Red Hat Fuse 7.13

Apache Karaf セキュリティーガイド

Apache Karaf コンテナーのセキュリティー保護

Last Updated: 2024-06-12

Red Hat Fuse 7.13 Apache Karaf セキュリティーガイド

Apache Karaf コンテナーのセキュリティー保護

法律上の通知

概要

このガイドでは、Red Hat Fuse コンテナー、Web コンソール、メッセージブローカー、ルーティングおよび統合コンポーネント、Web および RESTful サービスを保護する方法について説明し、LDAP 認証に関するチュートリアルを提供します。

目次

多様性を受け入れるオープンソースの強化

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

第1章 セキュリティーアーキテクチャー

概要

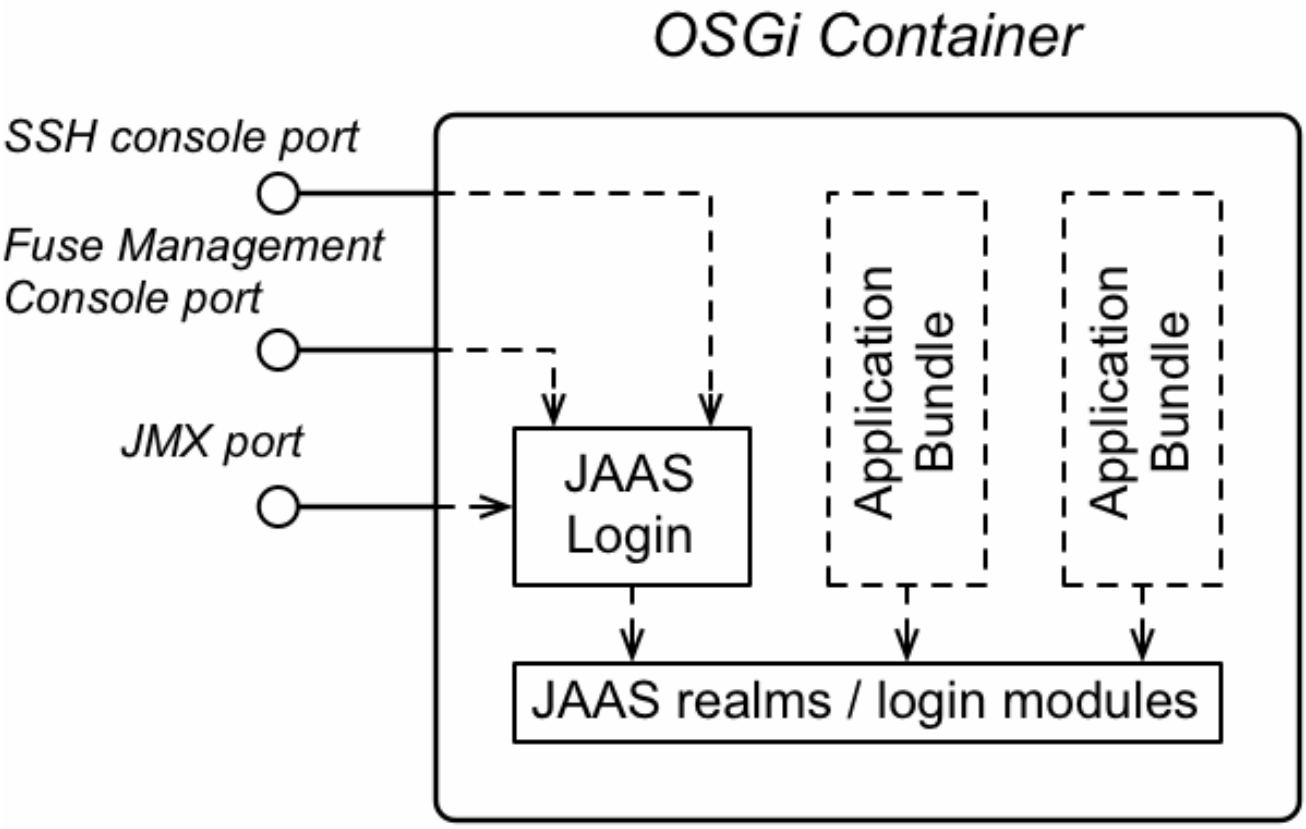
OSGi コンテナーでは、さまざまなセキュリティー機能をサポートするアプリケーションをデプロイできます。現在、Java 認証および認可サービス (JAAS) のみが、共通するコンテナー全体のインフラストラクチャーに基づいています。その他のセキュリティー機能は、コンテナーにデプロイされた個々の製品およびコンポーネントによって個別に提供されます。

1.1. OSGI コンテナーセキュリティー

概要

図1.1「OSGi コンテナーセキュリティーアーキテクチャー」 は、コンテナー全体で使用され、コンテナーにデプロイされているすべてのバンドルにアクセスできるセキュリティーインフラストラクチャーの概要を示しています。この一般的なセキュリティーインフラストラクチャーは、現在、JAAS レルム (またはログインモジュール) をすべてのアプリケーションバンドルで利用できるようにするためのメカニズムで設定されています。

図1.1 OSGi コンテナーセキュリティーアーキテクチャー



JAAS レルム

JAAS レルムまたはログインモジュールは、Java Authentication and Authorization Service (JAAS) の仕様で定義されているように、Java アプリケーションに認証および認可データを提供するプラグインモジュールです。

Red Hat Fuse は、JAAS ログインモジュール (Spring またはブループリントファイルのいずれか) を定義するための特別なメカニズムをサポートします。これにより、コンテナー内のすべてのバンドルにログインモジュールにアクセスできるようになります。これにより、OSGi コンテナーで実行されている

複数のアプリケーションが、セキュリティーデータを単一の JAAS レルムに統合することが容易になります。

karaf レルム

OSGi コンテナーには、事前定義された JAAS レルムである **karaf** レルムがあります。Red Hat Fuse は、**karaf** レルムを使用して、OSGi ランタイムのリモート管理、Fuse 管理コンソール、および JMX 管理の認証を提供します。**karaf** レルムは、認証データが **InstallDir/etc/users.properties** ファイルに保存される簡単なファイルベースのリポジトリーを使用します。

独自のアプリケーションで **karaf** レルムを使用できます。**karaf** を、使用する JAAS レルムの名前として設定するだけです。その後、アプリケーションは **users.properties** ファイルからのデータを使用して認証を実行します。

コンソールポート

Karaf クライアントでコンソールポートに接続するか、Karaf **ssh:ssh** コマンドを使用して、OSGi コンテナーをリモートで管理できます。コンソールポートは、**karaf** レルムに接続する JAAS ログイン機能によってセキュア化されます。コンソールポートへの接続を試みると、**karaf** レルムからアカウントのいずれかに一致する必要のあるユーザー名とパスワードの入力が要求されます。

JMX ポート

JMX ポートに接続することで (たとえば、Java の JConsole を使用して) OSGi コンテナーを管理できます。JMX ポートは、**karaf** レルムに接続する JAAS ログイン機能によっても保護されます。

アプリケーションバンドルと JAAS セキュリティー

OSGi コンテナーにデプロイするすべてのアプリケーションバンドルは、コンテナーの JAAS レルムにアクセスできます。アプリケーションバンドルは、名前を使用して既存の JAAS レルムの 1 つを参照します (これは JAAS ログインモジュールのインスタンスに対応します)。

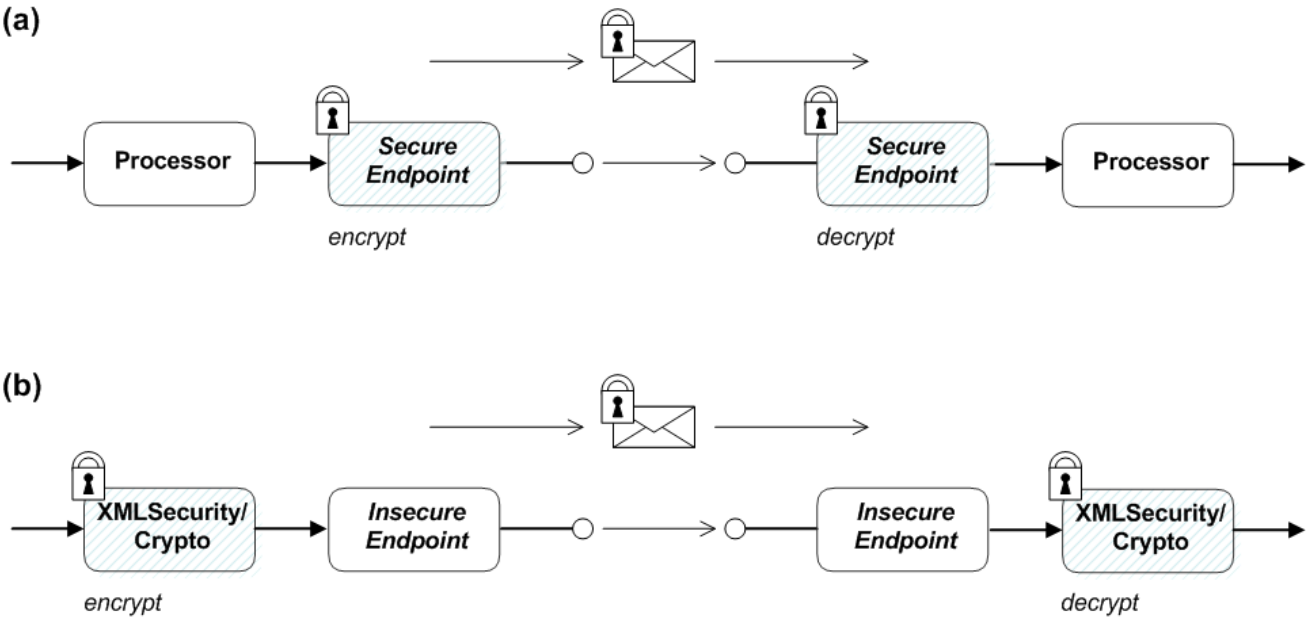
ただし、JAAS レルムは、OSGi コンテナー独自のログイン設定メカニズムを使用して定義されている必要があります。デフォルトでは、Java は単純なファイルベースのログイン設定実装を提供しますが、OSGi コンテナーのコンテキストでこの実装は **使用できません**。

1.2. APACHE CAMEL のセキュリティー

概要

図1.2「Apache Camel のセキュリティーアーキテクチャー」 は、Apache Camel のエンドポイント間でメッセージを安全にルーティングするための基本的なオプションの概要を示しています。

図1.2 Apache Camel のセキュリティーアーキテクチャー



Apache Camel のセキュリティーの代替手段

図1.2「Apache Camel のセキュリティーアーキテクチャー」 に示すように、メッセージを保護するための次のオプションがあります。

- **エンドポイントセキュリティー** - パート (a) は、セキュアなエンドポイントを持つ 2 つのルート間で送信されるメッセージを示しています。左側のプロデューサーエンドポイントは、右側のコンシューマーエンドポイントへの安全な接続を開きます (通常は SSL/TLS を使用)。このシナリオでは、両方のエンドポイントがセキュリティーをサポートしています。
  エンドポイントセキュリティーを使用すると、通常、何らかの形式のピア認証 (場合によっては認可) を実行できます。

- **ペイロードセキュリティー** - パート (b) は、エンドポイントが両方とも **セキュアではない** 2 つのルート間で送信されるメッセージを示しています。この場合、メッセージを不正なスヌーピングから保護するには、送信前にメッセージを暗号化し、受信後にメッセージを復号する **ペイロードプロセッサー** を使用します。
  ペイロードセキュリティーには、いかなる種類の認証または認可メカニズムも **提供しない** という制限があります。

エンドポイントセキュリティー

セキュリティー機能をサポートする Camel コンポーネントがいくつかあります。ただし、これらのセキュリティー機能は、Camel コア **ではなく**、個々のコンポーネントによって実装されていることに注意してください。つまり、サポートされるセキュリティー機能の種類とその実装の詳細は、コンポーネントごとに異なります。現在セキュリティーをサポートしている Camel コンポーネントの一部は次のとおりです。

- JMS および ActiveMQ - クライアントからブローカーへの通信およびブローカーからブローカーへの通信のための SSL/TLS セキュリティーと JAAS セキュリティー。
- Jetty - HTTP Basic 認証と SSL/TLS セキュリティー。
- CXF - SSL/TLS セキュリティーと WS-Security。
- Crypto - メッセージの整合性を保証するために、デジタル署名を作成および検証します。

- Netty - SSL/TLS セキュリティー。
- MINA - SSL/TLS セキュリティー。
- Cometd - SSL/TLS セキュリティー。
- glogin および gauth - Google アプリケーションのコンテキストでの認可。

ペイロードのセキュリティー

Apache Camel は、以下のペイロードセキュリティー実装を提供します。この実装では、暗号化手順および復号化手順は **marshal()** および **unmarshal()** 操作でデータ形式として公開されます。

- 「XMLSecurity データ形式」
- 「Crypto データ形式」

XMLSecurity データ形式

XMLSecurity データ形式は、XML ペイロードを暗号化するために特別に設計されています。このデータ形式を使用する場合、暗号化する XML 要素を指定できます。デフォルトの動作では、**すべての** XML 要素を暗号化します。この機能は、対称暗号化アルゴリズムを使用します。

詳細については、http://camel.apache.org/xmlsecurity-dataformat.html を参照してください。

Crypto データ形式

Crypto データ形式は、あらゆる種類のペイロードを暗号化できる汎用暗号化機能です。これは Java Cryptographic Extension に基づいており、対称 (共有キー) 暗号化と復号化のみを実装します。

詳細については、http://camel.apache.org/crypto.html を参照してください。

第2章 APACHE KARAF コンテナーのセキュリティー保護

概要

Apache Karaf コンテナーは、JAAS を使用してセキュリティー保護されています。JAAS レルムを定義することで、ユーザー認証情報の取得に使用されるメカニズムを設定できます。デフォルトのロールを変更することで、コンテナーの管理インターフェイスへのアクセスを調整することもできます。

2.1. JAAS 認証

概要

Java Authentication and Authorization Service (JAAS) は、Java アプリケーションに認証を実装するための一般的なフレームワークを提供します。認証の実装はモジュール式であり、個々の JAAS モジュール (またはプラグイン) が認証の実装を提供します。

JAAS の背景情報については、JAAS リファレンスガイド を参照してください。

2.1.1. デフォルトの JAAS レルム

このセクションでは、Karaf コンテナー内のデフォルト JAAS レルムのユーザーデータを管理する方法について説明します。

デフォルトの JAAS レルム

Karaf コンテナーには事前定義された JAAS レルムである **karaf** レルムがあります。これはデフォルトで、コンテナーのすべての側面をセキュアにするために使用されます。

アプリケーションを JAAS と統合する方法

独自のアプリケーションで **karaf** レルムを使用できます。**karaf** を、使用する JAAS レルムの名前として設定するだけです。

デフォルトの JAAS ログインモジュール

Karaf コンテナーを初めて起動する場合、**karaf** デフォルトレルムを使用するように設定されています。このデフォルト設定では、**karaf** レルムは 5 つの JAAS ログインモジュールをデプロイし、同時に有効にします。デプロイされたログインモジュールを表示するには、以下のように **jaas:realms** コンソールコマンドを入力します。

```
Index │ Realm Name │ Login Module Class Name
──────┼────────────┼────────────────────────────────────────────────────────────
1     │ karaf      │ org.apache.karaf.jaas.modules.properties.PropertiesLoginModule
2     │ karaf      │ org.apache.karaf.jaas.modules.publickey.PublickeyLoginModule
3     │ karaf      │ org.apache.karaf.jaas.modules.audit.FileAuditLoginModule
4     │ karaf      │ org.apache.karaf.jaas.modules.audit.LogAuditLoginModule
5     │ karaf      │ org.apache.karaf.jaas.modules.audit.EventAdminAuditLoginModule
```

ユーザーがログインしようとすると、リスト順に 5 つのモジュールで認証されます。各モジュールのフラグ値は、認証が成功するためにモジュールが正常に完了する必要があるかどうかを指定します。フラグ値は、モジュールの完了後に認証プロセスを停止するか、次のモジュールに進むかも指定します。

Optional フラグは、5 つの認証モジュールすべてに設定されます。**Optional** フラグ設定により、現在のモジュールが正常に完了するかどうかにかかわらず、認証プロセスが常に 1 つのモジュールから次のモジュールに渡されます。Karaf JAAS レルムのフラグ値はハードコーディングされており、変更できません。フラグの詳細については、表2.1「JAAS モジュールを定義するためのフラグ」 を参照してください。

重要

Karaf コンテナーでは、プロパティーログインモジュールと公開鍵ログインモジュールの **両方** が有効になっています。JAAS はユーザーを認証するときに、まずプロパティーログインモジュールを使用してユーザーを認証しようとします。それが失敗すると、公開鍵ログインモジュールを使用してユーザーの認証を試みます。そのモジュールも失敗すると、エラーが発生します。

2.1.1.1. 認証監査ロギングモジュール

Karaf コンテナーのデフォルトモジュールのリスト内では、最初の 2 つのモジュールのみがユーザー ID の確認に使用されます。残りのモジュールは、成功したログイン試行と失敗したログイン試行の監査証跡をログに記録するために使用されます。デフォルトのレルムには、次の監査ロギングモジュールが含まれています。

org.apache.karaf.jaas.modules.audit.LogAuditLoginModule

このモジュールは、**etc/org.ops4j.pax.logging.cfg** ファイル内の Pax ロギングインフラストラクチャーに設定されたロガーを使用して、認証の試行に関する情報を記録します。詳細については、JAAS Log Audit Login Module を参照してください。

org.apache.karaf.jaas.modules.audit.FileAuditLoginModule

このモジュールは、認証の試行に関する情報を、指定したファイルに直接記録します。ロギングインフラストラクチャーは使用しません。詳細については、JAAS File Audit Login Module を参照してください。

org.apache.karaf.jaas.modules.audit.EventAdminAuditLoginModule

このモジュールは、OSGi Event Admin サービスを使用して認証の試行を追跡します。

プロパティーログインモジュールでのユーザーの設定

プロパティーログインモジュールは、ユーザー名/パスワードのクレデンシャルをフラットファイル形式で保存するために使用されます。プロパティーログインモジュールで新規ユーザーを作成するには、テキストエディターを使用して **InstallDir/etc/users.properties** ファイルを開き、以下の構文の行を追加します。

```
Username=Password[,UserGroup|Role][,UserGroup|Role]...
```

たとえば、パスワード **topsecret** およびロール **admin** で **jdoe** ユーザーを作成するには、以下のようなエントリーを作成します。

```
jdoe=topsecret,admin
```

admin ロールは、**jdoe** ユーザーに完全な管理権限を付与します。

プロパティーログインモジュールでのユーザーグループの設定

ユーザーに直接ロールを割り当てる代わりに (またはそれに加えて)、プロパティーログインモジュールの **user groups** にユーザーを追加するオプションもあります。プロパティーログインモジュールでユー

ザーグループを作成するには、テキストエディターを使用して **InstallDir/etc/users.properties** ファイルを開き、以下の構文で行を追加します。

```
_g_\:GroupName=Role1,Role2,...
```

たとえば、ロール **group** および **admin** で **admingroup** ユーザーグループを作成するには、以下のようなエントリーを作成します。

```
_g_\:admingroup=group,admin
```

以下のユーザーエントリーを作成して、**majorclanger** ユーザーを **admingroup** に追加します。

```
majorclanger=secretpass,_g_:admingroup
```

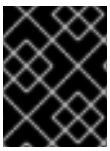
公開鍵ログインモジュールの設定

公開鍵ログインモジュールは、SSH 公開鍵のクレデンシャルをフラットファイル形式で保存するために使用されます。公開鍵ログインモジュールで新規ユーザーを作成するには、テキストエディターを使用して **InstallDir/etc/keys.properties** ファイルを開き、以下の構文の行を追加します。

```
Username=PublicKey[,UserGroup|Role][,UserGroup|Role]...
```

たとえば、以下のエントリーを 1 行で **InstallDir/etc/keys.properties** ファイルに追加することで、**admin** ロールで **jdoe** ユーザーを作成できます。

```
jdoe=AAAAB3NzaC1kc3MAAACBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2USZpRV1AIlH
7WT2NWPq/xfW6MPbLm1Vs14E7gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1fnfqimFQ8E+4
P208UewwI1VBNaFpEy9nXzrith1yrv8iIDGZ3RSAHHAAAAFQCXYFCPFSMLzLKSuYKi64QL8Fgc9QA
AAnEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCBgLRJFnEj6E
woFhO3zwkyjMim4TwWeotifI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImog9/hWuWfB
pKLZl6Ae1UlZAFMO/7PSSoAAACBAKKSU2PFl/qOLxIwmBZPPIcJshVe7bVUpFvyl3BbJDow8rXfskl8w
O63OzP/qLmcJM0+JbcRU/53Jj7uyk31drV2qxhIOsLDC9dGCWj47Y7TyhPdXh/0dthTRBy6bqGtRPxGa
7gJov1xm/UuYYXPIUR/3x9MAZvZ5xvE0kYXO+rx,admin
```



