基本的には、プロセスのスキャンで検出されたリソースの一覧を取得し、検出されたリソースの詳細を作成します。ProcessInfo はプロセスの詳細情報を取得し、検出した特定のタイプのリソースを最終リストから除外するために使用できます。

例4.9 HttpDiscoveryComponent.java

```
public class HttpDiscoveryComponent implements
    ResourceDiscoveryComponent
{
    public Set discoverResources(ResourceDiscoveryContext context)
        throws InvalidPluginConfigurationException, Exception
    {
        Set<DiscoveredResourceDetails> result =
            new HashSet<DiscoveredResourceDetails>();

        String key = "http://localhost:7080/"; // Jon server
        String name = key;
        String description = "Http server at " + key;
        Configuration configuration = null;
        ResourceType resourceType = context.getResourceType();
        DiscoveredResourceDetails detail =
            new DiscoveredResourceDetails(resourceType,
                key, name, null, description,
                configuration, null
            );

        result.add(detail);

        return result;
    }
```

サービス検出コンポーネント（で定義 **HttpServiceDiscoveryComponent.java**）は、検出スキャンではなく、GUI で渡される情報に依存し、そのリソースを設定します。Java ファイルの最初の定義はサーバー検出の場合と似ていますが、この定義には追加の定義があります。 **list<Configuration> childConfigs** UI 経由で渡される情報を処理する。これにより、ユーザーが提供する必要な **url** 情報に関する情報を取得します。

例4.10 サービスディスカバリー

```
public class HttpServiceDiscoveryComponent
    implements ResourceDiscoveryComponent<HttpServiceComponent>;
{
    public Set<DiscoveredResourceDetails> discoverResources
        (ResourceDiscoveryContext<HttpServiceComponent> context)
        throws InvalidPluginConfigurationException, Exception
    {
        Set<DiscoveredResourceDetails> result =
            new HashSet<DiscoveredResourceDetails>();
        ResourceType resourceType = context.getResourceType();

        List<Configuration> childConfigs =
            context.getPluginConfigurations();
        for (Configuration childConfig : childConfigs) {
            String key = childConfig.getSimpleValue("url", null);
            if (key == null)
                throw new InvalidPluginConfigurationException(
                    "No URL provided");
```

設定された URL の一覧は、検出プロセスで使用される一意のリソースキーのリソース名、説明、タイプ、および（重要）リソースとして処理され、リソースとして追加されます。

例4.11 HTTP URL リソースの一覧表示

```
            String name = key;
            String description = "Http server at " + key;
            DiscoveredResourceDetails detail =
                new DiscoveredResourceDetails(
                    resourceType, key, name, null,
                    description, childConfig, null
                );
            result.add(detail);
        }
        return result;
    }
```

バグを報告します。

4.4.3. プラグインコンポーネント（HttpComponent.java および HttpServiceComponent.java）の確認

プラグインコンポーネントは、検出の完了後に機能するプラグインの一部です。

サーバーコンポーネント(**HttpComponent.java**)では、プラグインは非常にシンプルです。コンポーネントは ResourceComponent インターフェースからプレースホルダーメソッドのみを実装し、サーバーの可用性を設定します。可用性を UP に設定すると、リソースコンポーネントが自動的に起動できるようになります。

例4.12 検出後のサーバーの可用性

```
    public AvailabilityType getAvailability() {
        return AvailabilityType.UP;
    }
```

サービスコンポーネント(**HttpServiceComponent.java**)は、プラグイン記述子に定義された操作を実行する必要があるため、より複雑です。

プラグイン記述子の各基本的な機能は、適切なエージェントファセットを使用して実装されます。すべての gent ファセットがに一覧表示され「プラグインの Facets」 ます。HTTP メトリクスコンポーネントは、特に記述子の **<metric>** 要素をにマップします。 **MeasurementFacet**.

各ファセットには独自の実装方法があります。処理メトリクスを必要とする監視およびその他の操作の場合、 **MeasurementFacet** 以下のメソッドを実装します。

```
getValues(MeasurementReport report, Set metrics)
```

The **MeasurementReport** 渡されるのは、監視結果が追加される場所です。The **metrics** value は、データを収集する必要のあるメトリクスの一覧です。この情報はすべて、**<metrics>** 要素または UI 設定で定義できます。

次の部分は、作業を行うプラグインコンポーネントです。

```
public class HttpComponent implements ResourceComponent,
    MeasurementFacet
{
    URL url;       // remote server url
    long time;     // response time from last collection
    String status; // Status code from last collection
```

内容を監視するには、 **getValues()** メソッド **MeasurementFacet** 実装は必須ですが、これを実行する最初のステップではありません。リソースがダウンしている場合はリソースを検出できないので、最初のステップでは、開始元となる起動値を設定します。 **ResourceContext** UP の可用性を付与します。

例4.13 サービスリソースの可用性

```
    public void start(ResourceContext context)
        throws InvalidPluginConfigurationException, Exception
    {
        url = new URL(context.getResourceKey());
        // Provide an initial status, so
        // getAvailability() returns up
        status = "200";
    }
```

サービスが起動したら、 **getValues()** 実装が可能です。これは、指定の URL から実際に監視データを収集します。

例4.14 getValues（）の実装

```
  public void getValues(MeasurementReport report,
          Set<MeasurementScheduleRequest> metrics)
      throws Exception
  {
      getData();
      // Loop over the incoming requests and
      // fill in the requested data
      for (MeasurementScheduleRequest request : metrics)
      {
          if (request.getName().equals("responseTime")) {
              report.addData(new MeasurementDataNumeric(
                  request, new Double(time)));
          } else if (request.getName().equals("status")) {
              report.addData(new MeasurementDataTrait
                  (request, status));
          }
      }
  }
```

最後のステップは、情報を処理することです。実装 **getData()** のメソッド **MeasurementFacet** 受信リクエストをループして、希望するメトリックを確認し、収集した値を指定します。データのタイプによっては、データは正しいデータにラップされる可能性があります。 **MeasurementData*** クラス。

例4.15 getData（）の実装

```
private void getData()
   {
      HttpURLConnection con = null;
      int code = 0;
      try {
         con = (HttpURLConnection) url.openConnection();
         con.setConnectTimeout(1000);
         long now = System.currentTimeMillis();
         con.connect();
         code = con.getResponseCode();
         long t2 = System.currentTimeMillis();
         time = t2 - now;
      } catch (Exception e) {
         e.printStackTrace();
      }
      if (con != null)
         con.disconnect();

      status = String.valueOf(code);
   }
```

HTTP Metrics プラグインの実装は非常に簡単です。URL コネクションを開き、接続に時間がかかり、ステータスコードを取得します。以上です。

バグを報告します。

4.5. 例： 組み込みおよび挿入されたプラグイン依存関係

JBoss ON エージェントプラグインには、プラグイン間の依存関係を定義する方法は複数あります（簡単な依存、埋め込み、およびインジェクト）。依存関係の定義方法は、プラグインの動作に影響します。詳細は、「」を参照してください 「プラグイン依存関係： プラグイン間の関係の定義」。

以下の例は、各依存関係タイプがエージェントプラグインのプラグイン記述子で定義される方法を示しています。

バグを報告します。

4.5.1. 簡単な依存関係： JBoss AS および JMX プラグイン

必要な依存関係は **<depends>** タグを使用して定義されます。つまり、これをデプロイする必要があるプラグインの前に、必要なプラグインを正常にデプロイする必要があります。この例は JBoss AS で、内部で実行している JMX サーバーがあります。JBoss ON の JBoss AS プラグインは、JMX プラグインの依存関係を設定します。

JMX プラグイン記述子は、JMX サーバーの設定を定義します。

例4.16 JMX プラグイン記述子

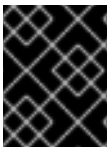
```
<plugin name="JMX">
  <server name="JMX Server" discovery="JMXDiscoveryComponent"
class="JMXServerComponent">
```

```
    ...
  </server>
</plugin>
```

JBoss AS プラグイン記述子は JMX プラグインを依存関係としてリストします。これにより、（**useClasses** 引数がに設定されたため **true**）JBoss AS プラグインクラスローダーがすべての JMX プラグインクラスを使用できるようにしますが、JBoss AS プラグイン記述子は JMX プラグインに関連するソースタイプを実際に定義または使用するわけではありません。

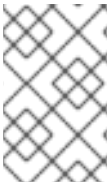
例4.17 JBoss AS プラグイン記述子

```
<plugin name="JBossAS">
  <depends plugin="JMX" useClasses="true"/>
  <server name="JBossAS Server" discovery="JBossASDiscoveryComponent"
class="JBossASServerComponent">
    ...
  </server>
</plugin>
```



重要

プラグインは、複数のプラグインを要求したり、依存したりできます。ただし、の値を持つことができる依存関係の 1 つのみです **useClasses=true**。



ヒント

JMX プラグインは汎用プラグインで、他の多くのプラグインで使用できます。JMX プラグインが管理できるすべてのアプリケーションは、JMX プラグインを使用してすべての依存関係を取得でき、JMX プラグインを拡張する EMS ライブラリーも取得できます。

バグを報告します。

4.5.2. Embedded Dependency: JVM MBeanServer および JBoss AS

Java 仮想マシンには、**プラットフォーム MBeanServer と呼ばれる JMX MBeanServer が組み込まれています。**このプラットフォームの MBeanServer を使用するすべての JVM は、MBeanServer の MBean を介してメモリー、ガベージコレクター、スレッド、およびその他のサブシステムに対して監視できます。

JBoss ON の JMX プラグインは各プラットフォーム MBean のリソースタイプを定義できます。そのため、JBoss ON エージェントはこれらの MBean を JBoss ON サービスリソースとして監視できます。

プラットフォーム MBeanServer は、スタンドアロンの JVM プロセスを使用するか、JBoss AS 仮想マシンプロセス内に組み込まれています。監視対象の JVM が JBoss サーバーに組み込まれている場合、JBoss ON プラグインは組み込みプラグインの依存関係で設定されます。**埋め込み依存関係** は、あるプラグインが内部で別のプラグインを実行していることを認識していることを意味します。

JMX プラグイン記述子は単に JMX サーバーを定義します。

例4.18 JMX プラグイン記述子

```
<plugin name="JMX">
  <server name="JMX Server" discovery="JMXDiscoveryComponent"
class="JMXServerComponent">
    <service name="VM Memory System"
          discovery="MBeanResourceDiscoveryComponent"
          class="MBeanResourceComponent"
          description="The memory system of the Java virtual machine">
    ...
    </service>
  ...
  </server>
</plugin>
```

埋め込みは JBoss AS プラグイン記述子で行われます。JMX プラグインに必要な依存関係を設定する他に、JBoss AS プラグインの **<server>** 定義は **sourcePlugin** および **sourceType** 属性でプルされます。これは、2 番目の JMX 検出スキャンを実行するためです。これは、を使用します。**org.rhq.plugins.jmx.EmbeddedJMXServerDiscoveryComponent** 特別な検出スキャンを実行するクラス。JBoss AS インスタンスに組み込まれた JVM を検索します。次に **sourcePlugin** および **sourceType** 属性はリソースタイプをコピーし、これを一意の名前にして、埋め込み JVM がスタンドアロン JVM とは異なるリソースタイプとして処理されるようにします。

例4.19 JBoss AS プラグイン記述子

```
<plugin name="JBossAS">
  <depends plugin="JMX" useClasses="true"/>
  <server name="JBossAS Server" discovery="JBossASDiscoveryComponent"
class="JBossASServerComponent">
    <server name="JBoss AS JVM"
          description="JVM of the JBossAS"
          sourcePlugin="JMX"
          sourceType="JMX Server"
          discovery="org.rhq.plugins.jmx.EmbeddedJMXServerDiscoveryComponent"
          class="org.rhq.plugins.jmx.JMXServerComponent">
    ...
    </server>
  ...
  </server>
</plugin>
```

このタイプの埋め込みプラグインは、ソースプラグインタイプとは異なる検出コンポーネントを使用して、埋め込みリソースタイプを検出できることを示しています。

バグを報告します。

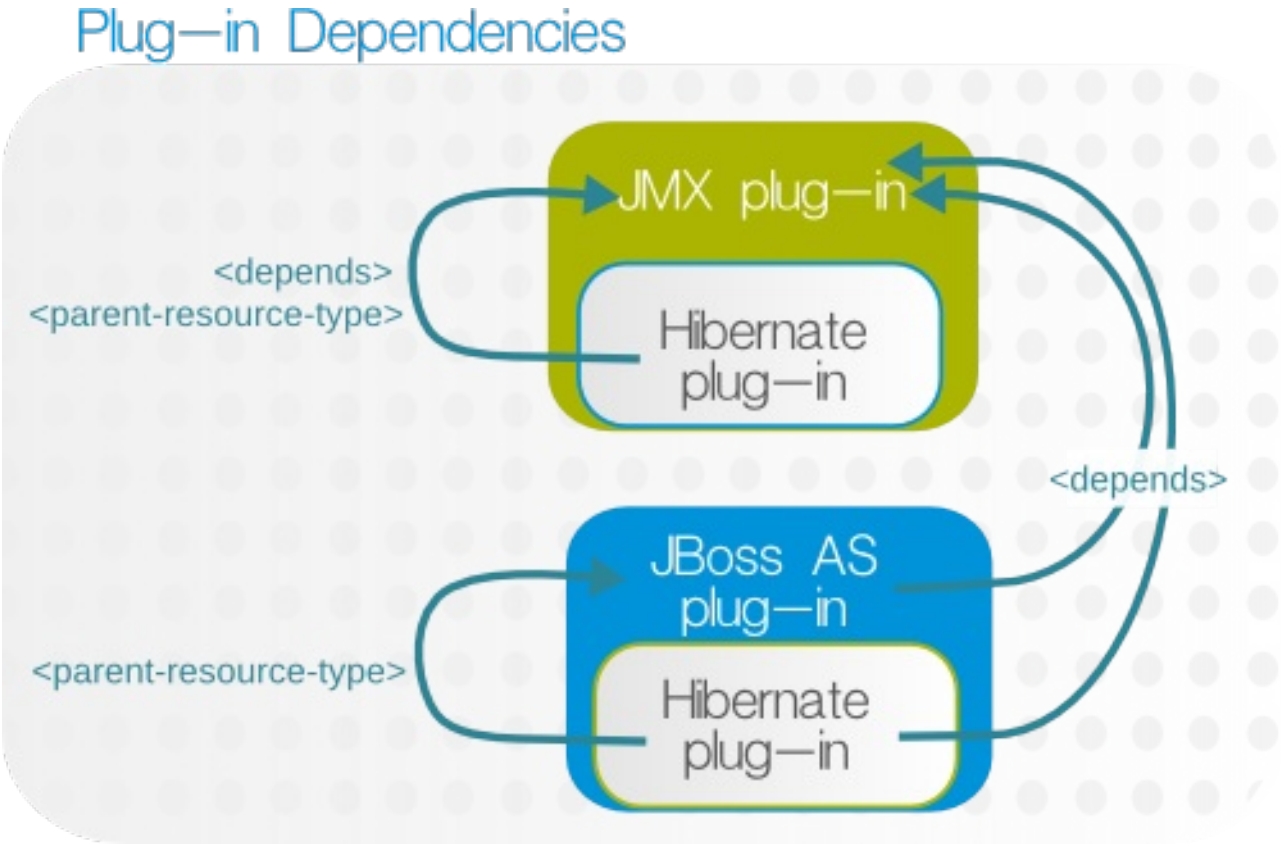
4.5.3. injected Dependency: Hibernate with JVM and JBoss AS