重要

ここで、**id_rsa.pub** ファイルの内容をすべて挿入しないでください。公開鍵自体を表す記号のブロックだけを挿入します。

公開鍵ログインモジュールでのユーザーグループの設定

ユーザーに直接ロールを割り当てる代わりに (またはそれに加えて)、公開鍵ログインモジュールの **user groups** にユーザーを追加するオプションもあります。公開鍵ログインモジュールでユーザーグループを作成するには、テキストエディターを使用して **InstallDir/etc/keys.properties** ファイルを開き、以下の構文の行を追加します。

```
_g_\:GroupName=Role1,Role2,...
```

たとえば、ロール **group** および **admin** で **admingroup** ユーザーグループを作成するには、以下のようなエントリーを作成します。

```
_g_\:admingroup=group,admin
```

以下のユーザーエントリーを作成して、**jdoe** ユーザーを **admingroup** に追加します。

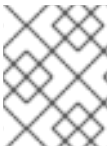
```
jdoe=AAAAB3NzaC1kc3MAAACBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2USZpRV1AIlH
7WT2NWPq/xfW6MPbLm1Vs14E7gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1fnfqimFQ8E+4
P208UewwI1VBNaFpEy9nXzrith1yrv8iIDGZ3RSAHHAAAAFQCXYFCPFSMLzLKSuYKi64QL8Fgc9QA
AAnEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCBgLRJFnEj6E
woFhO3zwkyjMim4TwWeotifI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImog9/hWuWfB
pKLZI6Ae1UIZAFMO/7PSSoAAACBAKKSU2PFl/qOLxIwmBZPPIcJshVe7bVUpFvyl3BbJDow8rXfskI8w
O63OzP/qLmcJM0+JbcRU/53Jj7uyk31drV2qxhIOsLDC9dGCWj47Y7TyhPdXh/0dthTRBy6bqGtRPxGa
7gJov1xm/UuYYXPIUR/3x9MAZvZ5xvE0kYXO+rx,_g_:admingroup
```

保存されたパスワードの暗号化

デフォルトでは、パスワードはプレインテキスト形式で **InstallDir/etc/users.properties** ファイルに保存されます。このファイルでパスワードを保護するには、管理者のみが読み取ることができるように **users.properties** ファイルのファイル権限を設定する必要があります。追加の保護を提供するために、オプションで、メッセージダイジェストアルゴリズムを使用して保存されたパスワードを暗号化できます。

パスワード暗号化機能を有効にするには、**InstallDir/etc/org.apache.karaf.jaas.cfg** ファイルを編集して、コメントで説明されているように暗号化プロパティーを設定します。たとえば、次の設定では、MD5 メッセージダイジェストアルゴリズムを使用した基本的な暗号化が有効になります。

```
encryption.enabled = true
encryption.name = basic
encryption.prefix = {CRYPT}
encryption.suffix = {CRYPT}
encryption.algorithm = MD5
encryption.encoding = hexadecimal
```



注記

org.apache.karaf.jaas.cfg ファイルの暗号化設定は、Karaf コンテナーのデフォルトの **karaf** レルム**のみに**適用されます。カスタムレルムには影響しません。

パスワード暗号化の詳細については、「保存されたパスワードの暗号化」 を参照してください。

デフォルトレルムのオーバーライド

JAAS レルムをカスタマイズする場合、最も便利なアプローチは、より高いランクの **karaf** レルムを定義してデフォルトの **karaf** レルムをオーバーライドすることです。これにより、すべての Red Hat Fuse セキュリティーコンポーネントがカスタムレルムを使用するように切り替わります。カスタム JAAS レルムを定義およびデプロイする方法の詳細については、「JAAS レルムの定義」 を参照してください。

2.1.2. JAAS レルムの定義

OSGi コンテナーで JAAS レルムを定義する場合、従来の JAAS ログイン設定 ファイルに定義を置くことは **できません**。代わりに、OSGi コンテナーは、Blueprint 設定ファイルで JAAS レルムを定義するために特別な **jaas:config** 要素を使用します。このように定義された JAAS レルムは、コンテナーにデプロイされた **すべて** のアプリケーションバンドルで利用できるようになり、コンテナー全体で JAAS セキュリティーインフラストラクチャーを共有できるようになります。

Namespace

jaas:config 要素は、**http://karaf.apache.org/xmlns/jaas/v1.0.0** 名前空間で定義されています。JAAS レルムを定義するときは、例2.1「JAAS ブループリント名前空間」 に示された行を含める必要があります。

例2.1 JAAS ブループリント名前空間

```
xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
```

JAAS レルムの設定

jaas:config 要素の構文は 例2.2「ブループリント XML での JAAS レルムの定義」 に示されています。

例2.2 ブループリント XML での JAAS レルムの定義

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0">

  <jaas:config name="JaasRealmName"
               rank="IntegerRank">
    <jaas:module className="LoginModuleClassName"
                 flags="[required|requisite|sufficient|optional]">
      Property=Value
      ...
    </jaas:module>
    ...
    <!-- Can optionally define multiple modules -->
    ...
  </jaas:config>

</blueprint>
```

要素は次のように使用されます。

jaas:config

JAAS レルムを定義します。この要素は以下の属性を持ちます。

- **name** – JAAS レルムの名前を指定します。
- **rank** – JAAS レルム間で命名の競合を解決するためのオプションのランクを指定します。2 つ以上の JAAS レルムが同じ名前で登録されている場合、OSGi コンテナーは常に最高ランクのレルムインスタンスを選択します。デフォルトのレルム **karaf** を上書きする場合は、以前にインストールされた **karaf** レルムをすべてオーバーライドするように、**rank** を **100** 以上に設定する必要があります。

jaas:module

現在のレルムで JAAS ログインモジュールを定義します。**jaas:module** には以下の属性があります。

- **className** – JAAS ログインモジュールの完全修飾クラス名。指定されたクラスは、バンドルクラスローダーから利用可能である必要があります。

- **flags** – ログイン操作の成功または失敗時に何が起こるかを決定します。表2.1「JAAS モジュールを定義するためのフラグ」 では、有効な値について説明しています。

表2.1 JAAS モジュールを定義するためのフラグ

値	説明
required	このログインモジュールの認証は成功する必要があります。成功または失敗に関係なく、常にこのエントリーの次のログインモジュールに進みます。
requisite	このログインモジュールの認証は成功する必要があります。成功した場合は、次のログインモジュールに進みます。失敗した場合は、残りのログインモジュールを処理せずにすぐに戻ります。
sufficient	このログインモジュールの認証は成功する必要はありません。成功した場合は、残りのログインモジュールを処理せずにすぐに戻ります。失敗した場合は、次のログインモジュールに進みます。
任意	このログインモジュールの認証は成功する必要はありません。成功または失敗に関係なく、常にこのエントリーの次のログインモジュールに進みます。

jaas:module 要素の内容は、JAAS ログインモジュールインスタンスの初期化に使用されるプロパティー設定のスペース区切りリストです。特定のプロパティーは JAAS ログインモジュールによって決定され、適切な形式にする必要があります。

注記

レルムに複数のログインモジュールを定義できます。

標準 JAAS ログインプロパティーから XML への変換

Red Hat Fuse は、標準の Java ログイン設定ファイルと同じプロパティーを使用しますが、Red Hat Fuse では少し異なる方法で指定する必要があります。JAAS レルムの定義に対する Red Hat Fuse のアプローチと標準の Java ログイン設定ファイルアプローチを比較するには、例2.3「標準 JAAS プロパティー」 に示すログイン設定を変換する方法を考慮します。これは、Red Hat Fuse プロパティーログインモジュールクラス **PropertiesLoginModule** を使用して **PropertiesLogin** レルムを定義します。

例2.3 標準 JAAS プロパティー

```
PropertiesLogin {
    org.apache.activemq.jaas.PropertiesLoginModule required
        org.apache.activemq.jaas.properties.user="users.properties"
```

```
        org.apache.activemq.jaas.properties.group="groups.properties";
};
```

Blueprint ファイルの **jaas:config** 要素を使用した、同等の JAAS レルム定義を 例2.4「ブループリント JAAS プロパティー」 に示します。

例2.4 ブループリント JAAS プロパティー

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0">

  <jaas:config name="PropertiesLogin">
    <jaas:module flags="required"
      className="org.apache.activemq.jaas.PropertiesLoginModule">
        org.apache.activemq.jaas.properties.user=users.properties
        org.apache.activemq.jaas.properties.group=groups.properties
    </jaas:module>
  </jaas:config>

</blueprint>
```

重要

ブループリント設定の JAAS プロパティーに二重引用符を使用しないでください。

例

Red Hat Fuse は、JAAS 認証データを X.500 サーバーに保存するためのアダプターも提供します。例 2.5「JAAS レルムの設定」 は、**ldap://localhost:10389** にある LDAP サーバーに接続する Red Hat Fuse の **LDAPLoginModule** クラスを使用するよう、**LDAPLogin** レルムを定義します。

例2.5 JAAS レルムの設定

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0">

  <jaas:config name="LDAPLogin" rank="200">
    <jaas:module flags="required"
      className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule">
        initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
        connection.username=uid=admin,ou=system
        connection.password=secret
        connection.protocol=
        connection.url = ldap://localhost:10389
        user.base.dn = ou=users,ou=system
        user.filter = (uid=%u)
        user.search.subtree = true
        role.base.dn = ou=users,ou=system
```

```
        role.filter = (uid=%u)
        role.name.attribute = ou
        role.search.subtree = true
        authentication = simple
    </jaas:module>
  </jaas:config>
</blueprint>
```

LDAP ログインモジュールの詳細な説明と使用例については、「JAAS LDAP ログインモジュール」 を参照してください。

2.1.3. JAAS プロパティーログインモジュール

JAAS プロパティーログインモジュールは、ユーザーデータをフラットファイル形式で保存します (オプションで、保存されたパスワードはメッセージダイジェストアルゴリズムを使用して暗号化できます)。ユーザーデータは、単純なテキストエディターを使用して直接編集することも、**jaas:*** コンソールコマンドを使用して管理することもできます。

たとえば、Karaf コンテナーはデフォルトで JAAS プロパティーログインモジュールを使用し、関連するユーザーデータを **InstallDir/etc/users.properties** ファイルに保存します。

サポートされている認証情報

JAAS プロパティーログインモジュールはユーザー名/パスワードのクレデンシャルを認証し、認証されたユーザーに関連付けられたロールのリストを返します。

実装クラス

次のクラスは、JAAS プロパティーのログインモジュールを実装します。

org.apache.karaf.jaas.modules.properties.PropertiesLoginModule

JAAS ログインモジュールを実装します。

org.apache.karaf.jaas.modules.properties.PropertiesBackingEngineFactory

OSGi サービスとして公開する必要があります。このサービスは、Apache Karaf シェルから **jaas:*** コンソールコマンドを使用して、ユーザーデータを管理できるようにします (Apache Karaf コンソールリファレンス を参照)。

オプション

JAAS プロパティーログインモジュールは、次のオプションをサポートしています。

ユーザー

ユーザープロパティーファイルの場所。

ユーザープロパティーファイルの形式

ユーザープロパティーファイルは、プロパティーログインモジュールのユーザー名、パスワード、およびロールデータを保存するために使用されます。各ユーザーは、ユーザープロパティーファイルの 1 行で表されます。この行の形式は、次のとおりです。

```
Username=Password[,UserGroup|Role][,UserGroup|Role]...
```

このファイルでは、ユーザーグループを定義することもできます。各ユーザーグループは、次の形式の1行で表されます。

```
_g_\:GroupName=Role1[,Role2]...
```

たとえば、次のように、ユーザー **bigcheese** および **guest**、ならびにユーザーグループ **admingroup** および **guestgroup** を定義できます。

```
# Users
bigcheese=cheesepass,_g_:admingroup
guest=guestpass,_g_:guestgroup

# Groups
_g_\:admingroup=group,admin
_g_\:guestgroup=viewer
```

ブループリント設定のサンプル

以下の Blueprint 設定は、プロパティーログインモジュールを使用して新しい **karaf** レルムを定義する方法を示しています。ここで、**rank** 属性を **200** に設定すると、デフォルトの **karaf** レルムが上書きされます。

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.1.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0">

  <type-converters>
    <bean class="org.apache.karaf.jaas.modules.properties.PropertiesConverter"/>
  </type-converters>

<!--Allow usage of System properties, especially the karaf.base property-->
  <ext:property-placeholder
      placeholder-prefix="$[" placeholder-suffix="]"/>

  <jaas:config name="karaf" rank="200">
    <jaas:module flags="required"
className="org.apache.karaf.jaas.modules.properties.PropertiesLoginModule">
        users= $[karaf.base]/etc/users.properties
    </jaas:module>
  </jaas:config>

  <!-- The Backing Engine Factory Service for the PropertiesLoginModule -->
  <service interface="org.apache.karaf.jaas.modules.BackingEngineFactory">
    <bean class="org.apache.karaf.jaas.modules.properties.PropertiesBackingEngineFactory"/>
  </service>

</blueprint>
```

必ず、**BackingEngineFactory** Bean を OSGi サービスとしてエクスポートし、**jaas:*** コンソールコマンドがユーザーデータを管理できるようにします。

2.1.4. JAAS OSGi 設定ログインモジュール

概要

JAAS OSGi 設定ログインモジュールは、**OSGi Config Admin Service**を利用してユーザーデータを保管します。このログインモジュールは、JAAS プロパティーのログインモジュールとかなり似ていますが (たとえばユーザーエントリーの構文は同じ)、ユーザーデータを取得するメカニズムは OSGi Config Admin Service に基づいています。

ユーザーデータは、対応する OSGi 設定ファイル **etc/PersistentID.cfg** を作成するか、OSGi Config Admin Service によってサポートされる任意の設定方法を使用して直接編集できます。ただし、**jaas:*** コンソールのコマンドはサポートされません。

サポートされている認証情報

JAAS OAGi 設定ログインモジュールはユーザー名/パスワードのクレデンシャルを認証し、認証されたユーザーに関連付けられたロールのリストを返します。

実装クラス

次のクラスは、JAASOSGi 設定ログインモジュールを実装します。

org.apache.karaf.jaas.modules.osgi.OsgiConfigLoginModule

JAAS ログインモジュールを実装します。

注記

OSGi 設定ログインモジュールのバッキングエンジンファクトリーはありません。つまり、**jaas:*** コンソールコマンドを使用してこのモジュールを管理することはできません。

オプション

JAAS OSGi 設定ログインモジュールは、以下のオプションをサポートします。

pid

ユーザーデータを含む OSGi 設定の **永続 ID**。OSGi Config Admin 規格で、永続 ID は関連する設定プロパティーのセットを参照します。

設定ファイルの場所

設定ファイルの場所は、永続 ID **PersistentID** の設定が以下のファイルに保存される通常の慣例に従います。

```
InstallDir/etc/PersistentID.cfg
```

設定ファイルの形式

PersistentID.cfg 設定ファイルは、OSGi config ログインモジュールにユーザー名、パスワード、およびロールデータを保存するために使用されます。各ユーザーは、設定ファイル内の 1 行で表されます。この行の形式は、次のとおりです。

```
Username=Password[,Role][,Role]...
```

注記

JAAS OSGi 設定ログインモジュールでは、ユーザーグループはサポートされて **いません**。

ブループリント設定のサンプル

以下の Blueprint 設定は、OSGi 設定ログインモジュールを使用して新しい **karaf** レルムを定義する方法を示しています。ここで、**rank** 属性を **200** に設定すると、デフォルトの **karaf** レルムが上書きされます。

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.1.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0">

  <jaas:config name="karaf" rank="200">
    <jaas:module flags="required"
className="org.apache.karaf.jaas.modules.osgi.OsgiConfigLoginModule">
        pid = org.jboss.example.osgiconfigloginmodule
    </jaas:module>
  </jaas:config>

</blueprint>
```

この例では、ユーザーデータはファイル **InstallDir/etc/org.jboss.example.osgiconfigloginmodule.cfg** に保存され、**jaas:*** コンソールを使用して設定を編集することはできません。

2.1.5. JAAS 公開鍵ログインモジュール

JAAS 公開鍵ログインモジュールは、ユーザーデータをフラットファイル形式で保存します。このファイル形式は、単純なテキストエディターを使用して直接編集できます。ただし、**jaas:*** コンソールのコマンドはサポートされません。

たとえば、Karaf コンテナーはデフォルトで JAAS パブリックキーログインモジュールを使用し、関連するユーザーデータを **InstallDir/etc/keys.properties** ファイルに保存します。

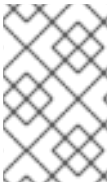
サポートされている認証情報

JAAS 公開鍵ログインモジュールは、SSH 鍵の認証情報を認証します。ユーザーがログインしようとすると、SSH プロトコルは保存されている公開鍵を使用してユーザーにチャレンジします。ユーザーは、チャレンジに答えるために、対応する秘密鍵を所有している必要があります。ログインが成功すると、ログインモジュールはユーザーに関連付けられているロールのリストを返します。

実装クラス

次のクラスは、JAAS 公開鍵ログインモジュールを実装します。

org.apache.karaf.jaas.modules.publickey.PublickeyLoginModule
: JAAS ログインモジュールを実装します。



注記

公開鍵ログインモジュールのバッキングエンジンファクトリーはありません。つまり、**jaas:*** コンソールコマンドを使用してこのモジュールを管理することはできません。

オプション

JAAS 公開鍵ログインモジュールは、次のオプションをサポートしています。

ユーザー
公開鍵ログインモジュールのユーザープロパティーファイルの場所。

鍵プロパティーファイルの形式

keys.properties ファイルは、公開鍵ログインモジュールのユーザー名、公開鍵、およびロールデータを保存するために使用されます。各ユーザーは、鍵プロパティーファイルの 1 行で表されます。この行の形式は、次のとおりです。

```
Username=PublicKey[,UserGroup|Role][,UserGroup|Role]...
```

ここで **PublicKey** は、SSH キーペアの公開鍵の部分です (通常は UNIX システムの ~/**.ssh/id_rsa.pub** にあるユーザーのホームディレクトリーにあります)。

たとえば、**admin** ロールでユーザー **jdoe** を作成するには、以下のようなエントリーを作成します。

```
jdoe=AAAAB3NzaC1kc3MAAACBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2USZpRV1AIlH
7WT2NWPq/xfW6MPbLm1Vs14E7gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1fnfqimFQ8E+4
P208UewwI1VBNaFpEy9nXzrith1yrv8iIDGZ3RSAHHAAAAFQCXYFCPFSMLzLKSuYKi64QL8Fgc9QA
AAnEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCBgLRJFnEj6E
woFhO3zwkyjMim4TwWeotifI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImog9/hWuWfB
pKLZl6Ae1UlZAFMO/7PSSoAAACBAKKSU2PFl/qOLxIwmBZPPIcJshVe7bVUpFvyl3BbJDow8rXfskl8w
O63OzP/qLmcJM0+JbcRU/53Jj7uyk31drV2qxhIOsLDC9dGCWj47Y7TyhPdXh/0dthTRBy6bqGtRPxGa
7gJov1xm/UuYYXPIUR/3x9MAZvZ5xvE0kYXO+rx,admin
```

重要

ここで、**id_rsa.pub** ファイルの内容をすべて挿入しないでください。公開鍵自体を表す記号のブロックだけを挿入します。

このファイルでは、ユーザーグループを定義することもできます。各ユーザーグループは、次の形式の 1 行で表されます。

```
_g_\:GroupName=Role1[,Role2]...
```

ブループリント設定のサンプル

以下の Blueprint 設定は、パブリックキーログインモジュールを使用して新しい **karaf** レルムを定義する方法を示しています。ここで、**rank** 属性を **200** に設定すると、デフォルトの **karaf** レルムが上書きされます。

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

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.1.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0">

<!--Allow usage of System properties, especially the karaf.base property-->
  <ext:property-placeholder
      placeholder-prefix="$[" placeholder-suffix="]"/>

  <jaas:config name="karaf" rank="200">
    <jaas:module flags="required"
classNam="org.apache.karaf.jaas.modules.publickey.PublickeyLoginModule">
        users = $[karaf.base]/etc/keys.properties
    </jaas:module>
  </jaas:config>

</blueprint>
```

この例では、ユーザーデータはファイル **InstallDir/etc/keys.properties** に保存され、**jaas:*** コンソールを使用して設定を編集することはできません。

2.1.6. JAAS JDBC ログインモジュール

概要

JAAS JDBC ログインモジュールを使用すると、Java Database Connectivity (JDBC) を使用してデータベースに接続し、ユーザーデータをデータベースバックエンドに格納できます。したがって、JDBC をサポートする任意のデータベースを使用してユーザーデータを格納できます。ユーザーデータを管理するには、ネイティブデータベースクライアントツールまたは **jaas:*** コンソールコマンドのいずれかを使用できます (バッキングエンジンは設定済みの SQL クエリーを使用して関連データベースの更新を実行します) 。

複数のログインモジュールを各ログインモジュールと組み合わせて、認証コンポーネントと認可コンポーネントの両方を提供できます。たとえば、デフォルトの **PropertiesLoginModule** と **JDBCLoginModule** を組み合わせてシステムにアクセスできるようにすることができます。