インジェクション依存関係 は、埋め込み依存関係の論理逆で、プラグインによって設定されたリソースが別のリソース内で実行されているプラグインにおける認識です。親リソースは依存関係として一覧表示されます。

この一般的な例は Hibernate です。Hibernate は、スタンドアロン J2SE JVM インスタンスまたは JBoss AS サーバーのいずれかで実行できます。この例では、JBoss AS インスタンスの JVM 内で実行

されます。プラグインには依存関係がチェーンされており、JMX および JBoss AS プラグインのいずれかが Hibernate プラグインに提供でき、Hibernate および JBoss AS プラグインはどちらも必要な依存関係として JMX プラグインをリストします。

図4.4 Hibernate、JMX、および JBoss AS 依存関係



前述のように、JMX プラグイン記述子は依存関係のない JMX プラグインのみを定義します。

例4.20 JMX プラグイン記述子

```
<plugin name="JMX">
  <server name="JMX Server" discovery="JMXDiscoveryComponent"
class="JMXServerComponent">
    ...
  </server>
</plugin>
```

JBoss AS プラグインは JMX プラグインに必要な依存関係を設定しますが、他の依存関係を定義する必要はありません（ただし、可能）。

例4.21 JBoss AS プラグイン記述子

```
<plugin name="JBoss AS">
  <depends plugin="JMX" useClasses="true"/>
  <server name="JBossAS Server" discovery="JBossASDiscoveryComponent"
class="JBossASServerComponent">
    ...
  </server>
</plugin>
```

最も複雑な定義は Hibernate プラグインのものです。これにより、**<depends>** 要素を使用して JMX プラグインに明示的な依存関係が設定されます。次に Hibernate プラグインは、潜在的な親タイプに対して検出スキャン（具体的には Hibernate Statistics リソース用）を実行して、リソースタイプがどのリソースタイプとして **動作** するかを定義します。親リソースタイプの一覧は **<runs-inside>** 要素に含まれ、潜在的な親は **<parent-resource-type>** 要素の名前およびプラグインタイプで識別されます。

例4.22 Hibernate プラグイン記述子

```
  <depends plugin="JMX" useClasses="true"/>
  <service name="Hibernate Statistics"
           discovery="org.rhq.plugins.jmx.MBeanResourceDiscoveryComponent"
class="StatisticsComponent">
    <runs-inside>
      <parent-resource-type name="JMX Server" plugin="JMX"/>
      <parent-resource-type name="JBossAS Server" plugin="JBossAS"/>
    </runs-inside>
  ...
  </service>
</plugin>
```

プラグインが別のプラグインに依存する場合、他のプラグインが必要な場合は暗黙的に依存します。たとえば、Hibernate は JBoss AS プラグインによって異なります。Hibernate プラグインは JMX プラグインの依存関係を明示的に示していませんでしたが、JBoss AS プラグインが必要であるため、JMX プラグインに依存します。

バグを報告します。

4.6. 拡張例： 誤差の監視

ドリフトモニタリングは、プラグインでデフォルトのドリフト定義を定義することで、リソースに対して許可されます。プラグイン設定では、誤差定義は、リソースがドリフト監視をサポートすることを示すデフォルトテンプレートを作成します。（追加テンプレートはユーザーが作成することも、デフォルトのテンプレートをリソースに適用したときに修正することができます。）

最も基本的なドリフト定義は、ドリフトシステムが監視するターゲットの場所を設定します。この場所は、リソースの複数の異なる設定エリアから特定できます。

- Filesystem - リソースにローカルとなるマシンの任意のディレクトリー
- pluginConfiguration: ホームディレクトリーなどのリソースプラグインでプロパティーを定義します。
- resourceConfiguration、リソース設定プロパティー
- measurementTrayt（リソースについての収集される特性）

このターゲットの場所は、**ベースディレクトリー** です。値が **コンテキスト** であるベースディレクトリーの **値を見つける場所を** 特定する要素。たとえば、のベースディレクトリーでは **/etc/**、drift 定義の要素は以下のようになります。

```
Value name: fileSystem
Value context: /etc
```

最も基本的なドリフト定義は、値の名前とコンテキストのみを定義する必要があります。

例4.23 ベースディレクトリーのみ

```
    <drift-definition name="Template-File System"
                     description="Monitor the file system for drift. Definitions should set a more specific
base directory as the file system root is not recommended.">
        <basedir>
            <value-context>fileSystem</value-context>
            <value-name>/</value-name>
        </basedir>
    </drift-definition>
```

サブディレクトリーや、明示的に含めるか除外する必要のあるファイルタイプなど、詳細情報を設定することができます。これらは、ベースディレクトリーの下にある追加の **パス** であり、ファイルタイプは **パターン** で識別できます。

ディレクトリーまたはファイルタイプを明示的に含めると、他のファイルおよびディレクトリーはすべて暗黙的に除外されます（また、何かが明示的に除外されている場合は、すべて暗示的に含められます）。複数のパスとパターンを定義することができます。

例4.24 含まれるパスおよびパターン

```
<drift-definition name="Template-Base Files"
                 description="Monitor base application server files for drift. It defines monitoring for
some standard sub-directories of the HOME directory.  Note, it is not recommeded to monitor all
files for an application server. There are many files, and many temp files.">
    <basedir>
        <value-context>pluginConfiguration</value-context>
        <value-name>homeDir</value-name>
    </basedir>
    <includes>
 <include path="bin" pattern="*/*.sh" />
        <include path="lib" />
        <include path="client" />
    </includes>
</drift-definition>
```

注記

リソースタイプには複数のドリフト定義を定義できますが、各ドリフト定義は単一のベースディレクトリーのみを定義してずれを監視できます。

バグを報告します。

4.7. 拡張例： プロビジョニングおよびコンテンツデプロイメント（バンドル）

バンドルシステムを使用してコンテンツをリソースにデプロイできるようにするには、コンテンツのプロビジョニング先として許可されるターゲットの場所を定義することで有効になります。

ドリフト設定と同様に、バンドル設定は 4 つの領域のいずれかの情報に基づいてターゲットの場所を特定します。

- Filesystem - リソースにローカルとなるマシンの任意のディレクトリー
- pluginConfiguration: ホームディレクトリーなどのリソースプラグインでプロパティーを定義します。
- resourceConfiguration、リソース設定プロパティー
- measurementTrayt（リソースについての収集される特性）

このエリアはバンドル定義の **値名** です。実際の値は context の **値です。**

例4.25 単一バンドルベースディレクトリー

```
    <bundle-target>
      <destination-base-dir name="Root File System" description="The top root directory on the
platform (/)" >
        <value-context>fileSystem</value-context>
        <value-name>/</value-name>
      </destination-base-dir>
    </bundle-target>
```

ターゲットの潜在的な場所は **<destination-base-dir>**、バンドルのプロビジョニング時にオプションとしてユーザーに提示されます。ユーザーは、そのベースディレクトリーの **下** にあるユーザー定義のディレクトリーにバンドルをデプロイできますが、そのディレクトリー **外** の場所にはデプロイできません。ユーザーが複数のディレクトリーにコンテンツをプロビジョニングする場合は、各ディレクトリーを **<bundle-target>** 定義に追加する必要があります。

例4.26 複数のバンドルベースディレクトリー

```
<bundle-target>
   <destination-base-dir name="Install Directory" description="The top directory where the
JBossAS Server is installed. ">
     <value-context>pluginConfiguration</value-context>
     <value-name>homeDir</value-name>
   </destination-base-dir>
   <destination-base-dir name="Profile Directory" description="The profile configuration directory.">
     <value-context>pluginConfiguration</value-context>
     <value-name>serverHomeDir</value-name>
   </destination-base-dir>
</bundle-target>
```