注記

JAAS JDBC ログインモジュールでは、ユーザーグループはサポートされて **いません**。

サポートされている認証情報

JAAS JDBC ログインモジュールは、ユーザー名/パスワードのクレデンシャルを認証し、認証されたユーザーに関連付けられたロールのリストを返します。

実装クラス

次のクラスは、JAASJDBC ログインモジュールを実装します。

org.apache.karaf.jaas.modules.jdbc.JDBCLoginModule

JAAS ログインモジュールを実装します。

org.apache.karaf.jaas.modules.jdbc.JDBCBackingEngineFactory

OSGi サービスとして公開する必要があります。このサービスは、Apache Karaf シェルから **jaas:*** コンソールコマンドを使用してユーザーデータを管理できるようにします (olink:FMQCommandRef/Consolejaas を参照)。

オプション

JAAS JDBC ログインモジュールは、次のオプションをサポートしています。

datasource

OSGi サービスまたは JNDI 名として指定された JDBC データソース。次の構文を使用して、データソースの OSGi サービスを指定できます。

```
osgi:ServiceInterfaceName[/ServicePropertiesFilter]
```

ServiceInterfaceName は、データソースの OSGi サービス (通常は **javax.sql.DataSource**) によってエクスポートされるインターフェイスまたはクラスです。

複数のデータソースを Karaf コンテナー内の OSGi サービスとしてエクスポートできるため、通常、必要な特定のデータソースを選択するためにフィルター **ServicePropertiesFilter** を指定する必要があります。OSGi サービスのフィルターは、サービスプロパティー設定に適用され、LDAP フィルター構文から借用した構文に従います。

query.password

ユーザーのパスワードを取得する SQL クエリー。クエリーには 1 つの疑問符 **?** を含めることができます。これは、実行時にユーザー名に置き換えられます。

query.role

ユーザーのロールを取得する SQL クエリー。クエリーには 1 つの疑問符 **?** を含めることができます。これは、実行時にユーザー名に置き換えられます。

insert.user

新しいユーザーエントリーを作成する SQL クエリー。クエリーには 2 つの疑問符 **?** を含めることができます。最初の疑問符はユーザー名に置き換えられ、2 つ目の疑問符は実行時にパスワードに置き換えられます。

insert.role

ユーザーエントリーにロールを追加する SQL クエリー。クエリーには 2 つの疑問符 **?** を含めることができます。最初の疑問符はユーザー名に置き換えられ、2 つ目の疑問符は実行時にロールに置き換えられます。

delete.user

ユーザーエントリーを削除する SQL クエリー。クエリーには 1 つの疑問符 **?** を含めることができます。これは、実行時にユーザー名に置き換えられます。

delete.role

ユーザーエントリーからロールを削除する SQL クエリー。クエリーには 2 つの疑問符 **?** を含めることができます。最初の疑問符はユーザー名に置き換えられ、2 つ目の疑問符は実行時にロールに置き換えられます。

delete.roles

ユーザーエントリーから複数のロールを削除する SQL クエリー。クエリーには 1 つの疑問符 **?** を含めることができます。これは、実行時にユーザー名に置き換えられます。

JDBC ログインモジュールの設定例

JDBC ログインモジュールを設定するには、次の主な手順を実行します。

1. 「データベーステーブルの作成」
2. 「データソースの作成」
3. 「データソースを OSGi サービスとして指定」

データベーステーブルの作成

JDBC ログインモジュールを設定する前に、ユーザーデータを格納するためにバッキングデータベースにユーザー users テーブルと roles テーブルを設定する必要があります。たとえば、以下の SQL コマンドは、適切な **users** テーブルと **roles** テーブルの作成方法を示しています。

```
CREATE TABLE users (
  username VARCHAR(255) NOT NULL,
  password VARCHAR(255) NOT NULL,
  PRIMARY KEY (username)
);
CREATE TABLE roles (
  username VARCHAR(255) NOT NULL,
  role VARCHAR(255) NOT NULL,
  PRIMARY KEY (username,role)
);
```

users テーブルにはユーザー名/パスワードデータが格納され、**roles** テーブルはユーザー名を 1 つ以上のロールに関連付けます。

データソースの作成

JDBC ログインモジュールで JDBC データソースを使用するには、データソースインスタンスを作成し、データソースを OSGi サービスとしてエクスポートするのが正しい方法です。そうすることで、JDBC ログインモジュールは、エクスポートされた OSGi サービスを参照することにより、データソースにアクセスできるようになります。たとえば、Blueprint ファイルに以下のようなコードを使用して、MySQL データソースインスタンスを作成し、OSGi サービス (**javax.sql.DataSource** 型) として公開できます。

```
<blueprint xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
  <bean id="mysqlDatasource"
        class="com.mysql.jdbc.jdbc2.optional.MysqlDataSource">
    <property name="serverName" value="localhost"></property>
    <property name="databaseName" value="DBName"></property>
    <property name="port" value="3306"></property>
    <property name="user" value="DBUser"></property>
    <property name="password" value="DBPassword"></property>
  </bean>

  <service id="mysqlDS" interface="javax.sql.DataSource"
        ref="mysqlDatasource">
    <service-properties>
      <entry key="osgi.jndi.service.name" value="jdbc/karafdb"/>
    </service-properties>
  </service>
</blueprint>
```

上記のフルーフリント設定は、OSGi バンドルとして Karaf コンテナーにパッケージ化およびインストールする必要があります。

データソースを OSGi サービスとして指定

データソースがインスタンス化され、OSGi サービスとしてエクスポートされると、JDBC ログインモジュールを設定する準備が整います。特に、JDBC ログインモジュールの **datasource** オプションは、以下の構文を使用してデータソースの OSGi サービスを参照できます。

```
osgi:javax.sql.DataSource/(osgi.jndi.service.name=jdbc/karafdb)
```

ここで、**javax.sql.DataSource** はエクスポートされた OSGi サービスのインターフェイスタイプで、フィルター **(osgi.jndi.service.name=jdbc/karafdb)** は、**osgi.jndi.service.name** サービスプロパティーに値 **jdbc/karafdb** を持つ特定の **javax.sql.DataSource** インスタンスを選択します。

たとえば、以下の Blueprint 設定を使用して、サンプル MySQL データソースを参照する JDBC ログインモジュールで **karaf** レルムをオーバーライドできます。

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.1.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0">

<!--Allow usage of System properties, especially the karaf.base property-->
  <ext:property-placeholder
      placeholder-prefix="$[" placeholder-suffix="]"/>

  <jaas:config name="karaf" rank="200">
    <jaas:module flags="required"
     className="org.apache.karaf.jaas.modules.jdbc.JDBCLoginModule">
       datasource = osgi:javax.sql.DataSource/(osgi.jndi.service.name=jdbc/karafdb)
       query.password = SELECT password FROM users WHERE username=?
       query.role = SELECT role FROM roles WHERE username=?
       insert.user = INSERT INTO users VALUES(?,?)
       insert.role = INSERT INTO roles VALUES(?,?)
       delete.user = DELETE FROM users WHERE username=?
       delete.role = DELETE FROM roles WHERE username=? AND role=?
       delete.roles = DELETE FROM roles WHERE username=?
    </jaas:module>
  </jaas:config>

  <!-- The Backing Engine Factory Service for the JDBCLoginModule -->
  <service interface="org.apache.karaf.jaas.modules.BackingEngineFactory">
    <bean class="org.apache.karaf.jaas.modules.jdbc.JDBCBackingEngineFactory"/>
  </service>

</blueprint>
```

注記

上記の設定に示されている SQL ステートメントは、実際にはこれらのオプションのデフォルト値です。したがって、これらの SQL ステートメントと一致するユーザーテーブルとロールテーブルを作成する場合は、オプション設定を省略して、デフォルトに依存することができます。

JDBCLoginModule を作成する他に、前述の Blueprint 設定も **JDBCBackingEngineFactory** インスタンスをインスタンス化し、エクスポートします。これにより、**jaas:*** コンソールコマンドを使用してユーザーデータを管理できます。

2.1.7. JAAS LDAP ログインモジュール

概要

JAAS LDAP ログインモジュールを使用すると、ユーザーデータを LDAP データベースに保存できます。保存されたユーザーデータを管理するには、標準の LDAP クライアントツールを使用します。ただし、**jaas:*** コンソールのコマンドはサポート**されません**。

Red Hat Fuse での LDAP の使用に関する詳細は、LDAP 認証チュートリアル を参照してください。

注記

ユーザーグループは、JAAS LDAP ログインモジュールではサポートされて **いません**。

サポートされている認証情報

JAAS LDAP 設定ログインモジュールはユーザー名/パスワードのクレデンシャルを認証し、認証されたユーザーに関連付けられたロールのリストを返します。

実装クラス

次のクラスは、JAAS LDAP ログインモジュールを実装します。

org.apache.karaf.jaas.modules.ldap.LDAPLoginModule

JAAS ログインモジュールを実装します。Karaf コンテナーにプリロードされているため、バンドルをインストールする必要はありません。

注記

LDAP ログインモジュールのバッキングエンジンファクトリーはありません。つまり、**jaas:*** コンソールコマンドを使用してこのモジュールを管理することはできません。

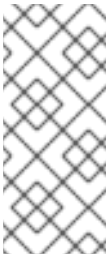
オプション

JAAS LDAP ログインモジュールは、次のオプションをサポートしています。

認証

LDAP サーバーにバインドする際に使用する認証方法を指定します。有効な値は以下のとおりです。

- **simple** – ユーザー名とパスワード認証でバインドします。**connection.username** および **connection.password** プロパティーを設定する必要があります。
- **none** – 匿名でバインドします。この場合、**connection.username** プロパティーおよび **connection.password** プロパティーは割り当て解除できます。



注記

ディレクトリーサーバーへの接続は、検索を実行するためにのみ使用されます。この場合、認証されたバインドよりも高速であるため、匿名バインドがよく使用されます (ただし、ファイアウォールの背後に配置するなどして、ディレクトリーサーバーが十分に保護されていることも確認する必要があります)。

connection.url

LDAP URL (**ldap://Host:Port**) を使用してディレクトリーサーバーの場所を指定します。オプションでこの URL を修飾するには、スラッシュ / とその後にディレクトリーツリーの特定ノードの DN を追加します。この接続で SSL セキュリティーを有効にするには、URL で **ldaps:** スキームを指定する必要があります (例: **ldaps://Host:Port**)。スペース区切りリストで複数の URL を指定することもできます。次に例を示します。

```
connection.url=ldap://10.0.0.153:2389 ldap://10.10.178.20:389
```

connection.username

ディレクトリーサーバーへの接続を開くユーザーの DN を指定します。たとえば、**uid=admin,ou=system** です。DN に空白が含まれる場合、**LDAPLoginModule** は解析できません。唯一の解決策は、空白を含む DN 名の前後に二重引用符を追加してから、バックスラッシュを追加して引用符をエスケープすることです。たとえば、**uid=admin,ou=\"system index\"** になります。

connection.password

connection.username からの DN と一致するパスワードを指定します。ディレクトリーサーバーでは通常、パスワードは対応するディレクトリーエントリーの **userPassword** 属性として保存されます。

context.com.sun.jndi.ldap.connect.pool

true の場合、LDAP 接続の接続プールを有効にします。デフォルトは **false** です。

context.com.sun.jndi.ldap.connect.timeout

LDAP サーバーへの TCP 接続を作成するためのタイムアウトをミリ秒単位で指定します。デフォルト値は無限になっているため、接続試行がハングアップする可能性があるため、このプロパティーを明示的に設定することを推奨します。

context.com.sun.jndi.ldap.read.timeout

LDAP 操作の読み取りタイムアウトをミリ秒単位で指定します。デフォルト値は無限になっているため、このプロパティーを明示的に設定することを推奨します。

context.java.naming.referral

LDAP 参照 は、一部の LDAP サーバーでサポートされている間接参照の形式です。LDAP 参照は、1 つ以上の URL を含む LDAP サーバーのエントリーです (通常、別の LDAP サーバーの 1 つ以上のノードを参照します)。**context.java.naming.referral** プロパティーを使用すると、フォローする参照を有効または無効にすることができます。次のいずれかの値に設定できます。

- **follow** は、参照をフォローします (LDAP サーバーによってサポートされることを前提とします)。
- **ignore** は、すべての参照を通知せずに無視します。
- **throw** は、リファールに遭遇するたびに **PartialResultException** を出力します。

disableCache

このプロパティーを **true** に設定すると、ユーザーおよびロールキャッシュを無効にできます。デフォルトは **false** です。

initial.context.factory

LDAP サーバーへの接続に使用されるコンテキストファクトリーのクラスを指定します。これは常に **com.sun.jndi.ldap.LdapCtxFactory** に設定する必要があります。

role.base.dn

ロールエントリーを検索する DIT のサブツリーの DN を指定します。たとえば、**ou=groups,ou=system** となります。

role.filter

ロールの検索に使用される LDAP 検索フィルターを指定します。これは、**role.base.dn** によって選択されるサブツリーに適用されます。たとえば、**(member=uid=%u)** となります。LDAP 検索操作に渡される前に、値は次のように文字列置換を受けます。

- **%u** は、受信クレデンシャルから抽出されたユーザー名に置き換えられます。
- **%dn** は、LDAP サーバーの対応するユーザーの RDN に置き換えられます (**user.filter** フィルターとの照合によって検出されます)。
- **%fqdn** は、LDAP サーバーの対応するユーザーの DN に置き換えられます (**user.filter** フィルターとの照合によって検出されます)。

role.mapping

LDAP グループと JAAS ロールの間のマッピングを指定します。マッピングが指定されていない場合、デフォルトで各 LDAP グループは同じ名前の対応する JAAS ロールにマッピングされます。ロールマッピングは、次の構文で指定されます。

```
ldap-group=jaas-role(,jaas-role)*(;ldap-group=jaas-role(,jaas-role)*)*
```

各 LDAP グループ **ldap-group** は Common Name (CN) によって指定されます。

たとえば、LDAP グループ **admin**、**devop**、および **tester** の場合は、以下のように JAAS ロールにマップできます。

```
role.mapping=admin=admin;devop=admin,manager;tester=viewer
```

role.name.attribute

ロール/グループの名前を含むロールエントリーの属性タイプを指定します。このオプションを省略すると、ロール検索機能は事実上無効になります。(例: **cn**)。

role.search.subtree

ロールエントリー検索範囲に、**role.base.dn** によって選択されたツリーのサブツリーが含まれるかどうかを指定します。**true** の場合、ロールルックアップは再帰的 (**SUBTREE**) になります。**false** の場合、ロールルックアップは最初のレベルでのみ実行されます (**ONELEVEL**)。

ssl

LDAP サーバーへの接続が SSL を使用して保護されているかどうかを指定します。**connection.url** が SSL **ldaps://** で始まる場合は、このプロパティーに関係なく使用されます。

ssl.provider

LDAP 接続に使用する SSL プロバイダーを指定します。指定しない場合、デフォルトの SSL プロバイダーが使用されます。

ssl.protocol

SSL 接続に使用するプロトコルを指定します。SSLv3 プロトコルが使用されないようにするには (POODLE 脆弱性)、このプロパティーを **TLSv1** に設定する **必要があります**。

ssl.algorithm

トラストストアマネージャーが使用するアルゴリズムを指定します。たとえば、**PKIX** です。

ssl.keystore

LDAP クライアント自身の X.509 証明書を格納するキーストアの ID (LDAP サーバーで SSL クライアント認証が有効になっている場合にのみ必要)。キーストアは、**jaas:keystore** 要素を使用してデプロイする必要があります (「Apache DS のサンプル設定」 を参照)。

ssl.keyalias

LDAP クライアント独自の X.509 証明書のキーストアエイリアス (**ssl.keystore** によって指定されたキーストアに複数の証明書が保存される場合にのみ必要)。

ssl.truststore

LDAP サーバーの証明書を検証するために使用される信頼できる CA 証明書を格納するキーストアの ID (LDAP サーバーの証明書チェーンは、トラストストア内の証明書の 1 つによって署名されている必要があります)。キーストアは、**jaas:keystore** 要素を使用してデプロイする必要があります。

user.base.dn

ユーザーエントリーを検索するための DIT のサブツリーの DN を指定します。たとえば、**ou=users,ou=system** です。

user.filter

ユーザーの認証情報の検索に使用される LDAP 検索フィルターを指定します。これは、**user.base.dn** によって選択されるサブツリーに適用されます。たとえば、**(uid=%u)** です。LDAP 検索操作に渡される前に、値は次のように文字列置換を受けます。

- **%u** は、受信クレデンシャルから抽出されたユーザー名に置き換えられます。

user.search.subtree

ユーザーエントリー検索範囲に、**user.base.dn** によって選択されたツリーのサブツリーが含まれるかどうかを指定します。**true** の場合、ユーザールックアップは再帰的 (**SUBTREE**) になります。**false** の場合、ユーザールックアップは最初のレベル (**ONELEVEL**) でのみ実行されます。

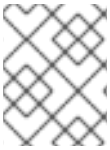
Apache DS のサンプル設定

以下の Blueprint 設定は、LDAP ログインモジュールを使用して新しい **karaf** レルムを定義する方法を示しています。ここで、**rank** 属性を **200** に設定すると、デフォルトの **karaf** レルムが上書きされ、LDAP ログインモジュールは Apache ディレクトリーサーバーに接続されます。

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.1.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0">

  <jaas:config name="karaf" rank="100">

    <jaas:module className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
flags="sufficient">
      debug=true

      <!-- LDAP Configuration -->
      initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
<!--  multiple LDAP servers can be specified as a space separated list of URLs -->
```

```
      connection.url=ldap://10.0.0.153:2389 ldap://10.10.178.20:389

<!--  authentication=none -->
      authentication=simple
      connection.username=cn=Directory Manager
      connection.password=directory

      <!-- User Info -->
      user.base.dn=dc=redhat,dc=com
      user.filter=(&amp;(objectClass=InetOrgPerson)(uid=%u))
      user.search.subtree=true

      <!-- Role/Group Info-->
      role.base.dn=dc=redhat,dc=com
      role.name.attribute=cn
<!--
      The 'dc=redhat,dc=com' used in the role.filter
      below is the user.base.dn.
-->
<!--      role.filter=(uniquemember=%dn,dc=redhat,dc=com) -->
      role.filter=(&amp;(objectClass=GroupOfUniqueNames)(UniqueMember=%fqdn))
      role.search.subtree=true

<!-- role mappings - a ';' separated list -->
      role.mapping=JBossAdmin=admin;JBossMonitor=viewer

<!-- LDAP context properties -->
      context.com.sun.jndi.ldap.connect.timeout=5000
      context.com.sun.jndi.ldap.read.timeout=5000

<!-- LDAP connection pooling -->
<!-- http://docs.oracle.com/javase/jndi/tutorial/ldap/connect/pool.html -->
<!-- http://docs.oracle.com/javase/jndi/tutorial/ldap/connect/config.html -->
     context.com.sun.jndi.ldap.connect.pool=true

<!-- How are LDAP referrals handled?

     Can be `follow`, `ignore` or `throw`.  Configuring `follow` may not work on all LDAP servers,
`ignore` will
     silently ignore all referrals, while `throw` will throw a partial results exception if there is a referral.
-->
     context.java.naming.referral=ignore

<!-- SSL configuration -->
     ssl=false
     ssl.protocol=SSL
<!-- matches the keystore/truststore configured below -->
     ssl.truststore=ks
     ssl.algorithm=PKIX
<!-- The User and Role caches can be disabled - 6.3.0 179 and later -->
     disableCache=true
    </jaas:module>
  </jaas:config>

  <!-- Location of the SSL truststore/keystore
  <jaas:keystore name="ks" path="file:///${karaf.home}/etc/ldap.truststore"
```

```
keystorePassword="XXXXXX" />
-->
</blueprint>
```



注記

SSL を有効にするには、**connection.url** 設定で **ldaps** スキームを使用する必要があります。

重要

Poodle 脆弱性 (CVE-2014-3566) から保護するには、**ssl.protocol** を **TLSv1**(以降) に設定する必要があります。

さまざまなディレクトリーサーバーのフィルター設定

ディレクトリーサーバー間の最も重要な違いは、LDAP ログインモジュールでのフィルターオプションの設定に関連しています。正確な設定は、最終的には DIT の設定によって異なりますが、次の表は、さまざまなディレクトリーサーバーに必要な一般的なロールフィルター設定を示しています。

Directory Server	一般的なフィルター設定
389-DS Red Hat DS	user.filter=(& (objectClass=InetOrgPerson)(uid=%u)) role.filter=(uniquemember=%fqdn)
MS Active Directory	user.filter=(&(objectCategory=person) (samAccountName=%u)) role.filter=(uniquemember=%fqdn)
Apache DS	user.filter=(uid=%u) role.filter=(member=uid=%u)
OpenLDAP	user.filter=(uid=%u) role.filter=(member:=uid=%u)

注記

上記の表では、オプション設定が Blueprint XML ファイルに組み込まれるため、**&** 記号 (論理 **And** 演算子を表す) は **&** としてエスケープ処理されます。

2.1.8. JAAS ログ監査ログインモジュール

ログインモジュール **org.apache.karaf.jaas.modules.audit.LogAuditLoginModule** は、認証試行の堅牢なロギングを提供します。最大ファイルサイズの設定、ログローテーション、ファイル圧縮、フィル