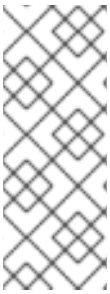
注記

リソースタイプのバンドル定義は 1 つだけですが、コンテンツのデプロイが許可される複数の場所を定義できます。

バグを報告します。

4.8. 拡張例： 非同期可用性チェック

可用性スキャンは、定義されたリソースタイプに対して、リソースプラグイン自体によって実行され、その後エージェントに報告されます。



注記

Async アベイラビリティーコレクターは、プラグイン内のリソースタイプにのみ適用されます。これは、エージェントの可用性スキャン時間を増やすよりもはるかに安全で、パフォーマンスに優れています。可用性スキャン時間はプラグインコンテナーのすべてのプラグインに適用されるため、タイムアウト間隔が長くなると、エージェントは JBoss ON サーバーへの可用性更新の送信を遅延したり、欠落したりすることがあり、プラットフォーム上のすべてのリソースの履歴を誤って保持する可能性があります。

リソースの可用性は、で説明されているように、リソースタイプの起動の一部としてプラグイン内で定義され 「プラグインコンポーネント（HttpComponent.java および HttpServiceComponent.java）の確認」 ます。リソースの可用性状態を取得および設定する方法はです。 **getAvailability()**.

リソースが起動すると、自動的に UP 状態になります。プラグインコンテナーは、エージェントの定期的な可用性とスキャン監視の一部として、その可用性状態をチェックします。

通常、可用性チェックは、非常に高速で、秒数です。プラグインコンテナーは、不正なプラグインがエージェントによって管理されるその他すべてのリソースの可用性レポートを遅延しないようにするために、可用性チェックを 5 秒に制限します。特定のプラグインまたはリソースタイプが 5 秒のタイムアウト期間よりも長いスキャンを行うインスタンスが存在する可能性があります。

カスタムプラグインは、特別な可用性コレクターを使用して **非同期の可用性チェックを実行できます**。基本的に、非同期の可用性チェックでは、リソースコンポーネントは独自の独立したスレッドを作成し、可用性チェックを実行します。そのスレッド内では、可用性チェックは、完了する必要がある限り時間がかかります。可用性チェックは、デフォルトで 1 分ごとに非常に頻繁に実行され、完全なチェックの完了までにかかる場合でも可用性の状態が現在の状態であることを確認することもできます。

コンポーネントによってキャッシュされ、最新の可用性の結果がプラグインコンテナーに報告されます。保存される最後の可用性は、プラグインコンテナーが想定する秒数で非常に迅速に提供されます。

Async 可用性チェックは、以下により実装されます。 **AvailabilityCollectorRunnable** クラス。

アベイラビリティーコレクターはプラグインの 3 つの部分で定義されます。

まず、可用性コレクター自体がデータメンバーとして追加されます。

例4.27 パート 1: コレクター

```
public class YourResourceComponent implements ResourceComponent {

    // your component needs this data member - it is your availability collector
    private AvailabilityCollectorRunnable availCollector;
```

次に、コレクターがリソースの start メソッド 「AvailabilityFacet」 内でに追加されます。ファセットオブジェクトはコレクターを起動し、可用性タイプを返し、可用性チェックのより長い間隔を設定します。ファセットは、リソースに定期的に接続して可用性を確認するものです。

コレクターはリソースコンテキストの一部で、start メソッドと stop メソッドの両方で定義されます。

例4.28 パート 2: アベイラビリティーコレクターの開始

```
    public void start(ResourceContext context) {
        availCollector = resourceContext.createAvailabilityCollectorRunnable(new AvailabilityFacet()
{
            public AvailabilityType getAvailability() {
                // Perform the actual check to see if the managed resource is up or not
                // This method is not on a timer and can return the availability in any amount of time
                // that it needs to take.
                return ...AvailabilityType...;
            }
        }, 60000L); // 1 minute - the minimum interval allowed

        // Now that you've created your availability collector, start it to assign it a thread in the pool.
        availCollector.start();

        // ... and the rest of your component's start method goes here ...
 }


    public void stop() {
        // Stop your availability collector to cancel the collector and kill its thread.
        availCollector.stop();

        // ... and the rest of your component's stop method goes here ...
    }
```

リソースの可用性（非同期の可用性チェックの有無にかかわらず）は、以下で収集されます。 **getAvailability()** メソッド。Async アベイラビリティーコレクターが作成されると、 **getAvailability()** メソッドは、新しい可用性スキャンの実行を試行するのではなく、コレクターに保存されている最後の既知の結果を返す必要があります。

したがって、非同期の可用性チェックにおける最後の設定ポイントは、の戻り値を設定することです。 **getAvailability()** メソッド。

例4.29 パート 2: 最後の既知の利用可能な状態に戻します。

```
    public AvailabilityType getAvailability() {
 // This method quickly returns the last known availability that was recorded
 // by the availability collector.
        return availCollector.getLastKnownAvailability();
    }
}
```

バグを報告します。

第5章 エージェントプラグインの作成： 手順

5.1. ヒント： XSD アノテーションの確認

エージェント（リソース）プラグイン **rhq-configuration.xsd** およびに要素を提供する XSD ファイルがいくつかあります **rhq-plugin.xsd**。

これらのスキーマファイルでは、異なるプロパティーと属性が定義されます。既存のデフォルトスキーマが使用できるようにアノテーションが付けられます。

たとえば、**subCategory** 属性の場合は以下のようになります。

```
<xs:attribute name="subCategory" use="optional">
  <xs:annotation>
    <xs:documentation>
      Resource types can be grouped into subcategories. A subcategory
      defines "like" resource types so they can, for example, be shown together
      in a UI group tab. You can, therefore, define multiple resource types
      and group them together by making their subCategory attributes the same.
    </xs:documentation>
  </xs:annotation>
</xs:attribute>
```

XSD ファイルのアノテーションを使用して読み取り、プラグインの動作を計画し、設定にカスタムスキーマが必要であるかどうかを判断します。

バグを報告します。

5.2. エージェントプラグインの検証

JBoss ON プラグインジェネレーターを使用してエージェントプラグインが生成され、Maven でビルドされた場合、プラグイン自体は Maven を使用して検証できます。JBoss ON/RHQ プラグインソースファイルには、特別な検証クラスがあります。

```
mvn org.rhq:rhq-plugin-validator:rhq-plugin-validate
```

JBoss ON/RHQ プラグインジェネレーターを使用してエージェントプラグインが作成されていない場合は、**<build>** 要素を追加してバリデーターをポイントし、要素を **<pluginRepositories>** 要素にポイントし、Maven リポジトリーを参照します。

```
<build>
  <plugins>
    <plugin>
      <groupId>org.rhq</groupId>
      <artifactId>rhq-core-plugin-validator</artifactId>
      <version>1.0.1-SNAPSHOT</version>
    </plugin>
  </plugins>
</build>
...

...
```

```
<pluginRepositories>
  <pluginRepository>
    <id>jboss</id>
    <name>JBoss Plugin Repository</name>
    <url>http://repository.jboss.org/maven2/</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </pluginRepository>
</pluginRepositories>
...
```

バグを報告します。

5.3. エージェントプラグインの編集に関する注意事項

リソースプラグインの設定を変更するには、**rhq-plugin.xml** ファイルを編集し、プラグインを再ビルドします。

重要

リソースプラグインの編集時に**は、リソースタイプの名前を変更しないでください**。これにより、古いバージョンのプラグインを使用していたリソースとの後方互換性が保たれます。

バグを報告します。

5.4. エージェントプラグインのデプロイ

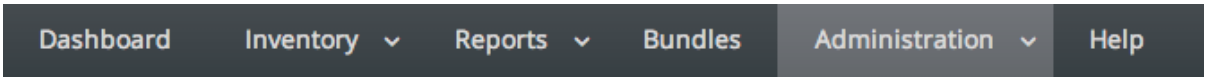
エージェントプラグインファイルは agentInstallDir/**rhq-agent/plugins/** ディレクトリーに保存さ **れ**ます。エージェントプラグインは JBoss ON **サーバー** にアップロードしてデプロイされ、JBoss ON サーバーはエージェントに配布します。サーバー側のプラグインと同様に、エージェントプラグインをローカルの JBoss ON サーバーまたは JBoss ON UI でデプロイできます。

エージェントの起動時にエージェントプラグインが読み込まれます。新しいエージェントプラグインが追加されると、エージェントを再起動するか、プラグインの手動読み込み操作を開始できます。

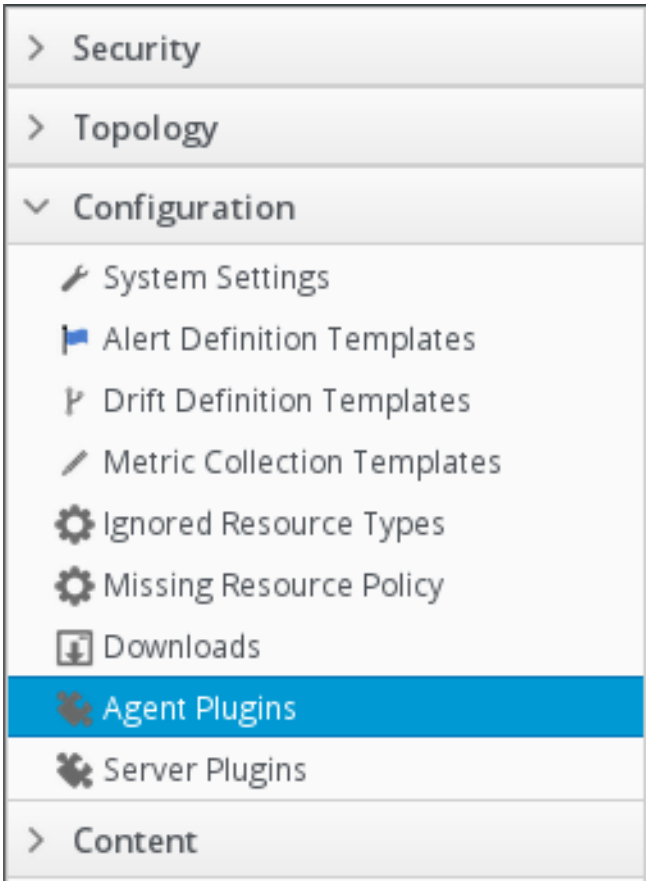
バグを報告します。

5.4.1. リモートでエージェントプラグインをデプロイ

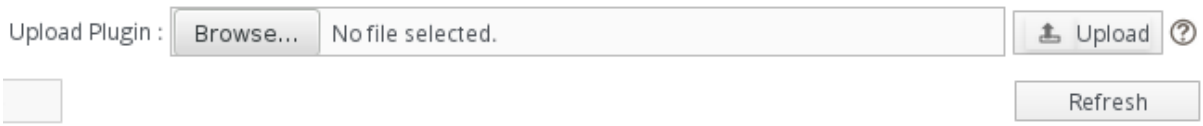
1. トップメニューで、**Administration** タブをクリックします。



2. 左側のナビゲーションバーの **Configuration** ボックスで、**Agent Plugins** リンクをクリックします。



3. ページの下部にある **Upload Plugin** セクションまでスクロールします。
4. **Browse...** ボタンをクリックして、プラグイン JAR ファイルの場所を参照します。
5. デプロイするプラグインがボックスに一覧表示されている場合は、**Upload** ボタンをクリックします。



6. エージェントプラグインのアップロードが完了したら、**Scan For Updates** ボタンを使用します。追加されたプラグインで定義されるリソースタイプのサイズと数によって、更新プロセスに数分かかる場合があります。
7. システム内のエージェントにプラグインの更新をデプロイするには、**Update Plugins On Agents** ボタンを使用します。

注記

プラグインは、各 RHQ Agent を再起動するか、各 RHQ Agent リソースの **Update All Plugins** リソース操作を呼び出してデプロイすることもできます。

- メイン **Inventory** タブを選択します。
- 左側の **Resources** メニューから **All Resources** オプションを選択します。
- 関連するエージェントの名前を選択します。
- **Operations** タブの下にスケジュールを作成します。

バグを報告します。

5.4.2. エージェントプラグインのデプロイ

各サーバーインストールにはトップレベルの **plugins/** ディレクトリーがあります。サーバーは定期的にこのディレクトリーをポーリングします。新規または更新された JAR ファイルはすべてサーバー設定の適切なディレクトリーにコピーされ、元の JAR ファイルが **plugins/** ディレクトリーから削除されます。

JAR ファイルが JBoss ON サーバーと同じホストマシンにある場合、JAR ファイルはその sourceRoot/**plugins/** ディレクトリーにコピーでき 、サーバーによってデプロイされます。

バグを報告します。

5.5. エージェントプラグインの更新

エージェントプラグインは、更新されたプラグイン JAR ファイルをデプロイすることで更新できます。プラグイン記述子には、プラグインパッケージのバージョン番号を含めることができます。サーバーはこのバージョン番号（JAR ファイルの **META-INF/MANIFEST.MF** ファイルにある **Implementation-Version** 設定）を使用して、プラグインの後続のバージョンを特定し、クラウドの JBoss ON サーバーのプラグインを更新します。

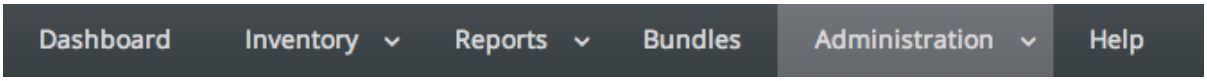
バグを報告します。

5.6. エージェントプラグインの無効化

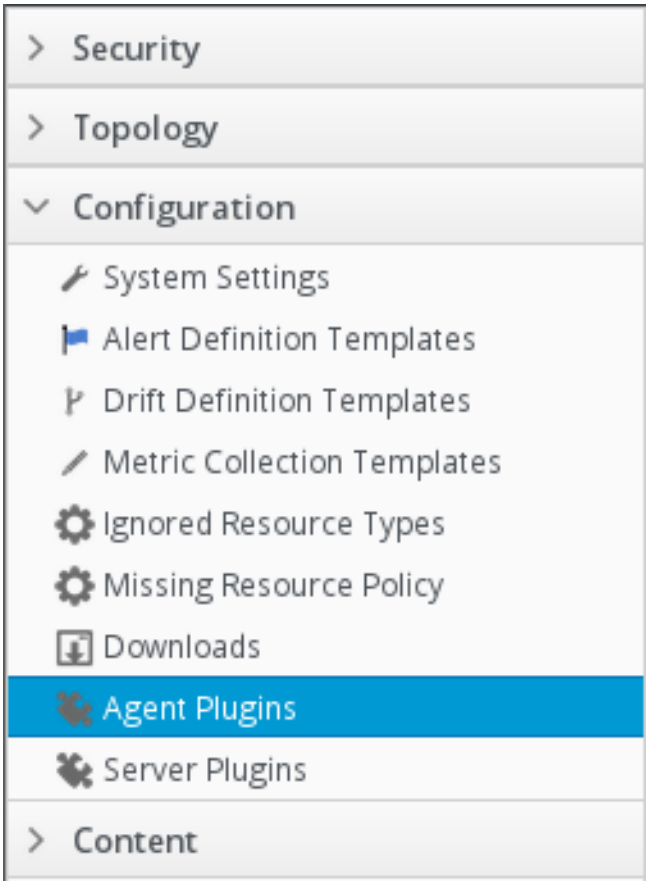
指定がない場合は、すべてのプラグインがデプロイ時に有効になります。プラグインが無効になっていると、クラウドのすべての JBoss ON サーバーの設定にリストされているままとなり、サーバーでロードまたは起動できなくなります。