タリングなどの標準的なログ管理機能をサポートしています。これらのオプションの設定は、ロギング設定ファイルで確立します。

デフォルトでは、認証監査ロギングは無効になっています。ロギングを有効にするには、ロギング設定と監査設定を定義してから、2 つをリンクする必要があります。ロギング設定では、**ファイルアペンダー** プロセスと **ロガー** プロセスのプロパティーを指定します。ファイルアペンダーは、認証イベントに関する情報を指定されたファイルに公開します。ロガーは、認証イベントに関する情報を取得し、指定したアペンダーがそれを利用できるようにするメカニズムです。標準の Karaf Log4j ロギング設定ファイル **etc/org.ops4j.pax.logging.cfg** でロギング設定を定義します。

監査設定により、監査ロギングとロギングフラストラクチャーへのリンクが使用可能になります。監査設定は、**etc/org.apache.karaf.jaas.cfg** ファイルに定義します。

アペンダー設定

デフォルトでは、標準の Karaf Log4j 設定ファイル (**etc/org.ops4j.pax.logging.cfg**) は、**AuditRollingFile** という名前の監査ロギングアペンダーを定義します。

次のサンプル設定ファイルの抜粋は、**${karaf.data}/security/audit.log** で監査ログファイルに書き込むアペンダーのプロパティーを示しています。

```
# Audit file appender
log4j2.appender.audit.type = RollingRandomAccessFile
log4j2.appender.audit.name = AuditRollingFile
log4j2.appender.audit.fileName = ${karaf.data}/security/audit.log
log4j2.appender.audit.filePattern = ${karaf.data}/security/audit.log.%i
log4j2.appender.audit.append = true
log4j2.appender.audit.layout.type = PatternLayout
log4j2.appender.audit.layout.pattern = ${log4j2.pattern}
log4j2.appender.audit.policies.type = Policies
log4j2.appender.audit.policies.size.type = SizeBasedTriggeringPolicy
log4j2.appender.audit.policies.size.size = 8MB
```

アペンダーを使用するには、アペンダーがログファイルに公開するための情報を提供するロガーを設定する必要があります。

ロガー設定

デフォルトでは、Karaf Log4j 設定ファイル (**etc/org.ops4j.pax.logging.cfg**) は、**org.apache.karaf.jaas.modules.audit** という名前の監査ロガーを定義します。次のサンプル設定ファイルの抜粋では、認証イベントに関する情報を、**AuditRollingFile** という名前のアペンダーに提供するようにデフォルトロガーが設定されています。

```
log4j2.logger.audit.name = org.apache.karaf.jaas.modules.audit
log4j2.logger.audit.level = INFO
log4j2.logger.audit.additivity = false
log4j2.logger.audit.appenderRef.AuditRollingFile.ref = AuditRollingFile
```

log4j2.logger.audit.appenderRef.AuditRollingFile.ref の値は、**etc/org.ops4j.pax.logging.cfg** の **Audit file appender** セクションの **log4j2.appender.audit.name** の値と一致する必要があります。

2.1.8.1. 認証監査ロギングの有効化

ロギング設定を確立した後、監査ロギングをオンにして、ロギング設定を監査設定に接続できます。

監査ロギングを有効にするには、以下の行を **etc/org.apache.karaf.jaas.cfg** に挿入します。

```
audit.log.enabled = true
audit.log.logger = <logger.name>
audit.log.level = <level>
```

<logger.name> は、**org.jboss.fuse.audit** や **com.example.audit** など、Apache Log4J ライブラリーと Log4J2 ライブラリーによって確立される標準のロガー (カテゴリー) 名のドット区切り形式を表します。**<level>`** は、**WARN**、**INFO**、**TRACE**、**DEBUG** などのログレベル設定を表します。

たとえば、以下のサンプル監査設定ファイルにある以下の抜粋で、監査ログが有効になり、**org.apache.karaf.jaas.modules.audit** という名前の監査ロガーを使用するように設定されます。

```
audit.log.enabled = true
audit.log.logger = org.apache.karaf.jaas.modules.audit
audit.log.level = INFO
```

audit.log.logger の値は、Karaf Log4j 設定ファイル (**etc/org.ops4j.pax.logging.cfg**) の **log4j2.logger.audit.name** の値と一致する必要があります。

ファイルを更新すると、Apache Felix ファイルインストールバンドルが変更を検出し、Apache Felix 設定管理サービス (**Config Admin**) の設定を更新します。次に、Config Admin からの設定がロギングインフラストラクチャーに渡されます。

設定ファイルを更新するための Apache Karaf シェルコマンド

<FUSE_HOME>/etc の設定ファイルは直接編集するか、Apache Karaf **config:*** コマンドを実行して、Config Admin を更新できます。

config* コマンドを使用して設定を更新すると、Apache Felix File Install バンドルは変更について通知され、関連する **etc/*.cfg** ファイルが自動的に更新されます。

例: **config** コマンドを使用した JAAS レルムのプロパティーのリスト表示

JAAS レルムのプロパティーをリスト表示するには、シェルプロンプトから次のコマンドを入力します。

config:property-list --pid org.apache.karaf.jaas

このコマンドは、次の例のとおり、レルムの現在のプロパティーを返します。

```
audit.log.enabled = true
audit.log.level = INFO
audit.log.logger = org.apache.karaf.jaas.modules.audit
encryption.algorithm = MD5
encryption.enabled = false
encryption.encoding = hexadecimal
encryption.name =
encryption.prefix = {CRYPT}
encryption.suffix = {CRYPT}
```

例: **config** コマンドを使用した監査ログレベルの変更

レルムの監査ログレベルを **DEBUG** に変更するには、シェルプロンプトで、**config:property-set --pid org.apache.karaf.jaas audit.log.level DEBUG**を入力します。

変更が有効であることを確認するには、再度プロパティーをリストして、**audit.log.level** の値を確認します。

2.1.9. JAAS ファイル監査ログインモジュール

認証モジュール **org.apache.karaf.jaas.modules.audit.FileAuditLoginModule** は、認証試行の堅牢なロギングを提供します。ファイル監査ログインモジュールは、指定されたファイルに直接書き込みます。Pax ロギングインフラストラクチャーに依存しないため、設定は簡単です。ただし、ログ監査ログインモジュール とは異なり、パターンフィルタリングやログファイルローテーションなどのログ管理機能はサポートされていません。

FileAuditLoginModule で監査ロギングを有効にするには、以下の行を **etc/org.apache.karaf.jaas.cfg** に挿入します。

```
audit.file.enabled = true
audit.file.file = ${karaf.data}/security/audit.log
```

注記

通常、**ファイル監査ログインモジュール** と **ログ監査ログインモジュール** の両方を使用して監査ログを設定することはありません。両方のモジュールでログを有効にする場合、一意のターゲットログファイルを使用するように各モジュールを設定することで、データの損失を回避できます。

2.1.10. 保存されたパスワードの暗号化

デフォルトでは、JAAS ログインモジュールはパスワードをプレインテキスト形式で保存します。ファイルのアクセス許可を適切に設定することでこのようなデータを保護でき、そのように保護する必要もありますが、パスワードを (**メッセージダイジェスト** アルゴリズムを使用して) 非表示形式で保存することで、パスワードをさらに保護できます。

Red Hat Fuse は、パスワード暗号化を有効にするための一連のオプションを提供します。これは、**任意** の JAAS ログインモジュール (不要な公開鍵ログインモジュールを除く) と組み合わせることができます。

重要

メッセージダイジェストアルゴリズムを解読するのは困難ですが、攻撃に対して無防備ではありません (たとえば、暗号化ハッシュ関数に関する Wikipedia の記事 を参照してください)。パスワードを含むファイルを保護するために、パスワードの暗号化の使用に加え、常にファイル権限を使用してください。

オプション

オプションで、次のログインモジュールのプロパティーを設定することにより、JAAS ログインモジュールのパスワード暗号化を有効にできます。これには、「Jasypt 暗号化を使用したログインモジュールの例」 の説明に従って **InstallDir/etc/org.apache.karaf.jaas.cfg** ファイルを編集するか、独自の Blueprint ファイルをデプロイします。

encryption.enabled

パスワード暗号化を有効にするには、**true** に設定します。

encryption.name

OSGi サービスとして登録されている暗号化サービスの名前。

encryption.prefix

暗号化されたパスワードの接頭辞。

encryption.suffix

暗号化されたパスワードの接尾辞。

encryption.algorithm

暗号化アルゴリズムの名前を指定します (例: **MD5** または **SHA-1**)。次の暗号化アルゴリズムのいずれかを指定できます。

- **MD2**
- **MD5**
- **SHA-1**
- **SHA-256**
- **SHA-384**
- **SHA-512**

encryption.encoding

暗号化されたパスワードのエンコード: **hexadecimal** または **base64**

encryption.providerName (Jasypt のみ)

ダイジェストアルゴリズムを提供する **java.security.Provider** インスタンスの名前。

encryption.providerClassName (Jasypt のみ)

ダイジェストアルゴリズムを提供するセキュリティープロバイダーのクラス名

encryption.iterations (Jasypt のみ)

ハッシュ関数を再帰的に適用する回数。

encryption.saltSizeBytes (Jasypt のみ)

ダイジェストの計算に使用されるソルトのサイズ。

encryption.saltGeneratorClassName (Jasypt のみ)

ソルトジェネレーターのクラス名。

role.policy

ロールプリンシパルを識別するためのポリシーを指定します。値 **prefix** または **group** を指定できます。

role.discriminator

ロールポリシーで使用される識別子の値を指定します。

暗号化サービス

Fuse が提供する暗号化サービスは 2 つあります。

- **encryption.name = basic**(「Basic 暗号化サービス」 で説明されている)
- **encryption.name = jasypt**(「Jasypt 暗号化」 で説明されている)

独自の暗号化サービスを作成することもできます。これを実行するには、以下を行います。

- **org.apache.karaf.jaas.modules.EncryptionService** インターフェイスを実装

- 実装を OSGI サービスとして公開します。

次のリストは、カスタム暗号化サービスを OSGI コンテナーに公開する方法を示しています。

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

  <service interface="org.apache.karaf.jaas.modules.EncryptionService">
    <service-properties>
      <entry key="name" value="jasypt" />
    </service-properties>
    <bean class="org.apache.karaf.jaas.jasypt.impl.JasyptEncryptionService"/>
  </service>
  ...
</blueprint>
```

Basic 暗号化サービス

Basic 暗号化サービスは、デフォルトで Karaf コンテナーにインストールされ、**encryption.name** プロパティーを **basic** という値に設定することで参照が可能です。Basic 暗号化サービスでは、メッセージダイジェストアルゴリズムは SUN セキュリティープロバイダー (Oracle JDK のデフォルトのセキュリティープロバイダー) によって提供されます。

Jasypt 暗号化

Jasypt 暗号化サービスは通常、デフォルトで Karaf にインストールされます。必要に応じて、以下のように **jasypt-encryption** 機能をインストールして明示的にインストールできます。

```
JBossA-MQ:karaf@root> features:install jasypt-encryption
```

このコマンドは、必要な Jasypt バンドルをインストールし、Jasypt 暗号化を OSGi サービスとしてエクスポートして、JAAS ログインモジュールで使用できるようにします。

Jasypt 暗号化の詳細については、Jasypt のドキュメント を参照してください。

Jasypt 暗号化を使用したログインモジュールの例

デフォルトでは、パスワードは **etc/users.properties** ファイルにクリアテキストで保存されます。**jasypt-encryption** 機能をインストールし、**etc/org.apache.karaf.jaas.cfg** 設定ファイルを変更することで、暗号化を有効にできます。

1. 機能 **jasypt-encryption** をインストールします。これにより、**jasypt** サービスがインストールされます。

   ```
   karaf@root> features:install jasypt-encryption
   ```

   これで、**jaas** コマンドを使用してユーザーを作成できます。

2. **$FUSE_HOME/etc/org.apache.karaf.jaas.cfg** ファイルを開き、次のように変更します。**encryption.enabled = true**、**encryption.name = jasypt**、およびこの場合は **encryption.algorithm = SHA-256** を設定します。その他の **encryption.algorithm** オプションは、要件に応じて設定できます。

   ```
   #
   # Boolean enabling / disabling encrypted passwords
   ```

```
#
encryption.enabled = true

#
# Encryption Service name
#   the default one is 'basic'
#   a more powerful one named 'jasypt' is available
#       when installing the encryption feature
#
encryption.name = jasypt

#
# Encryption prefix
#
encryption.prefix = {CRYPT}

#
# Encryption suffix
#
encryption.suffix = {CRYPT}

#
# Set the encryption algorithm to use in Karaf JAAS login module
# Supported encryption algorithms follow:
#   MD2
#   MD5
#   SHA-1
#   SHA-256
#   SHA-384
#   SHA-512
#
encryption.algorithm = SHA-256
```

3. Karaf コンソールで **jaas:realms** コマンドを入力し、デプロイされたログインモジュールを表示します。

```
karaf@root()> jaas:realms

Index │ Realm Name │ Login Module Class Name

──────┼────────────┼──────────────────────────────────────────────────────────
──────────────────────────────

1     │ karaf      │ org.apache.karaf.jaas.modules.properties.PropertiesLoginModule

2     │ karaf      │ org.apache.karaf.jaas.modules.publickey.PublickeyLoginModule

3     │ karaf      │ org.apache.karaf.jaas.modules.audit.FileAuditLoginModule

4     │ karaf      │ org.apache.karaf.jaas.modules.audit.LogAuditLoginModule

5     │ karaf      │ org.apache.karaf.jaas.modules.audit.EventAdminAuditLoginModule
```

4. 次のコマンドを入力して、ユーザーを作成します。

```
karaf@root()> jaas:realm-manage --index 1
karaf@root()> jaas:user-list
User Name │ Group      │ Role
──────────┼────────────┼──────────
admin     │ admingroup │ admin
admin     │ admingroup │ manager
admin     │ admingroup │ viewer
admin     │ admingroup │ systembundles
admin     │ admingroup │ ssh
karaf@root()> jaas:useradd usertest test123
karaf@root()> jaas:group-add usertest admingroup
karaf@root()> jaas:update
karaf@root()> jaas:realm-manage --index 1
karaf@root()> jaas:user-list
User Name │ Group      │ Role
──────────┼────────────┼──────────
admin     │ admingroup │ admin
admin     │ admingroup │ manager
admin     │ admingroup │ viewer
admin     │ admingroup │ systembundles
admin     │ admingroup │ ssh
usertest  │ admingroup │ admin
usertest  │ admingroup │ manager
usertest  │ admingroup │ viewer
usertest  │ admingroup │ systembundles
usertest  │ admingroup │ ssh
```

5. **$FUSE_HOME/etc/users.properties** ファイルを見ると、ユーザー **usertest** がファイルに追加されていることがわかります。

```
admin =
{CRYPT}WXX+4PM2G7nT045ly4iS0EANsv9H/VwmStGIb9bcbGhFH5RgMuL0D3H/GVTigpga
{CRYPT},_g_:admingroup

_g_\:admingroup = group,admin,manager,viewer,systembundles,ssh

usertest =
{CRYPT}33F5E76E5FF97F3D27D790AAA1BEE36057410CCDBDBE2C792239BB2853D176
54315354BB8B608AD5{CRYPT},_g_:admingroup
```

6. すでに **jaas:update** コマンドを実行しているので、新たに作成したログインを別のターミナルでテストできます。

2.1.11. JAAS と HTTP Basic 認証のインテグレーション

Servlet REST を使用すると、REST DSL を使用して Camel ルートに REST エンドポイントを定義できます。次の例は、HTTP 基本認証によって保護されている REST エンドポイントがユーザー認証を Karaf JAAS サービスに委任する方法を示しています。

手順

1. Apache Camel を **CamelInstallDir** にインストールした場合、以下のディレクトリーでサンプルを見つけることができます。

```
CamelInstallDir/examples/camel-example-servlet-rest-karaf-jaas
```

2. Maven を使用して、サンプルを OSGi バンドルとしてビルドおよびインストールします。コマンドプロンプトを開き、現在のディレクトリーを **CamelInstallDir/examples/camel-example-servlet-rest-karaf-jaas** に切り替え、以下のコマンドを入力します。

```
mvn install
```

3. セキュリティー設定ファイルを **KARAF_HOME/etc** フォルダーにコピーするには、以下のコマンドを入力します。

```
cp src/main/resources/org.ops4j.pax.web.context-camelrestdsl.cfg $KARAF_HOME/etc
```

4. Karaf に Apache Camel をインストールするには、Karaf シェルコンソールで次のコマンドを入力します。

```
feature:repo-add camel ${project.version}
feature:install camel
```

5. **camel-servlet**、**camel-jackson**、および **war** Karaf 機能も必要で、以下のコマンドを入力してこれらの機能をインストールします。

```
feature:install camel-servlet
feature:install camel-jackson
feature:install war
```

6. **camel-example-servlet-rest-karaf-jaas** サンプルをインストールするには、以下のコマンドを入力します。

```
install -s mvn:org.apache.camel.example/camel-example-servlet-rest-karaf-jaas/${project.version}
```

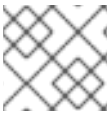
結果

アプリケーションが実行中であることを確認するには、以下のコマンドを入力してアプリケーションログファイルを表示できます (ログの表示を停止する場合は **ctrl+c** を使用)。

```
log:tail
```

REST **user** エンドポイントは以下の操作をサポートします。

- **GET /user/{id}** - 指定 ID を持つユーザーを表示する
- **GET /user/final** - すべてのユーザーを表示する
- **PUT /user** - ユーザーを更新/作成する



注記

view 操作は **HTTP GET** を使用し、update 操作は **HTTP PUT** を使用します。

2.1.11.1. Web ブラウザーから REST サービスにアクセス

以下の例を使用して、Web ブラウザーからサービスにアクセスできます (**admin** をユーザー、**admin** をパスワードとしてポップアップダイアログボックスに入力する必要があります)。

例: ユーザー ID 123 を表示

```
http://localhost:8181/camel-example-servlet-rest-blueprint/rest/user/123
```

例: すべてのユーザーをリスト表示

```
http://localhost:8181/camel-example-servlet-rest-blueprint/rest/user/findAll
```

2.1.11.2. コマンドラインから REST サービスへのアクセス

以下の例のように、コマンドラインから **curl** を使用して REST **user** エンドポイントにアクセスできます。

例: ユーザー ID 123 を表示

```
curl -X GET -H "Accept: application/json" --basic -u admin:admin http://localhost:8181/camel-example-servlet-rest-blueprint/rest/user/123
```

例: すべてのユーザーを表示します

```
curl -X GET -H "Accept: application/json" --basic -u admin:admin http://localhost:8181/camel-example-servlet-rest-blueprint/rest/user/findAll
```

例: ユーザー ID 234 を作成または更新します

```
curl -X PUT -d "{ \"id\": 234, \"name\": \"John Smith\"}" -H "Accept: application/json" --basic -u
admin:admin http://localhost:8181/camel-example-servlet-rest-blueprint/rest/user
```

2.2. ロールベースのアクセス制御

概要

このセクションでは、Karaf コンテナーでデフォルトで有効になっているロールベースのアクセス制御 (RBAC) 機能について説明します。標準のロール (**manager**、**admin** など) をユーザーのクレデンシャルに追加すると、すぐに RBAC 機能を活用できます。より高度な使用法として、各ロールが実行できることを正確に制御するために、アクセス制御リストをカスタマイズするオプションがあります。独自の OSGi サービスにカスタム ACL を適用するオプションもあります。

2.2.1. ロールベースのアクセス制御の概要

デフォルトでは、Fuse ロールベースアクセス制御は、Fuse 管理コンソール、JMX 接続、および Karaf コマンドコンソールを介したアクセスを保護します。デフォルトのアクセス制御レベルを使用するには、ユーザー認証データに標準ロールを追加します (例: **users.properties** ファイルを編集して)。関連するアクセス制御リスト (ACL) ファイルを編集して、アクセス制御をカスタマイズするオプションもあります。

メカニズム

Karaf のロールベースアクセス制御は、次のメカニズムに基づいています。

JMX Guard

Karaf コンテナーは JMX ガードで設定されています。このガードは、受信するすべての JMX 呼び出しをインターセプトし、設定された JMX アクセス制御リストを介して呼び出しをフィルタリングします。JMX ガードは JVM レベルで設定されているため、例外なく **すべての** JMX 呼び出しをインターセプトします。

OSGi サービスガード

いずれの OSGi サービスでも、OSGi サービスガードを設定できます。OSGi サービスガードはプロキシーオブジェクトとして実装され、クライアントと元の OSGi サービスの間に介入します。OSGi サービスガードは、OSGi サービスごとに明示的に設定する必要があり、デフォルトではインストールされません (事前設定されている Karaf コンソールコマンドを表す OSGi サービスを除く)。

保護タイプ

ロールベースのアクセス制御の Fuse 実装は、次のタイプの保護を提供できます。

Fuse Console (Hawtio)

Fuse Console (Hawtio) を介したコンテナーアクセスは、JMXACL ファイルによって制御されます。Fuse Console を提供する REST/HTTP サービスは、JMX の上に階層化された Jolokia テクノロジーを使用して実装されます。したがって、最終的に、すべての Fuse Console 呼び出しは JMX を通過し、JMXACL によって規制されます。

JMX

Karaf コンテナーの JMX ポートへの直接アクセスは、JMX ACL によって規制されています。さらに、JMX ガードは JVM レベルで設定されているため、Karaf コンテナーで実行されているアプリケーションによって開かれた追加の JMX ポートも JMX ACL によって規制されます。

Karaf コマンドコンソール

Karaf コマンドコンソールへのアクセスは、コマンドコンソール ACL ファイルによって規制されま

す。Karaf コンソールへのアクセス方法に関係なく、アクセス制御が適用されます。Fuse Console と SSH プロトコルのどちらを使用してコマンドコンソールにアクセスしても、アクセス制御が適用されます。

注記

コマンドラインで Karaf コンテナーを直接起動し (**./bin/fuse** スクリプトを使用するなどして)、ユーザー認証が実行されない特別なケースでは、**etc/system.properties** ファイルの **karaf.local.roles** プロパティーで指定されたロールを自動的に取得します。

OSGi サービス

Karaf コンテナーにデプロイされた OSGi サービスの場合、オプションで ACL ファイルを有効にできます。これにより、メソッド呼び出しは特定のロールに制限されます。

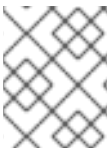
ロールのユーザーへの追加

ロールベースのアクセス制御システムでは、ユーザー認証データにロールを追加することにより、ユーザーにパーミッションを与えることができます。たとえば、**etc/users.properties** ファイルの以下のエントリーは **admin** ユーザーを定義し、**admin** ロールを割り当てます。