プラグインを無効にするには、以下を実行します。

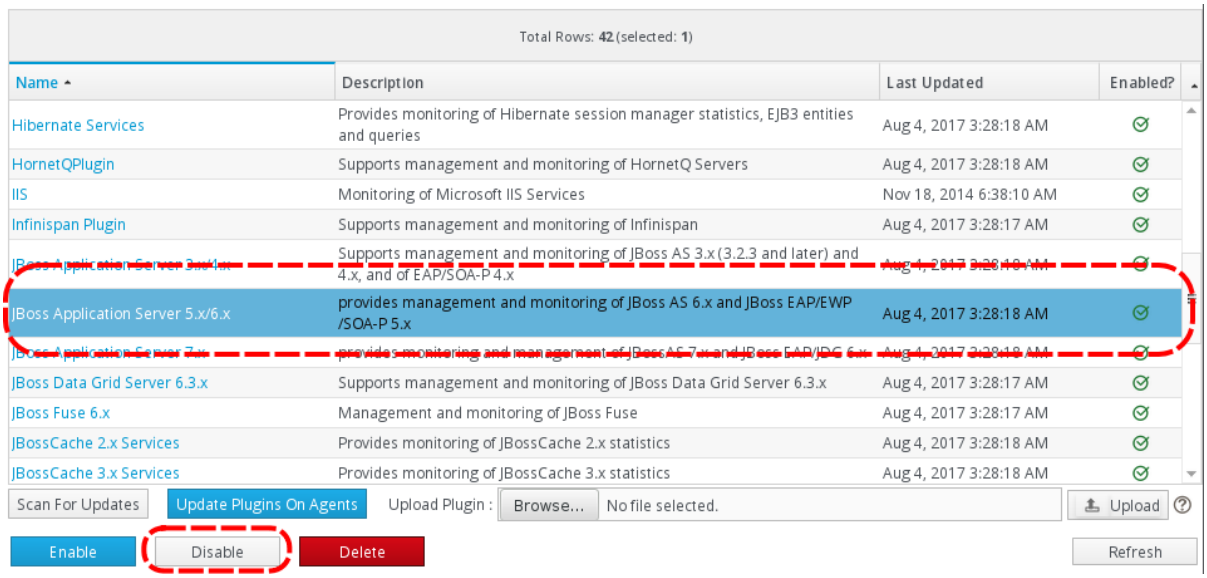
1. トップメニューで、**Administration** タブをクリックします。



2. 左側のナビゲーションバーの **Configuration** ボックスで、**Agent Plugins** リンクをクリックします。



3. 無効にするエージェントプラグインを選択します。
4. **DISABLE** ボタンをクリックします。



エージェントプラグインの管理ページの **'Enabled?'** フィールドは、プラグインが有効かどうかを示します。

無効にしたプラグインは後で再度有効にするには、そのプラグインを選択し、**ENABLE** ボタンをクリックします。

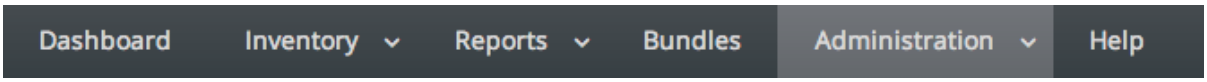
バグを報告します。

5.7. エージェントプラグインの削除

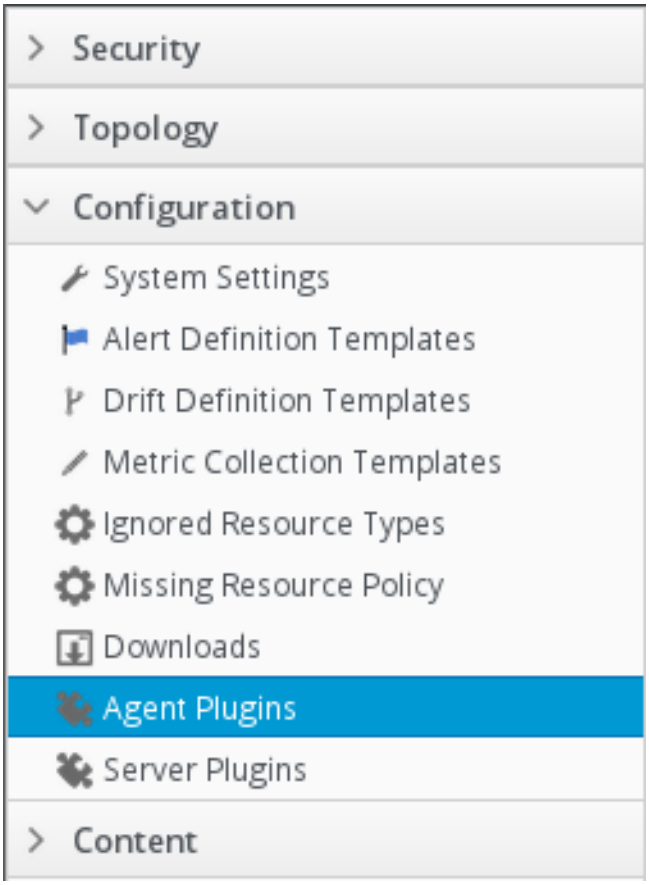
警告

プラグインを削除すると、プラグインに関連付けられたリソースタイプおよびリソースがすべて削除されます。この操作は元に戻すことはできません。

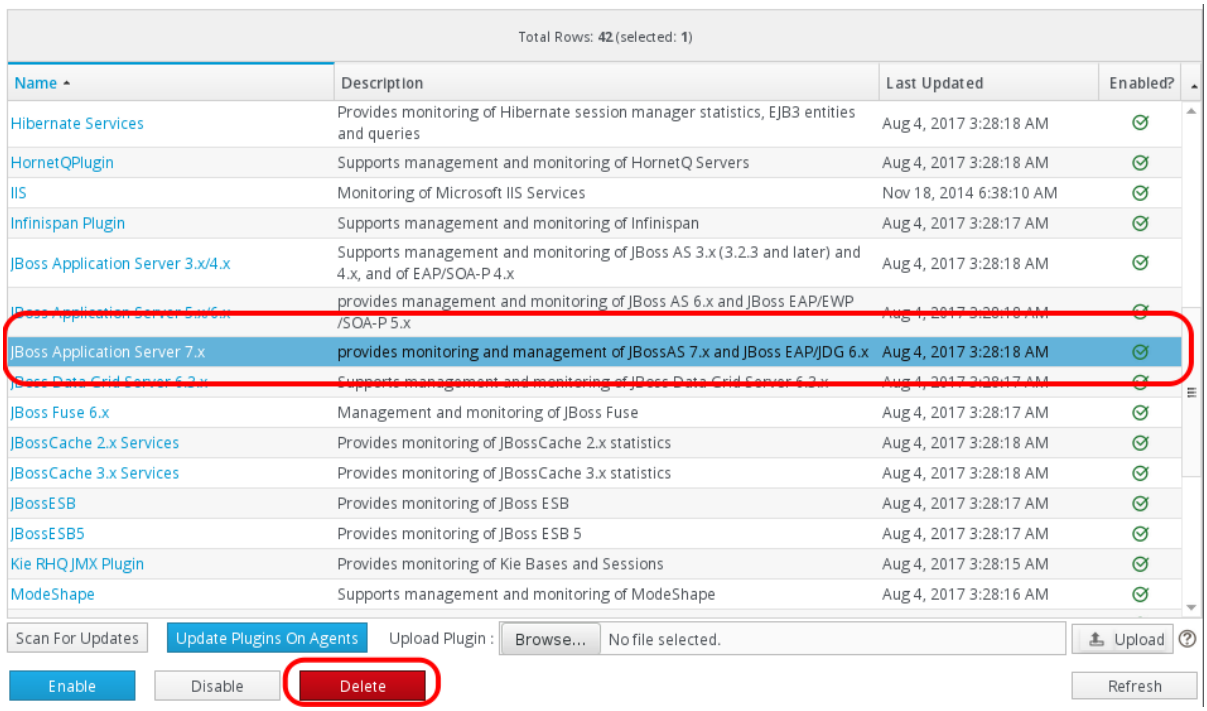
1. トップメニューで、**Administration** タブをクリックします。