```
admin = secretpass,group,admin,manager,viewer,systembundles,ssh
```

また、ユーザーグループを定義してから、ユーザーを特定のユーザーグループに割り当てるオプションもあります。たとえば、以下のように **admingroup** ユーザーグループを定義および使用できます。

```
admin = secretpass, _g_:admingroup

_g_\:admingroup = group,admin,manager,viewer,systembundles,ssh
```



注記

ユーザーグループは、すべてのタイプの JAAS ログインモジュールでサポートされているわけではありません。

標準のロール

表2.2「アクセス制御の標準的なロール」 は、JMX ACL およびコマンドコンソール ACL 全体で使用される標準のロールをリスト表示して説明しています。

表2.2 アクセス制御の標準的なロール

ロール	説明
viewer	Karaf コンテナーへの読み取り専用アクセスを許可します。

ロール	説明
manager	アプリケーションのデプロイや実行を行う通常のユーザーに、適切なレベルで読み取り/書き込みアクセスを許可します。ただし、機密性の高い Karaf コンテナー設定オプションへのアクセスをブロックします。
admin	Karaf コンテナーへのアクセスを無制限に許可します。
ssh	Karaf コマンドコンソールに (ssh ポート経由で) 接続する権限をユーザーに付与します。

ACL ファイル

標準的な ACL ファイルは、以下のように Fuse インストールの **etc/auth/** ディレクトリーにあります。

etc/auth/jmx.acl[.*].cfg

JMX ACL ファイル。

etc/auth/org.apache.karaf.command.acl.*.cfg

コマンドコンソール ACL ファイル。

ロールベースアクセス制御のカスタマイズ

デフォルトでは、JMX ACL ファイルとコマンドコンソール ACL ファイルの完全なセットが提供されます。システム要件に合わせ、必要に応じてこれらの ACL を自由にカスタマイズできます。詳しい方法は、次のセクションに記載されています。

アクセス制御に使用する追加プロパティー

etc ディレクトリーの **system.properties** ファイルには、Karaf コマンドコンソールおよび Fuse Console (Hawtio) を介してアクセスを制御する以下の追加プロパティーが提供されます。

karaf.local.roles

ユーザーが Karaf コンテナーコンソールを (たとえば、スクリプトを実行して) **ローカルで** 起動するときに適用されるロールを指定します。

hawtio.roles

Fuse Console を介して Karaf コンテナーにアクセスできるロールを指定します。この制約は、JMX ACL ファイルで定義されたアクセス制御に **加えて** 適用されます。

karaf.secured.command.compulsory.roles

コンソールコマンドが **etc/auth/org.apache.karaf.command.acl.*.cfg** コマンド ACL ファイルによって明示的に設定されていない場合に、Karaf コンソールコマンドの呼び出しに必要なデフォルトのロールを指定します。コマンドを呼び出すには、リストにあるロールの少なくとも 1 つを使用してユーザーを設定する必要があります。値は、ロールのコンマ区切りリストとして指定されます。

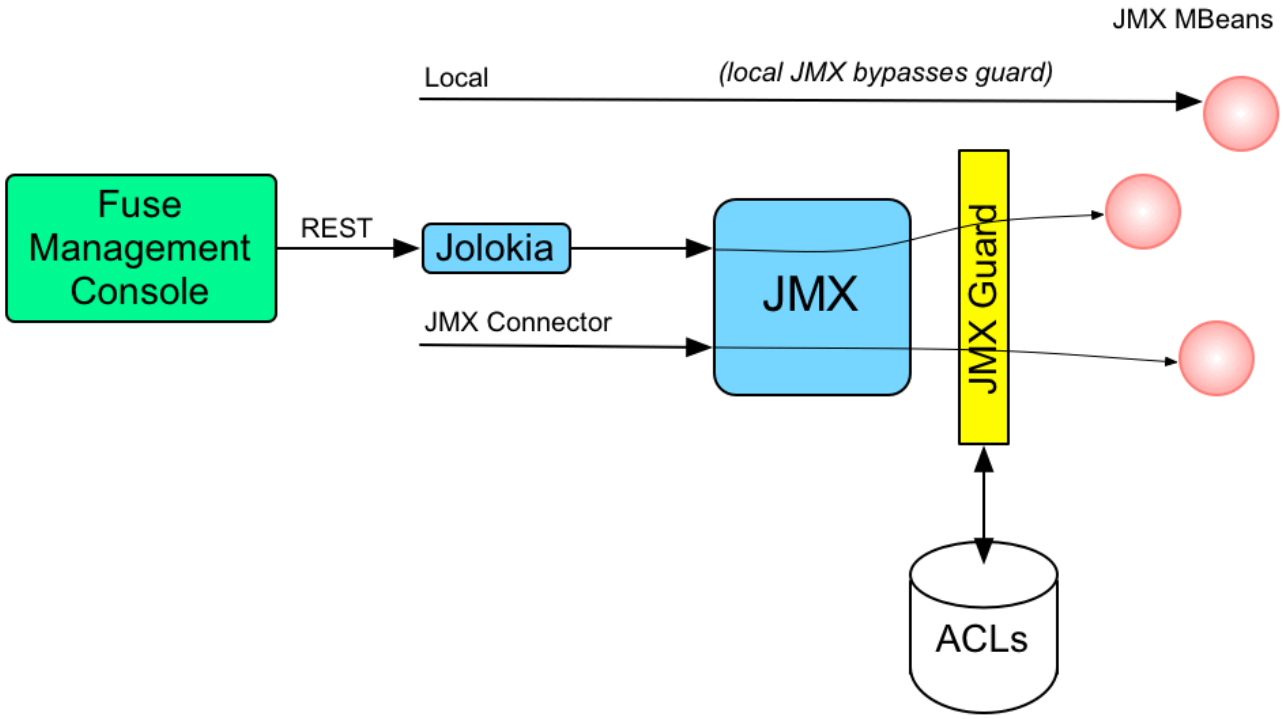
2.2.2. JMX ACL のカスタマイズ

JMX ACL は OSGi Config Admin Service に保存され、通常はファイル **etc/auth/jmx.acl.*.cfg** としてアクセスできます。このセクションでは、JMX ACL ファイルを自分で編集して JMX ACL をカスタマイズする方法について説明します。

アーキテクチャー

図2.1「JMX のアクセス制御メカニズム」 は、Karaf コンテナーへの JMX 接続のロールベースアクセス制御メカニズムの概要を示しています。

図2.1 JMX のアクセス制御メカニズム



仕組み

JMX アクセス制御は、特別な **javax.management.MBeanServer** オブジェクトを介して JMX へのリモートアクセスを提供することで動作します。このオブジェクトは、JMX ガードと呼ばれる **org.apache.karaf.management.KarafMBeanServerGuard** オブジェクトを呼び出すことによってプロキシーとして機能します。JMX ガードは、起動ファイルに特別な設定をしなくても使用できます。

JMX アクセス制御は次のように適用されます。

1. 非ローカルの JMX 呼び出しごとに、実際の MBean 呼び出しの前に JMX ガードが呼び出されます。
2. JMX Guard は、ユーザーがアクセスしようとしている MBean に関連する ACL を検索します (ACL は OSGi Config Admin サービスに格納されています)。
3. ACL は、MBean でこの特定の呼び出しを行うことが許可されているロールのリストを返します。
4. JMX Guard は、ロールリストを現在のセキュリティーサブジェクト (JMX 呼び出しを行っているユーザー) と照合して、現在のユーザーが必要なロールのいずれかを持っているか確認します。
5. 一致するロールが見つからない場合、JMX 呼び出しがブロックされ、**SecurityException** が発生します。

JMX ACL ファイルの場所

JMX ACL ファイルは **InstallDir/etc/auth** ディレクトリーにあります。ACL ファイルの名前は次の命名規則に従います。

```
etc/auth/jmx.acl[.*].cfg
```

技術的には、ACL は、パターン **jmx.acl[.*]** と一致する OSGi 永続 ID (PID) にマッピングされます。Karaf コンテナーは、デフォルトで OSGi PID をファイル **PID.cfg** として **etc/** ディレクトリー内に保存します。

MBean を ACL ファイル名にマッピング

JMX Guard は、JMX を介してアクセスされる **すべての** MBean クラス (独自のアプリケーションコードで定義する MBean を含む) にアクセス制御を適用します。特定の MBean クラスの ACL ファイルは、**jmx.acl** を接頭辞として追加して、MBean のオブジェクト名から派生します。たとえば、オブジェクト名が **org.apache.camel:type=context** によって付与される MBean の場合、対応する PID は以下のようになります。

```
jmx.acl.org.apache.camel.context
```

OSGi Config Admin サービスは、この PID データを次のファイルに保管します。

```
etc/auth/jmx.acl.org.apache.camel.context.cfg
```

ACL ファイル形式

JMX ACL ファイルの各行は、次の形式のエントリーです。

```
Pattern = Role1[,Role2][,Role3]...
```

Pattern は、MBean のメソッド呼び出しと一致するパターンで、等号記号の右側は、その呼び出しを行う権限をユーザーに付与するロールのコンマ区切りリストです。最も単純なケースでは、**Pattern** は単にメソッド名です。たとえば、(**jmx.acl.hawtio.OSGiTools.cfg** ファイルから)**jmx.acl.hawtio.OSGiTools** MBean の次の設定の場合は、以下のようになります。

```
getResourceURL = admin, manager, viewer
getLoadClassOrigin = admin, manager, viewer
```

また、複数のメソッド名と一致するためにワイルドカード文字 * を使用することもできます。たとえば、以下のエントリーは、名前が **set** で始まるすべてのメソッドを呼び出す権限を付与します。

```
set* = admin, manager, viewer
```

ただし、ACL 構文では、メソッド呼び出しのより詳細な制御を定義することもできます。特定の引数または正規表現に一致する引数で呼び出されたメソッドに一致するパターンを定義できます。たとえば、**org.apache.karaf.config** MBean パッケージの ACL はこの機能を利用して、通常のユーザーが機密性の高い設定を変更できないようにします。このパッケージの **create** メソッドは、以下のように制限されます。

```
create(java.lang.String)[/jmx[.]acl.*/] = admin
create(java.lang.String)[/org[.]apache[.]karaf[.]command[.]acl.+/] = admin
create(java.lang.String)[/org[.]apache[.]karaf[.]service[.]acl.+/] = admin
```

```
create(java.lang.String) = admin, manager
```

この場合、**manager** ロールには **create** メソッドを呼び出すパーミッションがありますが、**admin** ロールにのみ、**jmx.acl.***、**org.apache.karaf.command.acl.***、または **org.apache.karaf.service.*** に一致する PID 引数を使用して **create** を呼び出すパーミッションがあります。

ACL ファイルフォーマットの詳細は、**etc/auth/jmx.acl.cfg** ファイルのコメントを参照してください。

ACL ファイル階層

多くの場合、すべての MBean に ACL ファイルを提供することは非現実的であるため、Java パッケージのレベルで ACL ファイルを指定するオプションがあります。これにより、そのパッケージ内の **すべて** の MBean にデフォルト設定が提供されます。たとえば、**org.apache.cxf.Bus** MBean は、以下の PID レベルの**いずれか**で ACL 設定による影響を受ける可能性があります。

```
jmx.acl.org.apache.cxf.Bus
jmx.acl.org.apache.cxf
jmx.acl.org.apache
jmx.acl.org
jmx.acl
```

最も具体的な PID (リストの一番上) が最も具体的でない PID (リストの一番下) よりも優先される場合。

ルート ACL 定義

ルート ACL ファイル **jmx.acl.cfg** は、**すべて**の MBean に対してデフォルトの ACL 設定を提供するため、特別なケースです。ルート ACL には、デフォルトで次の設定があります。

```
list* = admin, manager, viewer
get* = admin, manager, viewer
is* = admin, manager, viewer
set* = admin
* = admin
```

これは、典型的な**読み取り**メソッドパターン (**list***、**get***、**is***) はすべての標準ロールにアクセスでき、通常の **書き込み** メソッドパターンおよびその他のメソッド (**set*** および *****) は admin ロール (**admin**) のみにアクセスできる事を意味します。

パッケージ ACL 定義

etc/auth/jmx.acl[.*].cfg で提供される標準の JMX ACL ファイルの多くは MBean パッケージに適用されます。たとえば、**org.apache.camel.endpoints** MBean パッケージの ACL は以下のパーミッションで定義されます。

```
is* = admin, manager, viewer
get* = admin, manager, viewer
set* = admin, manager
```

カスタム MBean の ACL

独自のアプリケーションでカスタム MBean を定義する場合、これらのカスタム MBean は ACL メカニズムと自動的に統合され、Karaf コンテナーにデプロイするときに JMX Guard によって保護されます。

ただしデフォルトでは、通常 MBean はデフォルトのルート ACL ファイル **jmx.acl.cfg** によってのみ保護されます。MBean に対してより細かな ACL を定義する場合は、標準の JMX ACL ファイルの命名規則を使用して、**etc/auth** 下に新しい ACL ファイルを作成します。

たとえば、カスタム MBean クラスに JMX オブジェクト名 **org.example:type=MyMBean** がある場合は、**etc/auth** ディレクトリー下に以下の名前を持つ新しい ACL ファイルを作成します。

```
jmx.acl.org.example.MyMBean.cfg
```

実行時の動的設定

OSGi Config Admin サービスは動的であるため、システムの実行中や、特定のユーザーがログオンしている間も、ACL 設定を変更できます。したがって、システムの実行中にセキュリティー違反を発見した場合は、Karaf コンテナーを再起動しなくても、関連する ACL ファイルを編集することで、システムの特定の部分へのアクセスをすぐに制限できます。

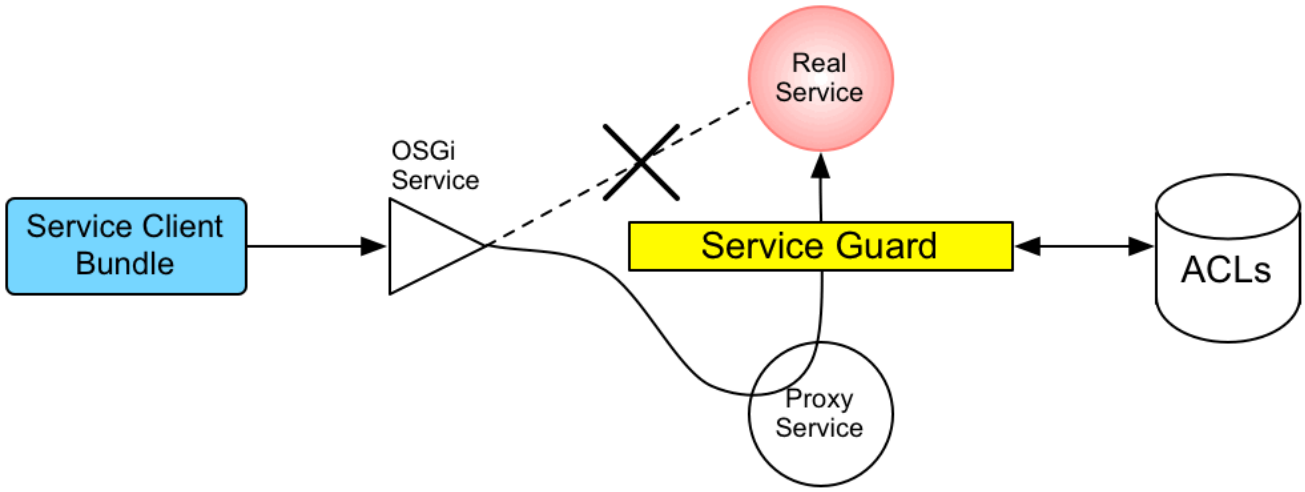
2.2.3. コマンドコンソール ACL のカスタマイズ

コマンドコンソール ACL は OSGi Config Admin Service に保存され、通常はファイル **etc/auth/org.apache.karaf.command.acl.*.cfg** としてアクセスできます。このセクションでは、コマンドコンソール ACL ファイルを自分で編集してコマンドコンソール ACL をカスタマイズする方法について説明します。

アーキテクチャー

図2.2「OSGi サービスのアクセス制御メカニズム」 は、Karaf コンテナー内の OSGi サービスに対するロールベースアクセス制御メカニズムの概要を示しています。

図2.2 OSGi サービスのアクセス制御メカニズム



仕組み

コマンドコンソールのアクセス制御メカニズムは、実際には、OSGi サービスの一般的なアクセス制御メカニズムに基づいています。そのため、コンソールコマンドが実装され、OSGi サービスとして公開されることがあります。Karaf コンソール自体は、OSGi Service Registry を介して使用可能なコマンドを検出し、OSGi サービスとしてコマンドにアクセスします。したがって、OSGi サービスのアクセス制御メカニズムを使用して、コンソールコマンドへのアクセスを制御できます。

OSGi サービスの保護メカニズムは、OSGi Service Registry フックに基づいています。これは高度な OSGi 機能であり、特定のコンシューマーを隠し OSGi サービスを隠し、OSGi サービスをプロキシーサービスに置き換えることができます。

特定の OSGi サービスにサービスガードが設定されている場合、OSGi サービスでのクライアント呼び出しは次のように進行します。

1. 呼び出しは、要求された OSGi サービスに直接送信 **されません**。代わりに、要求は、元のサービスと同じサービスプロパティー (およびいくつかの追加) を持つ置換プロキシーサービスにルーティングされます。
2. サービスガードは、ターゲット OSGi サービスに関連する ACL を検索します (ACL は OSGi Config Admin サービスに格納されています)。
3. ACL は、サービスでこの特定のメソッド呼び出しを行うことが許可されているロールのリストを返します。
4. このコマンドの ACL が見つからない場合、サービスガードはデフォルトで **etc/system.properties** ファイルの **karaf.secured.command.compulsory.roles** プロパティーに指定されたロールのリストになります。
5. サービスガードは、ロールリストを現在のセキュリティーサブジェクト (メソッド呼び出しを行っているユーザー) と照合して、現在のユーザーが必要なロールのいずれかを持っているか確認します。
6. 一致するロールが見つからない場合、メソッド呼び出しがブロックされ、**SecurityException** が発生します。
7. または、一致するロールが見つかった場合、メソッド呼び出しは元の OSGi サービスに委譲されます。

デフォルトのセキュリティーロールの設定

対応する ACL ファイルがないコマンドの場合は、**etc/system.properties** ファイルに **karaf.secured.command.compulsory.roles** プロパティーを設定し (ロールのコンマ区切りリストとして指定)、デフォルトのセキュリティーロールのリストを指定します。

コマンドコンソール ACL ファイルの場所

コマンドコンソールの ACL ファイルは、接頭辞 **org.apache.karaf.command.acl** で **InstallDir/etc/auth** ディレクトリーにあります。

コマンドスコープを ACL ファイル名にマッピング

コマンドコンソールの ACL ファイル名は、次の規則に従います。

```
etc/auth/org.apache.karaf.command.acl.CommandScope.cfg
```

CommandScope は、Karaf コンソールコマンドの特定グループの接頭辞に対応します。たとえば、**feature:install** および **features:uninstall** コマンドは、対応する ACL ファイル (**org.apache.karaf.command.acl.features.cfg**) を持つ **feature** コマンドスコープに属します。

ACL ファイル形式

コマンドコンソール ACL ファイルの各行は、次の形式のエントリーです。

```
Pattern = Role1[,Role2][,Role3]...
```

Pattern は、現在のコマンドスコープから Karaf コンソールコマンドと一致するパターンで、等号記号の右側は、その呼び出しを行うユーザー権限を付与するロールのコンマ区切りリストです。最も単純なケースでは、**Pattern** は、単にスコープのないコマンド名です。たとえば、**org.apache.karaf.command.acl.feature.cfg** ACL ファイルには、**feature** コマンドの以下のルールが含まれます。