2. 左側のナビゲーションバーの **Configuration** ボックスで、**Agent Plugins** リンクをクリックします。



3. 削除するプラグインを選択します。
4. **Delete** ボタンをクリックします。


Total Rows: 42 (selected: 1)
Name
Description
Last Updated
Enabled?
Hibernate Services
Provides monitoring of Hibernate session manager statistics, EJB3 entities and queries
Aug 4, 2017 3:28:18 AM
HornetQPlugin
Supports management and monitoring of HornetQ Servers
Aug 4, 2017 3:28:18 AM
IIS
Monitoring of Microsoft IIS Services
Nov 18, 2014 6:38:10 AM
Infinispan Plugin
Supports management and monitoring of Infinispan
Aug 4, 2017 3:28:17 AM
JBoss Application Server 3.x/4.x
Supports management and monitoring of JBoss AS 3.x (3.2.3 and later) and 4.x, and of EAP/SOA-P 4.x
Aug 4, 2017 3:28:18 AM
JBoss Application Server 5.x/6.x
provides management and monitoring of JBoss AS 6.x and JBoss EAP/EWP /SOA-P 5.x
Aug 4, 2017 3:28:18 AM
JBoss Application Server 7.x
provides monitoring and management of JBossAS 7.x and JBoss EAP/JDG 6.x
Aug 4, 2017 3:28:18 AM
JBoss Data Grid Server 6.2.x
Supports management and monitoring of JBoss Data Grid Server 6.2.x
Aug 4, 2017 3:28:17 AM
JBoss Fuse 6.x
Management and monitoring of JBoss Fuse
Aug 4, 2017 3:28:17 AM
JBossCache 2.x Services
Provides monitoring of JBossCache 2.x statistics
Aug 4, 2017 3:28:18 AM
JBossCache 3.x Services
Provides monitoring of JBossCache 3.x statistics
Aug 4, 2017 3:28:18 AM
JBossESB
Provides monitoring of JBoss ESB
Aug 4, 2017 3:28:17 AM
JBossESB5
Provides monitoring of JBoss ESB 5
Aug 4, 2017 3:28:17 AM
Kie RHQ JMX Plugin
Provides monitoring of Kie Bases and Sessions
Aug 4, 2017 3:28:15 AM
ModeShape
Supports management and monitoring of ModeShape
Aug 4, 2017 3:28:16 AM
Scan For Updates
Update Plugins On Agents
Upload Plugin :
Browse...
No file selected.
Upload
Enable
Disable
Delete
Refresh

バグを報告します。

第6章 エージェント ADVANCED MANAGEMENT PLUG-IN SYSTEM(AMPS)リファレンス

これは、エージェントプラグインの作成に使用される一般的なコンポーネントおよび要素の参照です。

バグを報告します。

6.1. DOMAIN オブジェクト

ドメインオブジェクトには、リソースとタイプを定義する管理インベントリーの基本部分が含まれます。

バグを報告します。

6.1.1. リソースおよびリソースタイプ

リソース は、プラットフォーム、サーバー、またはサービスとしてインベントリー内の単一のエンティティーを表します。プラットフォーム、サーバー、adn サービスのセマンティクスは曖昧であるため、リソースオブジェクトはカテゴリーに関係なくリソースをカプセル化します。

ResourceCategory は、各リソースに関連付けられ、リソースがプラットフォーム、サーバー、またはサービスとみなされるかどうかを示します。

ResourceType は、インベントリーに追加できるリソースインスタンスのタイプを表します。ResourceTypes はプラグイン記述子によって定義されます。もう 1 つの方法は、ResourceType が JBoss ON で管理できるアプリケーションまたはサービスを定義することです。JBoss AS プラグインはエージェントプラグインがデプロイされると、ResourceType が JBoss ON に追加されます。そのため、JBoss AS プラグインを使用すると JBoss サーバーを管理でき、Tomcat プラグインは Tomcat サーバーを管理でき、カスタムプラグインはカスタムアプリケーションを管理できます。

バグを報告します。

6.2. プラグインの FACETS

ファセット は、プラグインライターがプラグインコンテナーに公開するように選択し、最終的に JBoss ON システム全体に対して公開する機能の一部です。プラグインライターは、リソースコンポーネントに、すべて、またはいずれかを実装してください（実装および公開されるより多くのファセットは、より強力で有用なもの）。

バグを報告します。

6.2.1. AvailabilityFacet

このファセットは、基本的な可用性チェックを提供します。これは管理リソースのアップまたはダウンですか？プラグインコンテナーがリソースが実行されているかどうかを認識する必要がある場合、リソースコンポーネントの可用性に関するファセットが要求されます。他のファセットとは異なり、AvailabilityFacet はすべてのリソースコンポーネントで実装する必要があります。ResourceComponent インターフェースが AvailabilityFacet を拡張するため、これを実装する必要があります。オプションで、非同期アベイラビリティーコレクターを使用してアベイラビリティーチェックを実行できます。

バグを報告します。

6.2.2. ConfigurationFacet

このファセットは、リソースコンポーネントで実際の管理リソースの設定を変更する機能を提供します。リソースコンポーネントがこのファセットを実装すると、管理リソースの現在の設定を取得できる機能と変更できる機能があるとします。たとえば、JBoss AS Data Source Service リソースコンポーネントは、データソースの現在の設定（例： JDBC ドライバー、JNDI 名、接続プールサイズなど）をユーザーに報告し、ユーザーがこれらの設定を変更することができるため、ConfigurationFacet を実装します。

バグを報告します。

6.2.3. ContentFacet

リソースには、デプロイされたソフトウェアやソフトウェアの部分、その他のコンテンツなど、これらに関連するコンテンツが含まれる場合があります。このシステムは、これらのソフトウェアの部分をインベントリーし、インストールし、削除するために使用できます。デプロイされたコンテンツは、JBoss EAP で EAR および WAR アプリケーションまたはライブラリーおよびデプロイメントファイルになります。プラグインは、このシステムにおける任意のタイプのコンテンツをサポートします。

リソースには、設定ファイル、デプロイメントファイルなど、追加のファイル（「コンテンツ」）を関連付けることができます。コンテンツが関連付けられているこれらのリソースは ContentFacet を実装し、そのコンテンツの作成、削除、管理に役立ちます。

バグを報告します。

6.2.4. ManualAddFacet

このファセットは、JBoss ON GUI を使用してインベントリーに手動で追加できるリソースタイプの ResourceDiscoveryComponent クラスにより実装する必要があります。さらに、プラグイン記述子の該当する server またはサービス要素には、supportManualAdd="true" 属性が含まれている必要があります。手動による追加は、特定のリソースを何らかの理由で自動検出できない場合に役立ちます。

バグを報告します。

6.2.5. MeasurementFacet

このファセットは、管理されたリソースから測定データを収集し、そのデータをサーバーに戻すコンポーネントの機能を公開します。測定ファセットが機能するには、プラグインは、プラグイン記述子にリソースコンポーネントのリソースタイプのメトリクス定義を 1 つ以上定義する必要があります。リソースコンポーネントは、測定コレクションのスケジュール方法やデータの収集タイミングに関する懸念は必要ありません。MeasurementFacet でリソースコンポーネントが必要とする唯一の問題は、実際の管理リソースにアクセスして要求されたデータを収集することです。プラグインコンテナーはすべての測定コレクションスケジュールを管理し、時間が適切である場合にのみリソースの MeasurementFacet に呼び出しを行い、その時点で収集する必要があるメトリクスのみを要求するだけです。

measurement ファセットは、JBoss ON GUI コンソールで確認できる測定データのグラフを提供するものです。

バグを報告します。

6.2.6. OperationFacet

このファセットを使用すると、リソースコンポーネントは管理リソース自体で操作（制御アクション）を実行できます。たとえば、JBoss AS Server リソースコンポーネントは JBoss AS サーバーを起動および停止する機能を提供します。その他の操作例は、データソースの接続プールをクリアしたり、リ