```
list = admin, manager, viewer
repo-list = admin, manager, viewer
info = admin, manager, viewer
version-list = admin, manager, viewer
repo-refresh = admin, manager
repo-add = admin, manager
repo-remove = admin, manager
install = admin
uninstall = admin
```

重要

特定のコマンド名に一致するものが見つからない場合、そのコマンドにロールは不要であると見なされており、任意のユーザーが呼び出すことができます。

特定の引数または正規表現に一致する引数で呼び出されたコマンドに一致するパターンも定義できます。たとえば、**org.apache.karaf.command.acl.bundle.cfg** ACL ファイルはこの機能を利用して、通常のユーザーが **-f** (force) フラグで **bundle:start** コマンドと **bundle:stop** コマンドを呼び出さないようにします (システムバンドルを管理するために指定する必要があります)。この制限は、ACL ファイルで次のようにコーディングされています。

```
start[/.*[-][f].*/] = admin
start = admin, manager
stop[/.*[-][f].*/] = admin
stop = admin, manager
```

この場合、**manager** ロールには、通常 **bundle:start** および **bundle:stop** コマンドを呼び出す権限がありますが、**admin** ロールのみが force オプション **-f** でこれらのコマンドを呼び出す権限を持ちます。

ACL ファイルフォーマットの詳細は、**etc/auth/org.apache.karaf.command.acl.bundle.cfg** ファイルのコメントを参照してください。

実行時の動的設定

コマンドコンソールの ACL 設定は完全に動的です。つまり、システム実行中に ACL 設定を変更でき、すでにログオンしているユーザーであっても、変更は数秒以内に反映されます。

2.2.4. OSGi サービスの ACL 定義

任意の OSGi サービス (システムレベルまたはアプリケーションレベル) のカスタム ACL を定義することができます。デフォルトでは、OSGi サービスでアクセス制御は有効になっていません (コマンドコンソール ACL ファイルで事前に設定されている Karaf コンソールコマンドを公開する OSGi サービスを除く)。このセクションでは、OSGi サービスのカスタム ACL を定義する方法と、指定されたロールを使用してそのサービスでメソッドを呼び出す方法について説明します。

ACL ファイル形式

OSGi サービス ACL ファイルには、次のように、その ACL が適用される OSGi サービスを識別する 1 つの特別なエントリーがあります。

```
service.guard = (objectClass=InterfaceName)
```

service.guard の値は、一致する OSGi サービスを選択するために OSGi サービスプロパティーのレジストリーに適用される LDAP 検索フィルターです。最も単純なフィルタータイプ **(objectClass=InterfaceName)** は、指定された Java インターフェイス名 **InterfaceName** で OSGi サービスを選択します。

ACL ファイルの残りのエントリーは、次の形式です。

```
Pattern = Role1[,Role2][,Role3]...
```

Pattern は、サービスメソッドと一致するパターンで、等号記号の右側は、その呼び出しを行う権限をユーザーに付与するロールのコンマ区切りリストです。これらのエントリーの構文は、基本的に JMX ACL ファイルのエントリーと同じです。これについては、「ACL ファイル形式」 を参照してください。

カスタム OSGi サービスの ACL を定義する方法

カスタム OSGi サービスの ACL を定義するには、以下の手順を実行します。

1. 一般的には、Java インターフェイスを使用して OSGi サービスを定義します (通常の Java クラスを使用することもできますが、推奨しません)。たとえば、OSGi サービスとして公開する予定の Java インターフェイス **MyService** について考えてみましょう。

```
package org.example;

public interface MyService {
  void doit(String s);
}
```

2. Java インターフェイスを OSGi サービスとして公開するには、通常 **service** 要素を OSGi Blueprint XML ファイルに追加します (Blueprint XML ファイルは通常、Maven プロジェクトの **src/main/resources/OSGI-INF/blueprint** ディレクトリーに保存されます)。たとえば、**MyServiceImpl** が **MyService** インターフェイスを実装するクラスであると仮定すると、以下のように **MyService** OSGi サービスを公開できます。

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           default-activation="lazy">

  <bean id="myserviceimpl" class="org.example.MyServiceImpl"/>

  <service id="myservice" ref="myserviceimpl" interface="org.example.MyService"/>

</blueprint>
```

3. OSGi サービスに ACL を定義するには、**org.apache.karaf.service.acl** 接頭辞が付いた OSGi Config Admin PID を作成する必要があります。

たとえば、Karaf コンテナーの場合 (OSGi Config Admin PID は **etc/auth/** ディレクトリー下に **.cfg** ファイルとして格納される)、**MyService** OSGi サービスに以下の ACL ファイルを作成できます。

```
etc/auth/org.apache.karaf.service.acl.myservice.cfg
```

注記

必要な接頭辞 **org.apache.karaf.service.acl** で始まる限り、このファイルの命名方法は重要ではありません。この ACL ファイルに対応する OSGi サービスは、実際にはこのファイルのプロパティー設定によって指定されます (次の手順で説明しています)。

4. ACL ファイルのコンテンツを次のような形式で指定します。

```
service.guard = (objectClass=InterfaceName)
Pattern = Role1[,Role2][,Role3]...
```

service.guard 設定は OSGi サービスの **InterfaceName** を指定します (LDAP 検索フィルターの構文を使用)。これは OSGi サービスプロパティーに適用されます。ACL ファイルの他のエントリーは、一致するメソッドを指定されたロールに関連付ける **Pattern** メソッドで設定されます。たとえば、**org.apache.karaf.service.acl.myservice.cfg** ファイルで以下の設定を使用して、**MyService** OSGi サービスの簡単な ACL を定義できます。

```
service.guard = (objectClass=org.example.MyService)
doit = admin, manager, viewer
```

5. 最後に、この OSGi サービスの ACL を有効にするには、**etc/system.properties** ファイルの **karaf.secured.services** プロパティーを編集する必要があります。**karaf.secured.services** プロパティーの値は LDAP 検索フィルターの構文を持ちます (OSGi サービスプロパティーに適用される)。一般的に、OSGi サービス **ServiceInterface** の ACL を有効にするには、以下のようにこのプロパティーを変更する必要があります。

```
karaf.secured.services=(|(objectClass=ServiceInterface)(...ExistingPropValue...))
```

たとえば、**MyService** OSGi サービスを有効にするには、以下を行います。

```
karaf.secured.services=(|(objectClass=org.example.MyService)(&(osgi.command.scope=*)
(osgi.command.function=*)))
```

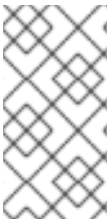
karaf.secured.services プロパティーの初期値には、コマンドコンソール ACL を有効にする設定があります。これらのエントリーを削除または破損すると、コマンドコンソール ACL が機能しなくなる場合があります。

RBAC で保護された OSGi サービスを呼び出す方法

カスタム OSGi サービスでメソッドを呼び出す Java コードを記述する場合 (つまり、OSGi サービスのクライアントを実装する場合)、Java セキュリティー API を使用して、サービスの呼び出しに使用するロールを指定する必要があります。たとえば、**manager** ロールを使用して **MyService** OSGi サービスを呼び出すには、以下のようなコードを使用します。

```
// Java
import javax.security.auth.Subject;
```

```
import org.apache.karaf.jaas.boot.principal.RolePrincipal;
// ...
Subject s = new Subject();
s.getPrincipals().add(new RolePrincipal("Deployer"));
Subject.doAs(s, new PrivilegedAction() {
  public Object run() {
    svc.doit("foo"); // invoke the service
  }
}
```



注記

この例では、Karaf ロールタイプ **org.apache.karaf.jaas.boot.principal.RolePrincipal** を使用します。必要な場合は、代わりに独自のカスタムロールクラスを使用することもできますが、その場合は OSGi サービスの ACL ファイルで構文 **className:roleName** を使用してロールを指定する必要があります。

OSGi サービスに必要なロールを見つける方法

ACL によって保護された OSGi サービスに対してコードを記述する場合、サービスの呼び出しが許可されているロールを確認すると便利な場合があります。このため、プロキシーサービスは、追加の OSGi プロパティー **org.apache.karaf.service.guard.roles** をエクスポートします。このプロパティーの値は **java.util.Collection** オブジェクトで、そのサービスでメソッドを呼び出す可能性があるすべてのロールのリストが含まれます。

2.3. 暗号化されたプロパティープレースホルダーの使用方法

Karaf コンテナーを保護するときは、設定ファイルでプレインテキストのパスワードを使用しないでください。プレインテキストのパスワードの使用を回避する 1 つの方法は、可能な限り暗号化されたプロパティープレースホルダーを使用することです。詳細は以下のトピックを参照してください。

- 「値を暗号化するためのマスターパスワード」
- 「暗号化されたプロパティープレースホルダーの使用」
- 「**jasypt:digest** コマンドの呼び出し」
- 「**jasypt:decrypt** コマンドの呼び出し」

2.3.1. 値を暗号化するためのマスターパスワード

Jasypt を使用して値を暗号化するには、マスターパスワードが必要です。マスターパスワードは、ユーザー自身または管理者が選択できます。Jasypt では、いくつかの方法でマスターパスワードを設定できます。

まず、Blueprint 設定でプレインテキストを使用してマスターパスワードを指定する方法があります。以下はその例です。

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:enc="http://karaf.apache.org/xmlns/jasypt/v1.0.0">

  <enc:property-placeholder>
    <enc:encryptor class="org.jasypt.encryption.pbe.StandardPBEStringEncryptor">
      <property name="config">
```

```
      <bean class="org.jasypt.encryption.pbe.config.EnvironmentStringPBEConfig">
        <property name="algorithm" value="PBEWithMD5AndDES" />
        <property name="password" value="myPassword" />
      </bean>
    </property>
  </enc:encryptor>
</enc:property-placeholder>

</blueprint>
```

マスターパスワードをプレインテキストで指定する代わりに、次のいずれかを実行できます。

- 環境変数をマスターパスワードに設定します。Blueprint 設定ファイルで、この環境変数を **passwordEnvName** プロパティーの値として指定します。たとえば、**MASTER_PW** 環境変数をマスターパスワードに設定すると、Blueprint 設定ファイルにこのエントリーが作成されます。
  <property name="passwordEnvName" value="MASTER_PW">

- Karaf システムプロパティーをマスターパスワードに設定します。Blueprint 設定ファイルで、このシステムプロパティーを **passwordSys** プロパティーの値として指定します。たとえば、**karaf.password** システムプロパティーをマスターパスワードに設定すると、Blueprint 設定ファイルにこのエントリーが作成されます。
  <property name="passwordSys" value="karaf.password">

2.3.2. 暗号化されたプロパティープレースホルダーの使用

Karaf コンテナーを保護する場合は、Blueprint 設定ファイルで暗号化されたプロパティープレースホルダーを使用します。

前提条件

- 値を暗号化するためのマスターパスワードを知っている。

手順

1. デフォルトの暗号化アルゴリズム **PBEWithMD5AndDES** を使用するか、使用する暗号化アルゴリズムを次のように選択します。

   a. **jasypt:list-algorithms** コマンドを実行して、現在の Java 環境でサポートされるアルゴリズムを検出します。

   ```
   karaf@root()> jasypt:list-algorithms
   ```

   引数やオプションはありません。出力は、サポートされているダイジェストおよび Password Based Encryption (PBE) アルゴリズムの識別子リストです。このリストには、Fuse 7.13 の一部である Bouncy Castle ライブラリーによって提供されるアルゴリズムが含まれています。このリストは長くなる可能性があります。その短い部分は次のようになります。

   ```
   karaf@root()> jasypt:list-algorithms
   DIGEST ALGORITHMS:
    - 1.0.10118.3.0.55
    - 1.2.804.2.1.1.1.1.2.2.1
      ...
    - 2.16.840.1.101.3.4.2.9
   ```

```
 - BLAKE2B-160
 - BLAKE2B-256
   ...
 - MD4
 - MD5
 - OID.1.0.10118.3.0.55
   ...
 - SHA3-512
 - SKEIN-1024-1024
 - SKEIN-1024-384
   ...
 - TIGER
 - WHIRLPOOL

PBE ALGORITHMS:
 - PBEWITHHMACSHA1ANDAES_128
 - PBEWITHHMACSHA1ANDAES_256
    ...
 - PBEWITHSHA1ANDRC2_128
 - PBEWITHSHA1ANDRC2_40
    ...
 - PBEWITHSHAANDIDEA-CBC
 - PBEWITHSHAANDTWOFISH-CBC
```

b. リストを確認し、使用する暗号化アルゴリズムの識別子を見つけます。アルゴリズムの選択については、ご使用のサイトのセキュリティー専門家に相談することを推奨します。

2. 設定ファイルで使用するパスワードなど、機密性の高い設定値を暗号化するには、**jasypt:encrypt** コマンドを実行します。コマンドの形式は以下のとおりです。
jasypt:encrypt [options] [input]

オプションを指定せずにこのコマンドを呼び出し、暗号化する値を指定しない場合、コマンドによりマスターパスワードと暗号化する値の入力が求められ、他のオプションにはデフォルトが適用されます。以下に例を示します。

```
karaf@root()> jasypt:encrypt
Master password: ********
Master password (repeat): ********
Data to encrypt: *****
Data to encrypt (repeat): *****
Algorithm used: PBEWithMD5AndDES
Encrypted data: oT8/LImAFQmOfXxuFGRDTAjD1l1+GxKL+TnHxFNwX4A=
```

暗号化する値ごとに **jasypt:encrypt** コマンドを実行します。

デフォルトの動作を変更するには、次のオプションを 1 つ以上指定します。

オプション	説明	例

オプション	説明	例
-w または **--password-property**	このオプションの後に、マスターパスワードの値に設定されている環境変数またはシステムプロパティーを指定します。Jasypt は、この値を暗号化アルゴリズムと組み合わせて使用して、暗号化キーを取得します。 **-w** または **-W** オプションを指定しない場合は、コマンドの呼び出し後に、マスターパスワードの入力と確認が求められます。	**-w MASTER_PW**
-W または **--password**	このオプションの後に、選択したマスターパスワードのプレインテキスト値を指定します。マスターパスワードのプレインテキスト値は履歴に表示されます。 Jasypt は、この値を暗号化アルゴリズムと組み合わせて使用して、暗号化キーを取得します。 **-w** または **-W** オプションを指定しない場合は、コマンドの呼び出し後に、マスターパスワードの入力と確認が求められます。	**-W "M@s!erP#"**
-a または **--algorithm**	このオプションの後に、**jasypt:encrypt** コマンドが最初の暗号鍵を取得するのに使用するアルゴリズムの識別子を指定します。デフォルトは **PBEWithMD5AndDES** です。 **jasypt-list-algorithms** コマンドが出力するリストに含まれるすべてのアルゴリズムがサポートされます。コマンドラインでアルゴリズム名を指定すると、自動補完を利用できます。	例: **-a PBEWITHMD5ANDRC2**
-i または **--iterations**	このオプションの後に、初期キーのハッシュを繰り返し作成する回数を整数で指定します。繰り返すたびに、前のハッシュ結果を取得し、それを再度ハッシュします。その結果が最終的な暗号化キーです。デフォルトは 1000 です。	例 e: **-i 5000**
-h または **--hex**	16 進数の出力を取得するには、このオプションを指定します。デフォルトの出力は Base64 です。	例: **-h**
--help	コマンド構文とオプションに関する情報を表示します。	**jasypt:encrypt --help**

3. **jasypt:encrypt** コマンドを実行して、取得した暗号化された値が含まれるプロパティーファイルを作成します。**ENC()** 関数で暗号化された各値をラップします。
たとえば、一部の LDAP クレデンシャルを **etc/ldap.properties** ファイルに保存する場合は、ファイルの内容は次のようになります。

```
#ldap.properties
ldap.password=ENC(VMJ5S566MEDhQ5r6jiIqTB+fao3NN4pKnQ9xU0wiDCg=)
ldap.url=ldap://192.168.1.74:10389
```

4. 暗号化されたプロパティープレースホルダーに必要な namespace を **blueprint.xml** ファイルに追加します。これらの namespace は、Aries 拡張機能と ApacheKarafJasypt 用です。以下に例を示します。

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0"
  xmlns:enc="http://karaf.apache.org/xmlns/jasypt/v1.0.0">
...
</blueprint>
```

5. 使用した Jasypt 暗号化アルゴリズムの識別子とプロパティーファイルの場所を設定します。次の例は、以下を行う方法を示しています。

   - **etc/ldap.properties** ファイルからプロパティーを読み取るように **ext:property-placeholder** 要素を設定します。
   - **enc:property-placeholder** 要素を次のように設定します。
     - **PBEWithMD5AndDES** 暗号化アルゴリズムを特定します。
     - Karaf **bin/setenv** ファイルで定義した環境変数 **JASYPT_ENCRYPTION_PASSWORD** からマスターパスワードを読み取ります。

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0"
  xmlns:enc="http://karaf.apache.org/xmlns/jasypt/v1.0.0">

  <ext:property-placeholder>
    <ext:location>file:etc/ldap.properties</ext:location>
  </ext:property-placeholder>

  <enc:property-placeholder>
    <enc:encryptor class="org.jasypt.encryption.pbe.StandardPBEStringEncryptor">
      <property name="config">
        <bean class="org.jasypt.encryption.pbe.config.EnvironmentStringPBEConfig">
          <property name="algorithm" value="PBEWithMD5AndDES" />
          <property name="passwordEnvName"
value="JASYPT_ENCRYPTION_PASSWORD" />
        </bean>
      </property>
    </enc:encryptor>
  </enc:property-placeholder>
…
</blueprint>
```

初期化 Vector プロパティーの設定

以下のアルゴリズムでは、**ivGenerator** という名前の初期化ベクタープロパティーを Blueprint 設定に追加する必要があります。

```
PBEWITHHMACSHA1ANDAES_128
```

```
PBEWITHHMACSHA1ANDAES_256
PBEWITHHMACSHA224ANDAES_128
PBEWITHHMACSHA224ANDAES_256
PBEWITHHMACSHA256ANDAES_128
PBEWITHHMACSHA256ANDAES_256
PBEWITHHMACSHA384ANDAES_128
PBEWITHHMACSHA384ANDAES_256
PBEWITHHMACSHA512ANDAES_128
PBEWITHHMACSHA512ANDAES_256
```

以下の例は、必要に応じて、Blueprint 設定に **ivGenerator** プロパティーを追加する方法を示しています。

```
<enc:property-placeholder>
  <enc:encryptor class="org.jasypt.encryption.pbe.StandardPBEStringEncryptor">
    <property name="config">
      <bean class="org.jasypt.encryption.pbe.config.EnvironmentStringPBEConfig">
        <property name="algorithm" value="PBEWITHHMACSHA1ANDAES_128"/>
        <property name="passwordEnvName" value="JASYPT_ENCRYPTION_PASSWORD"/>
        <property name="ivGenerator">
          <bean class="org.jasypt.iv.RandomIvGenerator" />
        </property>
      </bean>
    </property>
  </enc:encryptor>
</enc:property-placeholder>
```

暗号化されたプロパティープレースホルダーを使用する LDAP JAAS レルム設定

以下の例は、Jasypt 暗号化プロパティープレースホルダーを使用する LDAP JAAS レルム設定を表示し、前述の例の **blueprint.xml** ファイルに追加します。

注記

このトピックで説明されているプロセスを使用してプロパティーを暗号化する場合は、**@PropertyInject** アノテーションを使用してプロパティーを復号化できません。代わりに、この Blueprint の例のように、XML を使用してプロパティーを Java オブジェクトに挿入します。

この例では、コンテナーの初期化中に **${ldap.password}** プレースホルダーは、**etc/ldap.properties** ファイルからの **ldap.password** プロパティーの復号化された値に置き換えられます。

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0"
  xmlns:enc="http://karaf.apache.org/xmlns/jasypt/v1.0.0">

  <ext:property-placeholder>
    <location>file:etc/ldap.properties</location>
  </ext:property-placeholder>

  <enc:property-placeholder>
    <enc:encryptor class="org.jasypt.encryption.pbe.StandardPBEStringEncryptor">
      <property name="config">
        <bean class="org.jasypt.encryption.pbe.config.EnvironmentStringPBEConfig">
          <property name="algorithm" value="PBEWithMD5AndDES" />
```

```
        <property name="passwordEnvName" value="JASYPT_ENCRYPTION_PASSWORD" />
      </bean>
    </property>
  </enc:encryptor>
 </enc:property-placeholder>

 <jaas:config name="karaf" rank="200">
  <jaas:module className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
flags="required">
    initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
    debug=true
     connectionURL=${ldap.url}
     connectionUsername=cn=mqbroker,ou=Services,ou=system,dc=jbossfuse,dc=com
     connectionPassword=${ldap.password}
     connectionProtocol=
     authentication=simple
     userRoleName=cn
     userBase = ou=User,ou=ActiveMQ,ou=system,dc=jbossfuse,dc=com
     userSearchMatching=(uid={0})
     userSearchSubtree=true
     roleBase = ou=Group,ou=ActiveMQ,ou=system,dc=jbossfuse,dc=com
     roleName=cn
     roleSearchMatching= (member:=uid={1})
     roleSearchSubtree=true
  </jaas:module>
 </jaas:config>

</blueprint>
```

環境変数またはシステムプロパティーの指定例

値を暗号化するときにプレインテキストのマスターパスワードを指定するのではなく、マスターパスワードに設定されている環境変数またはシステムプロパティーを指定できます。たとえば、**bin/setenv** ファイルに以下が含まれているとします。

```
export MASTER_PASSWORD=passw0rd
```

次のコマンドで値を暗号化できます。

```
karaf@root()> jasypt:encrypt -w MASTER_PASSWORD "$en$!t!ve"
Algorithm used: PBEWithMD5AndDES
Encrypted data: /4DZCwqXD7cQ++TKQjt9QzmmcWv7TwmylCPkHumv2LQ=
```

etc/system.properties ファイルに以下が含まれている場合:

```
master.password=passw0rd
```

次のコマンドで値を暗号化できます。

```
karaf@root()> jasypt:encrypt -w master.password "$en$!t!ve"
Algorithm used: PBEWithMD5AndDES
Encrypted data: 03+8UTJJtEXxHaJkVCmzhqLMUYtT8TBG2RMvOBQlfmQ=
```

2.3.3. **jasypt:digest** コマンドの呼び出し

Jasypt ダイジェストは、MD5 などの暗号化ハッシュ関数を値に適用した結果です。ダイジェストの生成は、一方向暗号化の一種です。ダイジェストを生成した後にダイジェストから元の値を再構築することはできません。特に機密性の高い値の場合、値を暗号化するのではなく、ダイジェストを生成することを推奨します。その後、ダイジェストをプロパティープレースホルダーとして指定できます。

ダイジェストを生成するコマンドを呼び出すための形式は次のとおりです。

jasypt:digest [options] [input]

オプションを指定せず、ダイジェストを作成する入力を指定しない場合、コマンドにより暗号化する値の指定を要求され、オプションのデフォルト値が適用されます。以下に例を示します。

```
karaf@root()> jasypt:digest
Input data to digest: ********
Input data to digest (repeat): ********
Algorithm used: MD5
Digest value:
8D4C0B3D5EE133BCFD7585A90F15C586741F814BC527EAE2A386B9AA6609B926AD9B3C418937
251373E08F18729AD2C93815A7F14D878AA0EF3268AA04729A614ECAE95029A112E9AD56FEDD
3FD7E28B73291C932B6F4C894737FBDE21AB382
```

以下の例は、コマンドラインでの **input** 引数の指定を示しています。

```
karaf@root()> jasypt:digest ImportantPassword
```

このコマンドは、デフォルトのオプションを適用し、**ImportantPassword** の一方向暗号化を提供するダイジェストを生成します。コマンド出力は次のようになります。

```
karaf@root()> jasypt:digest ImportantPassword
Algorithm used: MD5
Digest value:
0bL90nno/nHiTEdzx3dKa61LBDcWQQZMpjaONtY3b1fJBuDWbWTTtZ6tE5eOOPKh7orLTXS7XRt2bl
A2DrfnjWIIIETjge9n
```

一方向暗号化が必要な各値に **jasypt:digest** コマンドを実行します。

デフォルトの動作を変更するには、次のオプションを 1 つ以上指定します。

オプション	説明	例
-a または **--algorithm**	このオプションの後に、**jasypt:digest** コマンドでダイジェストを生成するために使用するダイジェストアルゴリズムの識別子を指定します。デフォルトは **MD5** です。 **jasypt-list-algorithms** コマンドが出力するリストに含まれるすべてのダイジェストアルゴリズムがサポートされます。コマンドラインでアルゴリズム名を指定すると、自動補完を利用できます。	例: **-a SHA-12**

オプション	説明	例
-i または **--iterations**	このオプションの後に、初期ダイジェストのハッシュを繰り返し作成する回数を整数で指定します。繰り返すたびに、前のハッシュ結果を取得し、それを再度ハッシュします。その結果が最終的なダイジェストです。デフォルトは 1000 です。	例 e: **-i 5000**
-s または **--salt-size**	このオプションの後に、**jasypt:digest** ダイジェストの作成に適用される salt のバイト数を示す整数を指定します。これは、機密性の高い値のダイジェストを生成する場合や、複数の場所にダイジェストを指定する必要がある場合に便利です。たとえば、同じ入力値で異なる salt サイズを使用して **jasypt:digest** を呼び出すことができます。入力が同じであっても、各コマンドは異なるダイジェストを生成します。デフォルトは 8 です。	例: **-s 12**
-h または **--hex**	16 進数の出力を取得するには、このオプションを指定します。デフォルトの出力は Base64 です。	例: **-h**
--help	コマンド構文とオプションに関する情報を表示します。	**jasypt:digest --help**

ダイジェストの取得後、暗号化されたプロパティープレースホルダーの使用 に記載されているのと同じ方法で使用できます。

デフォルト以外の値を使用すると、計算に時間がかかります。以下に例を示します。

```
karaf@root()> jasypt:digest --iterations 1000000 --salt-size 32 -a SHA-512 --hex passw0rd
Algorithm used: SHA-512
Digest value:
4007A85C4932A399D8376B4F2B3221E34F0AF349BB152BEAC80F03BEB2B368DA7900F0990C186DB36D61741FA147B96DC9F73481991506FAA3662EA1693642CDAB89EB7E6B1DC21E1443D06D70A5842EB2851D37E262D5FC77A1D0909B3B2783
```

2.3.4. jasypt:decrypt コマンドの呼び出し

暗号化されたプレースホルダーの元の値を確認するには、プレースホルダーで **jasypt:decrypt** コマンドを使用します。

前提条件

- **jasypt:encrypt** コマンドを呼び出してプレースホルダーを生成しておく必要があります。

以下を知っている必要があります。

- マスターパスワード、またはマスターパスワードとして使用する環境変数またはシステムプロパティー。
- **jasypt:encrypt** で使用される暗号化アルゴリズム。

- **jasypt:encrypt** 反復の数。

jasypt:decrypt コマンドを呼び出すための形式は次のとおりです。

jasypt:decrypt [options] [input]

注記

options と **input** を指定せずにコマンドを実行できますが、**jasypt:encrypt** コマンドでデフォルトを使用している場合に **限り** ます。

この場合、マスターパスワードと復号化する値を指定する必要があります。その他のオプションはすべてデフォルト値を持ちます。

例

この場合、マスターのパスワードおよびデータを入力して、プロンプトで復号化します。デフォルトのアルゴリズム **PBEWithMD5AndDES** は、値を復号化するための復号化キーを作成します。

```
karaf@root()> jasypt:decrypt
Master password: ********
Data to decrypt: ********************************************
Algorithm used: PBEWithMD5AndDES
Decrypted data: $en$!t!ve
```

2.3.4.1. jasypt:decrypt のオプションの指定

デフォルトの動作を変更するには、次のオプションを 1 つ以上指定します。

オプション	説明	注記	例
-w または **--password-property**	マスターパスワードの値に設定された環境変数またはシステムプロパティー。**Jasypt** は、この値を復号化アルゴリズムとともに使用して、最初の復号化キーを作成します。	**-w** または **-W** オプションを指定しない場合は、コマンドの呼び出し後に、マスターパスワードの入力と確認が求められます。	**-w MASTER_PW**
-W または **--password**	このオプションの後に、選択したマスターパスワードのプレインテキスト値を指定します。マスターパスワードのプレインテキスト値は履歴に表示されます。 **Jasypt** は、この値を復号化アルゴリズムと組み合わせて使用して、最初の復号化キーを取得します。	**-w** または **-W** オプションを指定しない場合は、コマンドの呼び出し後に、マスターパスワードの入力と確認が求められます。	**-W "M@s!erP#"**

オプション	説明	注記	例
-a または **--algorithm**	このオプションの後に、**jasypt:decrypt** コマンドが最初の復号鍵を取得するのに使用するアルゴリズムの識別子を指定します。デフォルトは **PBEWithMD5AndDES** です。 **jasypt-list-algorithms** コマンドが出力するリストに含まれるすべてのアルゴリズムがサポートされます。コマンドラインでアルゴリズム名を指定すると、自動補完を利用できます。	**jasypt:decrypt** コマンドは、指定したプレースホルダー入力を生成するために **jasypt:encrypt** コマンドが使用したものと同じアルゴリズムを使用する必要があります。	**-a PBEWITHMD5ANDRC2**
-i または **--iterations**	このオプションの後に、初期キーのハッシュを繰り返し作成する回数を整数で指定します。繰り返すたびに、前のハッシュ結果を取得し、それを再度ハッシュします。その結果が最終的な復号化キーです。デフォルトは 1000 です。	**jasypt:decrypt** コマンドは、指定したプレースホルダー入力を生成するために **jasypt:encrypt** コマンドが使用したものと同じ反復回数を使用する必要があります。	**-i 5000**
-h または **--hex**	16 進数の出力を取得するには、このオプションを指定します。デフォルトの出力は Base64 です。		**-h**
-E または **--use-empty-iv-generator**	以前のバージョンの Jasypt で暗号化したパスワードを復号化するには、固定された IV ジェネレーターを使用します。		**-E**
--help	コマンド構文とオプションに関する情報を表示します。		**--help**

2.3.4.2. 環境変数またはシステムプロパティーの指定

jasypt:decrypt コマンドに値をパラメーターとして追加する代わりに、環境変数またはシステムプロパティーを使用できます。

2.3.4.2.1. 環境変数の使用

環境変数を使用するには、パラメーターを **bin/setenv** ファイルに追加します。

例

```
export MASTER_PASSWORD=passw0rd
```

環境変数 **MASTER_PASSWORD** を使用して値を復号化できます。

例

```
karaf@root()> jasypt:decrypt -a -w MASTER_PASSWORD
Data to decrypt: ********************************************
Algorithm used: PBEWithMD5AndDES
Decrypted data: $en$!t!ve
```

2.3.4.2.2. システムプロパティーの使用

環境変数を使用するには、パラメーターを **etc/system.properties** ファイルに追加します。

例

```
master.password=passw0rd
```

このシステムプロパティー **master.password** を使用して値を復号化できます。

例

```
karaf@root()> jasypt:decrypt -w master.password
Data to decrypt: ********************************************
Algorithm used: PBEWithMD5AndDES
Decrypted data: $en$!t!ve
```

2.4. リモート JMX SSL の有効化

概要

Red Hat JBoss Fuse は、MBean を使用した Karaf コンテナーのリモート監視および管理を可能にする JMX ポートを提供します。ただし、デフォルトでは、JMX 接続を介して送信するクレデンシャルは暗号化されておらず、スヌーピングに対して脆弱です。JMX 接続を暗号化し、パスワードスヌーピングから保護するには、SSL で JMX を設定して JMX 通信を保護する必要があります。

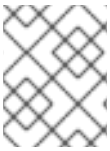
SSL で JMX を設定するには、次の手順を実行します。

1. jbossweb.keystore ファイルを作成します
2. keystore.xml ファイルを作成してデプロイします
3. 必要なプロパティーを org.apache.karaf.management.cfg に追加します
4. Fuse コンテナーを再起動します

SSL アクセスで JMX を設定した後、接続をテストする必要があります。

警告

SSL/TLS セキュリティーを有効にする予定がある場合は、Poodle 脆弱性 (CVE-2014-3566) に対して保護するために、SSLv3 プロトコルを明示的に無効にする必要があります。詳細は、Disabling SSLv3 in JBoss Fuse 6.x and JBoss A-MQ 6.x を参照してください。



注記

Red Hat JBoss Fuse の実行中に SSL で JMX を設定した場合は、再起動する必要があります。

前提条件

以下を実行していない場合、実行する必要があります。

- **JAVA_HOME** 環境変数を設定します。
- **admin** ロールでの Karaf ユーザーの設定
  InstallDir/etc/users.properties ファイルを編集し、以下のエントリーを 1 行に追加します。

  ```
  admin=YourPassword,admin
  ```

  これにより、ユーザー名 **admin**、パスワード **YourPassword**、および **admin** ロールを持つ新規ユーザーが作成されます。

jbossweb.keystore ファイルを作成します

コマンドプロンプトを開き、現在の場所が Karaf インストールの **etc/** ディレクトリーであることを確認します。

```
cd etc
```

コマンドラインで、アプリケーションに適した **-dname** 値 (識別名) を使用して、以下のコマンドを入力します。

```
$JAVA_HOME/bin/keytool -genkey -v -alias jbossalias -keyalg RSA -keysize 1024 -keystore jbossweb.keystore -validity 3650 -keypass JbossPassword -storepass JbossPassword -dname "CN=127.0.0.1, OU=RedHat Software Unit, O=RedHat, L=Boston, S=Mass, C=USA"
```

重要

1 つのコマンドラインでコマンド全体を入力します。

このコマンドは、次のような出力を返します。

```
Generating 1,024 bit RSA key pair and self-signed certificate (SHA256withRSA) with a validity of 3,650 days
```

```
 for: CN=127.0.0.1, OU=RedHat Software Unit, O=RedHat, L=Boston, ST=Mass, C=USA
New certificate (self-signed):
[
[
  Version: V3
  Subject: CN=127.0.0.1, OU=RedHat Software Unit, O=RedHat, L=Boston, ST=Mass, C=USA
  Signature Algorithm: SHA256withRSA, OID = 1.2.840.113549.1.1.11

  Key:  Sun RSA public key, 1024 bits
  modulus:
112308602579056704360496299050191816946109837286427320179534244008039380 8

159410077607500864745991099141380637280072294767016640781490175445910072027 9046

394462181373817732403106426038265948319382617744876203043766931839107261986 7218
    036972335210839062722456085328301058362052369248473659880488338711351959835357
  public exponent: 65537
  Validity: [From: Thu Jun 05 12:19:52 EDT 2014,
              To: Sun Jun 02 12:19:52 EDT 2024]
  Issuer: CN=127.0.0.1, OU=RedHat Software Unit, O=RedHat, L=Boston, ST=Mass, C=USA
  SerialNumber: [    4666e4e6]

Certificate Extensions: 1
[1]: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: AC 44 A5 F2 E6 2F B2 5A   5F 88 FE 69 60 B4 27 7D  .D.../.Z_..i`.'.
0010: B9 81 23 9C                                        ..#.
]
]

]
  Algorithm: [SHA256withRSA]
  Signature:
0000: 01 1D 95 C0 F2 03 B0 FD   CF 3A 1A 14 F5 2E 04 E5  .........:......
0010: DD 18 DD 0E 24 60 00 54   35 AE FE 36 7B 38 69 4C  ....$`.T5..6.8iL
0020: 1E 85 0A AF AE 24 1B 40   62 C9 F4 E5 A9 02 CD D3  .....$.@b.......
0030: 91 57 60 F6 EF D6 A4 84   56 BA 5D 21 11 F7 EA 09  .W`.....V.]!....
0040: 73 D5 6B 48 4A A9 09 93   8C 05 58 91 6C D0 53 81  s.kHJ.....X.l.S.
0050: 39 D8 29 59 73 C4 61 BE   99 13 12 89 00 1C F8 38  9.)Ys.a........8
0060: E2 BF D5 3C 87 F6 3F FA   E1 75 69 DF 37 8E 37 B5  ...<..?..ui.7.7.
0070: B7 8D 10 CC 9E 70 E8 6D   C2 1A 90 FF 3C 91 84 50  .....p.m....<..P

]
[Storing jbossweb.keystore]
```

InstallDir/etc にファイル **jbossweb.keystore** が含まれるかどうかを確認します。

keystore.xml ファイルを作成してデプロイします

1. 好みの XML エディターを使用して、< **installDir> /jboss-fuse-7.13.0.fuse-7_13_0-00012-redhat-00001/etc ディレクトリーに keystore. xml** ファイルを作成して保存します。

2. このテキストをファイルに含めます。

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
```

```
xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0">
<jaas:keystore name="sample_keystore"
rank="1"
path="file:etc/jbossweb.keystore"
keystorePassword="JbossPassword"
keyPasswords="jbossalias=JbossPassword" />
</blueprint>
```

3. **keystore.xml** ファイルを **InstallDir/deploy** ディレクトリー (ホットデプロイディレクトリー) にコピーして、Karaf コンテナーにデプロイします。

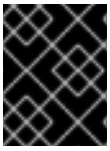
注記

その後、**keystore.xml** ファイルをアンデプロイする必要がある場合は、**Karaf コンテナーの実行中**に **deploy/** ディレクトリーから **keystore.xml** ファイルを削除して実行できます。

必要なプロパティーを org.apache.karaf.management.cfg に追加します

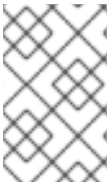
InstallDir/etc/org.apache.karaf.management.cfg ファイルを編集して、ファイルの最後に次のプロパティーを含めます。

```
secured = true
secureProtocol = TLSv1
keyAlias = jbossalias
keyStore = sample_keystore
trustStore = sample_keystore
```



重要

Poodle 脆弱性 (CVE-2014-3566) から保護するには、**secureProtocol** を **TLSv1** に設定する必要があります。



注記

必要に応じて、**enabledCipherSuites** プロパティーを設定して、JMX TLS 接続に使用する特定の暗号スイートをリストできます。このプロパティーを設定すると、デフォルトの暗号スイートがオーバーライドされます。

Karaf コンテナーを再起動します。

新しい JMX SSL/TLS 設定を有効にするには、Karaf コンテナーを再起動する必要があります。

セキュアな JMX 接続のテスト

1. コマンドプロンプトを開き、現在の場所が Fuse インストールの **etc/** ディレクトリーであることを確認します。

```
cd <installDir>/jboss-fuse-7.13.0.fuse-7_13_0-00012-redhat-00001/etc
```

2. ターミナルを開き、次のコマンドを入力して JConsole を起動します。

```
jconsole -J-Djavax.net.debug=ssl -J-Djavax.net.ssl.trustStore=jbossweb.keystore -J-
Djavax.net.ssl.trustStoreType=JKS -J-Djavax.net.ssl.trustStorePassword=JbossPassword
```

-J-Djavax.net.ssl.trustStore オプションは、**jbossweb.keystore** ファイルの場所を指定します (この場所を正しく指定しないと、SSL/TLS ハンドシェイクに失敗します)。**-J-Djavax.net.debug=ssl** 設定により SSL/TLS ハンドシェイクメッセージのロギングが有効になるため、SSL/TLS が正常に有効になっていることを確認できます。

重要

同じコマンドラインでコマンド全体を入力します。

3. JConsole が開いたら、**New Connection** ウィザードで **Remote Process** オプションを選択します。
4. **Remote Process** オプションで、**service:jmx:<protocol>:<sap>** 接続 URL に、次の値を入力します。

```
service:jmx:rmi://localhost:44444/jndi/rmi://localhost:1099/karaf-root
```

(**etc/users.properties** ファイルに設定されているように)**Username** および **Password** フィールドに有効な JAAS 認証情報を入力します。

```
Username: admin
Password: YourPassword
```

2.5. ELYTRON クレデンシャルストアの使用

Fuse には、JBoss EAP の一部である Elytron クレデンシャルストア機能が含まれています。クレデンシャルストアは、機密性の高いテキスト文字列をストレージファイルで暗号化することで、それらを安全に保護できます。各コンテナーには、必ず 1 つのクレデンシャルストアを含めることができます。

セキュアな設定では、パスワードの保存方法が一般的な問題です。たとえば、さまざまなアプリケーションからデータベースにアクセスするためのパスワードについて考えてみます。多くの認証方法では、パスワードをクリアテキストで使用できなければ、サーバーはデータベースサーバーにクレデンシャルを送信できません。通常、クリアテキストパスワードをテキスト設定ファイルに保存することは適切ではありません。

Elytron クレデンシャルストアはこの問題を解決します。パスワードやその他の機密性の高い値は、PKCS#12 仕様に準拠した暗号化ファイルであるクレデンシャルストアにセキュアに保存できます。クレデンシャルストアは、暗号化されていない値を保存しません。クレデンシャルストアは、PBE (Password Based Encryption) を使用して、パスワードなどの機密性の高い値とストア自体の両方を暗号化します。

詳細は以下のセクションを参照してください。

- 「クレデンシャルストアの使用」
- 「システムプロパティーがクレデンシャルストア設定を保持している場合の動作」
- 「クレデンシャルストアのシステムプロパティーと環境変数の説明」
- 「**credential-store:create** コマンドリファレンス」

- 「**credential-store:store** コマンドリファレンス」
- 「**credential-store:list** コマンドリファレンス」
- 「**credential-store:remove** コマンドリファレンス」
- 「クレデンシャルストアの使用を有効化する設定管理プロパティーの例」

2.5.1. クレデンシャルストアの使用

Fuse を実行している Apache Karaf コンテナーで、クレデンシャルストアを使用するには、クレデンシャルストアを作成および設定してから、それに値を追加します。Fuse は引き続き実行され、クレデンシャルストアは使用可能になります。

前提条件

- クレデンシャルストアを作成する際に、次のデフォルトを使用する。
  - PKCS#12 クレデンシャルストアを作成する。
  - **masked-SHA1-DES-EDE** アルゴリズムを適用してクレデンシャルストアを暗号化する。
  - アルゴリズムを 200000 回繰り返す。
  - **${karaf.etc}/credential.store.p12** でクレデンシャルストアを見つける。
- クレデンシャルストア設定を **${karaf.etc}/system.properties** に保存する。

この動作のいずれかを変更する必要がある場合は、**credential-store:create** コマンドの呼び出し に関する情報を参照してください。

手順

1. クレデンシャルストアのパスワードを選択します。
   後で、クレデンシャルストアに値を追加するとき、または値を復号化するとき、クレデンシャルストアコマンドはクレデンシャルストアのパスワードを使用して値を暗号化および復号化します。
2. **credential-store:create** コマンドを実行します。これにより、選択したクレデンシャルストアのパスワードを入力するよう要求されます。

```
karaf@root()> credential-store:create --persist
Credential store password: *****
Credential store password (repeat): *****

Credential store configuration was persisted in ${karaf.etc}/system.properties and is effective.