ソースを空のデータキャッシュに依頼したりできます。リソースコンポーネントは、管理対象リソースを JBoss ON ユーザーに操作として公開できます。

バグを報告します。

6.2.7. ResourceFactoryFacet

一部のリソースコンポーネントは、子リソースの作成および削除をサポートします（たとえば、JBoss AS サーバーを表すリソースコンポーネントは *-ds.xml ファイルを作成して削除して JBoss AS データソースサービスを作成および削除できます）。このファセットはこの機能を公開します。

バグを報告します。

6.2.8. SupportFacet

管理リソースをサポートするには、ログファイル、データファイル、設定ファイルの内容など、組織が管理リソースに関する知識を希望するデータがあります。SupportFacet は、管理リソースのこの「サポートおよびメンテナンス」ビューにフックを提供します。

バグを報告します。

6.3. プラグインコンポーネント

6.3.1. ResourceDiscoveryComponent

Discovery コンポーネントは、プラグイン作成者が作成する実装で、実際の管理対象リソースの検出を実行します。検出コンポーネントのジョブは、プラットフォームをスキャンします（つまり、エージェント/プラグインコンテナー/プラグインが実行中であるマシン）。検出コンポーネントは、直接管理しているリソースを見つけます。JBoss AS プラグイン検出コンポーネントは、すべての Apache Web サーバーを検出する責任ではないため、JBoss AS リソースのみを見つける必要があります（Apache プラグインへの Apache Web サーバーの検出を削除します）。

検出コンポーネントは、プラグインコンテナーにより適切なタイミングでリソースを探すように指示されます。プラグインコンテナーがより多くのリソースを検出するように検出コンポーネントを要求すると、**ResourceDiscoveryContext** オブジェクトで検出コンポーネントに送信されます。このコンテキストには、コンポーネントが新規リソースを見つけ、作成するために必要なすべての情報が含まれます。Discovery コンテキストは、検出コンポーネントの代わりにプラグインコンテナーが新しいリソースを検出できる場合に、検出コンポーネントにリソースを注入するために使用されます。プラグインコンテナーは、プラグインの記述子を介して適切なメタデータが提供される場合にのみ、自動的に検出できます。

バグを報告します。

6.3.2. ResourceComponent

リソースコンポーネントは、実際の管理されたリソースを表すプラグインの抽象化です。リソースコンポーネントは、プラグインコンテナーによって管理されるライフサイクルがステートフルです。

プラグインコンテナーは、適切なタイミングでリソースコンポーネントを起動し、停止します。リソースコンポーネントが起動すると、通常は基盤のリソース（これが表す管理リソース）に接続し、プラグインコンテナーが停止するまでその接続を維持します（これは、プラグインライターが自由に変更できるようにする実装詳細です）。リソースコンポーネントの実装は、プラグインコンテナーが管理リソー

スの利用可能を要求する方法（「このリソースを稼働させるか、停止させるか？」）を提供し、プラグインライターが公開するオプションの機能ファセットにアクセスできます。以下のファセットではさらに表示されます。

バグを報告します。

6.4. ネイティブシステム情報アクセス

すべてのプラグインは、プラグインが実行しているマシンの詳細について、プラグインコンポーネントに基礎となるオペレーティングシステムに要求できるようにするネイティブライブラリーのセットにアクセスできます。これらの機能の一部は、ハードウェアおよび OS プラットフォームすべてでは使用できません。利用可能なネイティブライブラリーを持つプラットフォームのみが、以下のすべての機能をサポートすることができます。ただし、ネイティブライブラリーが利用できないプラットフォームでは、機能セットは限定されます。

バグを報告します。

6.4.1. SystemInfoFactory および SystemInfo

プラグインは、プラグインが実行しているハードウェア/OS プラットフォームに固有の SystemInfo オブジェクトにアクセスできます。プラグインがコンテキスト（ **ResourceDiscoveryContext** またはのいずれか **ResourceContext**）から SystemInfo オブジェクトを取得したら、ネイティブライブラリーに呼び出しを行い、オペレーティングシステムから要求されたデータを取得するオブジェクトに呼び出しを行うことができます。ネイティブライブラリーが利用できない場合、SystemInfo は、SystemInfo インターフェースで定義されたメソッドの一部の純粋な Java 実装でサポートされます（このインターフェースの JavaSystemInfo 実装を参照してください）。純粋な Java 実装でサポートされていないメソッドはをスローし **UnsupportedOperationException**ます。

バグを報告します。

6.4.2. ProcessInfoQuery

SystemInfo インターフェースは、オペレーティングシステムのプロセステーブルをプローブする機能を提供します。ResourceDiscoveryComponent 実装は、実行中のプロセスの一覧をスキャンし、検出を担当する管理リソースを自動検出できるかどうかを判断できるために役立ちます。

ProcessInfoQuery オブジェクトを使用すると、PIQL(Process Info Query Language)で定義される特定の基準セットに一致するプロセスを見つけることができます。process-scan タグを使用してプラグインの記述子に事前定義済みの PIQL クエリーを設定し、プラグインコンテナーがプラグインの代わりにプロセステーブルをスキャンできるようにします。

注記

PIQL の構文と使用方法は、「 ProcessInfoQuery」 http://git.fedorahosted.org/git/rhq/rhq.git?p=rhq/rhq.git;a=blob;hb=master;f=modules/core/native-system/src/main/java/org/rhq/core/system/pquery/ProcessInfoQuery_java を参照してください。

バグを報告します。

第7章 ドキュメント情報

本ガイドは、JBoss ON のユーザーおよび管理者向けの全ガイドの一部です。目的は明確性、完全性、使いやすさです。

バグを報告します。

7.1. フィードバック提供

本ガイドにエラーがある場合や、ドキュメントを改善する方法がある場合は、お知らせください。Bugs は、Bugzilla(http://bugzilla.redhat.com/bugzilla)のドキュメントのドキュメントに対して報告できます。できるだけ具体的にバグレポートを作成してください。したがって、問題の修正をより効果的に行えます。

1. **JBoss** products グループを選択します。
2. 一覧 **Red Hat JBoss Operations Network** から選択します。
3. コンポーネントをに設定し **Documentation**ます。
4. バージョン番号を 3.3 に設定します。
5. エラーの場合は、ページ番号（PDF ファイル用）または URL（HTML の場合）を付け、問題（正しくない手順や typo など）の説明を付与します。

   機能強化のために、追加する必要のある情報と理由を示します。
6. バグの明確なタイトルを指定します。たとえば、**"Incorrect command example for setup script options"** はよりも優れてい **"Bad example"**ます。

新しいセクション、修正、改良、改良、機能拡張、ドキュメントや新しいスタイルの配信方法などに対するフィードバックは、あらゆるご意見やご意見をお寄せください。本ガイドの作成者から Red Hat Customer Content Services に直接お問い合わせください。

バグを報告します。

付録A ドキュメント履歴

改訂 3.3.2-7 **Thu Jun 25 2015** **Jared イタリア**
JBoss ON 3.3.2 リリース向けに準備済み。新しいスプラッシュページ用にブックブランドに若干変更を加える

改訂 3.3.1-1 **Wed Feb 18 2015** **Jared イタリア**
JBoss ON 3.3.1 リリースの準備

改訂 3.3-10 **Mon Nov 17 2014** **Jared イタリア**
JBoss ON 3.3 GA 向けに更新されました。