Credential store was written to /data/servers/fuse-karaf-7.4.0.fuse-
740060/etc/credential.store.p12

By default, only system properties are encrypted.
Encryption of configuration admin properties can be enabled by setting
felix.cm.pm=elytron in etc/config.properties.
```

   このコマンドは、**etc/system.properties** に以下のような設定を書き込みます。

```
credential.store.location = /data/servers/fuse-karaf-7.4.0.fuse-740060/etc/credential.store.p12
credential.store.protection.algorithm = masked-SHA1-DES-EDE
credential.store.protection.params =
MDkEKFJId25PaXlVQldKUWw5R2tLclhZQndpTGhhVXJsWG5lNVJMbTFCZEMCAwMNQAQI
0Whepb7H1BA=
credential.store.protection = m+1BcfRyCnI=
```

3. 以下のように **credential-store:store** コマンドを実行し、暗号化された値をクレデンシャルストアに追加します。
credential-store:store alias

alias を一意の鍵値に置き換えます。ツールは、後でクレデンシャルストアに追加する暗号化された値を取得するために、このエイリアスを使用します。たとえば、コードで **db.password** システムプロパティーを使用し、**etc/system.properties** ファイルに **db.password** プロパティーをデータベースの実際のパスワードに設定するエントリーがあるとします。システムプロパティー **db.password** をエイリアスとして指定することが推奨されます。

このコマンドを呼び出すと、クレデンシャルストアに追加する機密値を入力して確認するように求められます。プロンプトが表示されたら、**db.password** エイリアスの例を続行し、データベースの実際のパスワードを入力します。

```
karaf@root()> credential-store:store db.password
Secret value to store: ******
Secret value to store (repeat): ******
Value stored in the credential store. To reference it use: CS:db.password
```

4. **etc/system.properties** ファイルのエントリーを更新するか、新しいエントリーを追加します。更新または追加するエントリーは、**credential-store:store** コマンドで指定したエイリアスを、コマンドが出力する参照値に設定します。以下に例を示します。

```
db.password = CS:db.password
```

Fuse が設定されたクレデンシャルストアで稼働している場合、**db.password** システムプロパティーなどの各インスタンスを、クレデンシャルストアにある実際のシークレット値に動的に置き換えます。

5. **credential-store:store** コマンドでは、指定したエイリアスがすでに使用されているシステムプロパティーである場合は、次のステップをスキップします。コードでシークレットに指定したエイリアスをまだ使用していない場合は、シークレットを必要とする各ファイルで、前の手順でシステムプロパティーとして追加したエイリアスを指定します。たとえば、コードは **db.password** を参照します。

6. クレデンシャルストアに追加する値ごとに、前の 3 つの手順を繰り返します。

結果

クレデンシャルストアを使用する準備が整いました。Fuse が起動するか、クレデンシャルストアバンドルが再起動すると、システムプロパティーを処理し、クレデンシャルストアエントリーへの参照を検索します。実行するシステムプロパティーごとに、Fuse はクレデンシャルストアから関連する値を取得し、システムプロパティーを実際のシークレット値に置き換えます。実際のシークレット値は、そのシステムプロパティーのインスタンスを含むすべてのコンポーネント、バンドル、およびコードで使用できます。

2.5.2. システムプロパティーがクレデンシャルストア設定を保持している場合の動作

クレデンシャルストアが使用されており、システムプロパティーを使用してその設定パラメーターを保持しているとします。Fuse が起動すると、すべてのシステムプロパティーが処理されます。Fuse は、**CS:** 接頭辞を持つ値に設定されたシステムプロパティーをクレデンシャルストアにある関連する値に置き換えます。Fuse は **java.lang:type=Runtime** JMX MBean をプロキシーし、JMX **getSystemProperties()** メソッドの呼び出しごとに復号化された値を非表示にします。

たとえば、クレデンシャルストアに 1 つのエントリーがあるとします。

```
karaf@root()> credential-store:list --show-secrets
Alias       | Reference      | Secret value
------------+----------------+-------------
db.password | CS:db.password | sec4et
```

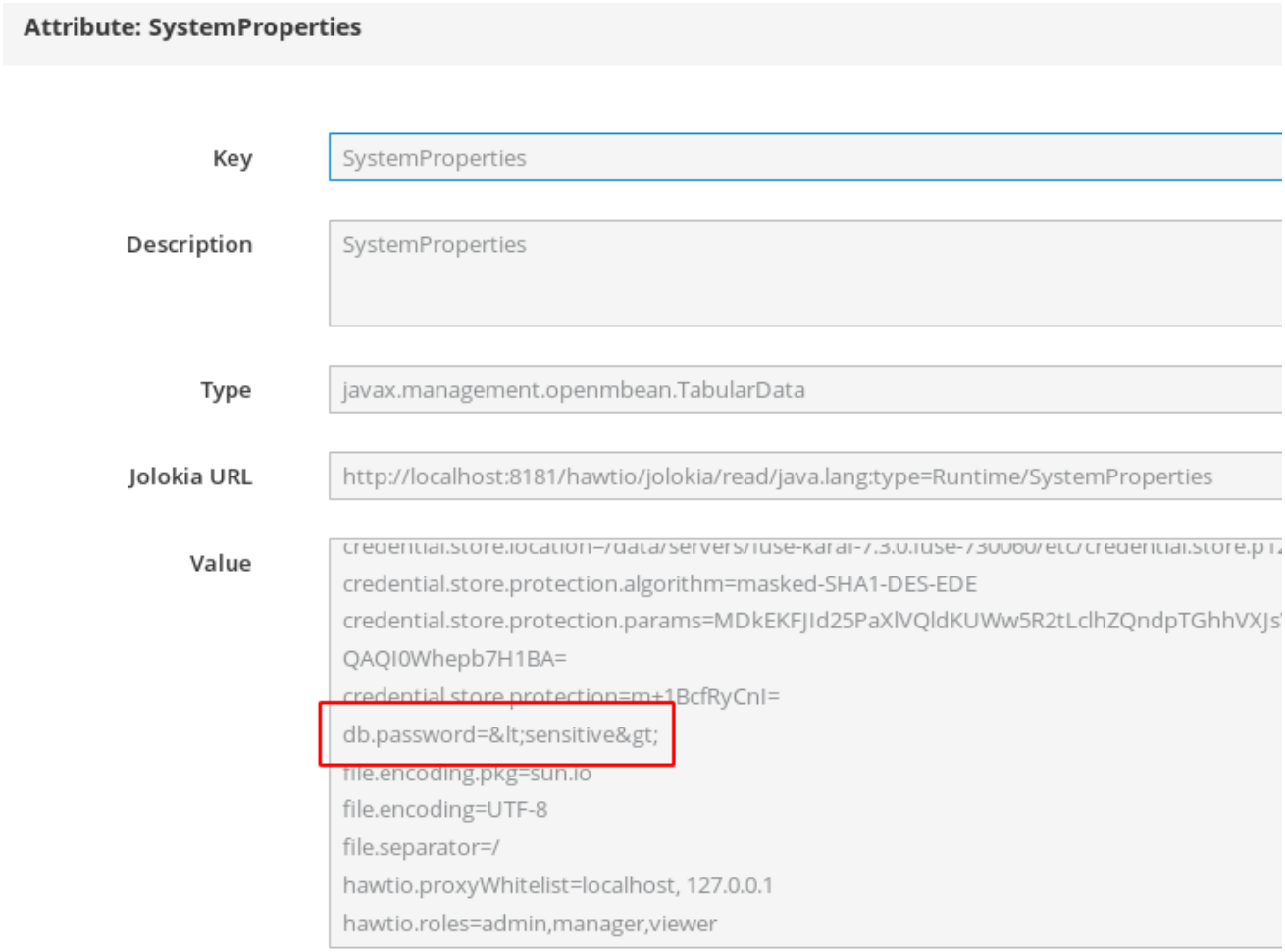
このエントリーをクレデンシャルストアに追加した後に、**etc/system.properties** ファイルを編集し、次のエントリーを追加したとします。

db.password = CS:db.password

Fuse の起動時に、または **org.jboss.fuse.modules.fuse-credential-store-core** バンドルを再起動すると、Fuse は **db.password** システムプロパティーへの参照をチェックします。各参照で、Fuse は **CS:db.password** エイリアスを使用してクレデンシャルストアから関連する値を取得します。これを確認するには、次のコマンドを呼び出します。

```
karaf@root()> system:property db.password
sec4et
```

ただし、JMX を使用してこれを確認すると、クレデンシャルストアの値は非表示になります。



2.5.3. クレデンシャルストアのシステムプロパティーと環境変数の説明

システムプロパティーまたは環境変数を使用して、クレデンシャルストアの設定パラメーターを保持できます。クレデンシャルストアを作成するときに指定するオプションによって、次のことが決まります。

- プロパティーまたは変数を自分で設定する必要があるかどうか。
- プロパティーまたは変数に設定されている、または設定する必要がある正確な値。

プロパティー/変数を理解すると、クレデンシャルストアがどのように機能するかを理解するのに役立ちます。

credential-store:create コマンドを実行し、**--persist** オプションのみを指定すると、コマンドはシステムプロパティーをクレデンシャルストア設定パラメーターに設定します。クレデンシャルストアのシステムプロパティーは明示的に設定する必要はありません。

代わりにクレデンシャルストアの環境変数を使用するか、**credential-store:create** コマンドのデフォルト動作を変更する場合は、クレデンシャルストアの作成時に指定できるオプションについて詳述した **credential-store:create** コマンドリファレンス を参照してください。

クレデンシャルストアを作成するコマンドを呼び出すと、指定したオプションによってクレデンシャルストアのプロパティーまたは変数の設定が決まります。プロパティーまたは変数を独自に設定する必要がある場合、**credential-store:create** コマンドの出力には、その手順が含まれます。つまり、クレデンシャルストアのシステムプロパティーまたは環境変数の設定を決定するのはユーザーではありません。**credential-store:create** コマンドを実行すると常に設定が決定されます。

次の表は、クレデンシャルストアのプロパティーと変数について説明しています。特定のパラメーターについて、環境変数とシステムプロパティーの両方が設定されている場合、環境変数の設定が優先されます。

Name	説明
環境変数: **CREDENTIAL_STORE_PROTECION_ALGORITHM** システムプロパティー: **credential.store.protection.algorithm**	クレデンシャルストアコマンドが暗号化キーを取得するために使用する Password Based Encryption (PBE) アルゴリズム。
環境変数: **CREDENTIAL_STORE_LOCATION** システムプロパティー: **credential.store.location**	クレデンシャルストアの場所。
環境変数: **CREDENTIAL_STORE_PROTECTION_PARAMS** システムプロパティー: **credential.store.protection.params**	クレデンシャルストアが暗号化キーを取得するために使用するパラメーター。パラメーターには、反復回数、初期ベクトル、およびソルトが含まれます。

Name	説明
環境変数: **CREDENTIAL_STORE_PROTECTION** システムプロパティー: **credential.store.protection**	クレデンシャルストアコマンドが、パスワードまたはその他のセキュアなデータをクレデンシャルストアから回復するために復号化する必要があるパスワード。**credential-store:create** コマンドを実行すると、このコマンドによりパスワードの指定が求められます。そのパスワードの暗号化は、この環境変数またはシステムプロパティーの設定です。

2.5.4. credential-store:create コマンドリファレンス

クレデンシャルストアを作成および設定するには、以下の形式の **credential-store:create** コマンドを実行します。

credential-store:create [options]

オプションを指定しない場合、コマンドは以下を行います。

- 選択したクレデンシャルストアのパスワードの入力を求める。
- PKCS#12 クレデンシャルストアを作成する。
- **masked-SHA1-DES-EDE** アルゴリズムを使用してクレデンシャルストアを暗号化します。
- アルゴリズムを 200000 回繰り返す。
- **${karaf.etc}/credential.store.p12** でクレデンシャルストアを見つけます。
- クレデンシャルストア設定を保存しない。

次の表は、デフォルトの動作を変更するために指定できるオプションを示しています。

オプション	説明
-w または **--password-property**	このオプションの後に、マスターパスワードの値に設定されている環境変数またはシステムプロパティーを指定します。クレデンシャルストアは、この値をアルゴリズムと組み合わせて使用して、暗号化キーまたは復号化キーを取得します。 **-w** または **-W** オプションを指定しない場合は、コマンドの呼び出し後に、マスターパスワードの入力と確認が求められます。 例: **-w MASTER_PW**

オプション	説明
-W または **--password**	このオプションの後に、選択したマスターパスワードのプレインテキスト値を指定します。マスターパスワードのプレインテキスト値は履歴に表示されます。 クレデンシャルストアは、この値をアルゴリズムと組み合わせて使用して、暗号化キーまたは復号化キーを取得します。 **-w** または **-W** オプションを指定しない場合は、コマンドの呼び出し後に、マスターパスワードの入力と確認が求められます。 例: **-W "M@s!erP#"**
-f または **--force**	クレデンシャルストアの作成を強制します。新しいクレデンシャルストアを作成する場所に既存のクレデンシャルストアが存在する場合、このオプションを指定すると、コマンドは既存のクレデンシャルストアを上書きします。既存のクレデンシャルストアのコンテンツはすべて失われます。 デフォルトの動作では、目的の場所にクレデンシャルストアがすでに存在する場合、コマンドはクレデンシャルストアを作成しません。
-l または **--location**	新しいクレデンシャルストアの場所を指定します。デフォルトの場所である **${karaf.etc}/credential.store.p12** を使用することが推奨されます。
-ic または **--iteration-count**	このオプションの後に、使用されている暗号化アルゴリズムを繰り返し適用する回数が整数で示されます。反復するごとに、前の結果を取得して、再度アルゴリズムを適用します。その結果が最終的なマスクされたパスワードです。デフォルトは 200000 です。
-a または **--algorithm**	このオプションの後に、マスクされたパスワードを生成するために **credential-store:create** コマンドで使用されるアルゴリズムの識別子を指定します。デフォルトの **masked-SHA1-DES-EDE** を使用することが推奨されます。
-p または **--persist**	新しいクレデンシャルストアの設定を **${karaf.etc}/system.properties** に保存します。このオプションを指定しないと、**credential-store:create** コマンドは次に行う手順が含まれる設定情報をコンソールに送信します。この表の後の例を参照してください。 クレデンシャルストア設定パラメーター値を確認する場合、このオプションを省略します。または、**etc/system.properties** ファイルを使用せずにクレデンシャルストア設定パラメーターをアプリケーションに渡す予定であるため、このオプションを省略することもできます。
--help	コマンド構文とオプションに関する情報を表示します。

--persist を指定しない場合のクレデンシャルストアの作成例

以下のコマンドはクレデンシャルストアを作成しますが、クレデンシャルストアの設定は **${karaf.etc}/system.properties** に保存されません。このコマンドは、デフォルトの **masked-SHA1-DES-EDE** アルゴリズムを使用します。

```
karaf@root()> credential-store:create
Credential store password: *****
Credential store password (repeat): *****

Credential store was written to /data/servers/fuse-karaf-7.4.0.fuse-740060/etc/credential.store.p12

By default, only system properties are encrypted.
Encryption of configuration admin properties can be enabled by
setting felix.cm.pm=elytron in etc/config.properties.

Credential store configuration was not persisted and is not
effective. Please use one of the following configuration options and restart Fuse.
Option #1: Configure these system properties (e.g., in etc/system.properties):
 - credential.store.protection.algorithm=masked-SHA1-DES-EDE
 -
credential.store.protection.params=MDkEKGdOSkpRWXpndjhkVVZYbHF4elVpbUszNW0wc3NXczhNS
1A5cVlhZzcCAwMNQAQIDPzQ+BDGwX4=
 - credential.store.protection=0qudlx1XZFM=
 - credential.store.location=/data/servers/fuse-karaf-7.4.0.fuse-740060/etc/credential.store.p12
Option #2: Configure these environmental variables (e.g., in bin/setenv):
 - CREDENTIAL_STORE_PROTECTION_ALGORITHM=masked-SHA1-DES-EDE
 -
CREDENTIAL_STORE_PROTECTION_PARAMS=MDkEKGdOSkpRWXpndjhkVVZYbHF4elVpbUszN
W0wc3NXczhNS1A5cVlhZzcCAwMNQAQIDPzQ+BDGwX4=
 - CREDENTIAL_STORE_PROTECTION=0qudlx1XZFM=
 - CREDENTIAL_STORE_LOCATION=/data/servers/fuse-karaf-7.4.0.fuse-
740060/etc/credential.store.p12
```

2.5.5. credential-store:store コマンドリファレンス

暗号化された値をクレデンシャルストアに追加するには、以下の形式の **credential-store:store** コマンドを実行します。

credential-store:store alias [secret]

alias を一意の鍵値に置き換えます。ツールは、クレデンシャルストアに追加する暗号化された値を取得するために、このエイリアスを使用します。

必要に応じて、**secret** を暗号化してクレデンシャルストアに追加する値に置き換えます。通常、これはパスワードですが、暗号化する任意の値を指定できます。

コマンドラインで **secret** を指定すると、プレーンテキストの値が履歴に表示されます。コマンドラインで **secret** を指定しないと、コマンドによりその入力を求められ、値は履歴に表示されません。

コマンドに関する情報を表示するには、次のように入力します。

credential-store:store --help。

次のコマンドラインは、クレデンシャルストアにエントリーを追加する例です。

```
karaf@root()> credential-store:store db.password sec4et
Value stored in the credential store. To reference it use: CS:db.password
```

クレデンシャルストアには、**CS:db.password** を指定して参照できるエントリーが含まれるになります。

2.5.6. credential-store:list コマンドリファレンス

クレデンシャルストアのエントリーのエイリアスを取得するには、**credential-store:list** コマンドを実行します。これにより、クレデンシャルストアのすべてのエントリーのリストが表示されます。以下に例を示します。

```
karaf@root()> credential-store:list
Alias       │ Reference
────────────┼──────────────────────
db.password  │ CS:db.password
db2.password | CS:db2.password
```

クレデンシャルストアで暗号化されているシークレット値の復号化リストも表示するには、次のようにコマンドを呼び出します。

```
karaf@root()> credential-store:list --show-secrets
Alias       │ Reference       │ Secret value
────────────┼─────────────────┼──────────────
db.password  │ CS:db.password  │ sec4et
db2.password | CS:db2.password | t0pSec4et
```

コマンドに関する情報を表示するには、次のように入力します。

```
karaf@root()> credential-store:list --help
```

2.5.7. credential-store:remove コマンドリファレンス

クレデンシャルストアからエントリーを削除するには、以下の形式の **credential-store:remove** コマンドを実行します。

credential-store:remove alias

alias は、エントリーをクレデンシャルストアに追加したときに **alias** 引数に指定した一意の鍵値に置き換えます。**CS:** 接頭辞は指定しないでください。**credential-store:list** コマンドを実行してエイリアスを取得できます。

credential-store:remove コマンドは、指定したエイリアスを持つエントリーのクレデンシャルストアをチェックし、見つかった場合は削除します。以下に例を示します。

```
karaf@root()> credential-store:remove db.password

Alias       │ Reference       │ Secret value
────────────┼─────────────────┼──────────────
db2.password | CS:db2.password | t0pSec4et
```

コマンドに関する情報を表示するには、次のように入力します。

karaf@root()> credential-store:remove --help

2.5.8. クレデンシャルストアの使用を有効化する設定管理プロパティーの例

開発環境では、設定管理サービスプロパティーを使用して、クレデンシャルストアの使用を有効にできます。Configuration Admin プロパティーは **etc/*.cfg** ファイルで定義されます。

重要

クレデンシャルストアの使用を有効化する設定管理プロパティーは、テクノロジープレビュー機能でのみ使用できます。テクノロジープレビュー機能は、Red Hat 製品サポートのサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではない場合があります。Red Hat は、実稼働環境でこれらを使用することを推奨していません。これらの機能により、近日発表予定の製品機能をリリースに先駆けてご提供でき、お客様は開発プロセス時に機能をテストして、フィードバックをお寄せいただくことができます。Red Hat のテクノロジープレビュー機能のサポート範囲に関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

準備

- **credential-store:create** コマンドを呼び出してクレデンシャルストアを作成します。**credential-store:create command reference** を参照してください。
- Configuration Admin プロパティーの使用を有効にするには、**etc/config.properties** ファイルを編集して、**felix.cm.pm = elytron** が含まれる行のコメントを解除します。

```
# When uncommented, configuration properties handled by Configuration Admin service will be encrypted when storing
# in etc/ and in bundle data. Values of the properties will actually be aliases to credential store entries.
# Please consult the documentation for more details.
felix.cm.pm = elytron
```

Fuse 起動時の動作

1. **felix.configadmin** バンドル:
   - **felix.cm.pm** プロパティーが設定されているため、**ConfigurationAdmin** サービスを登録する遅延。
   - **name=cm** OSGi サービス登録プロパティーで **org.apache.felix.cm.PersistenceManagerOSGi** サービスの可用性を待ちます。
2. Fuse クレデンシャルストアバンドル:
   a. **credential.store.*** システムプロパティーまたは **CREDENTIAL_STORE_*** 環境変数に設定した値を使用して、クレデンシャルストアをロードします。
   b. **org.apache.felix.cm.PersistenceManagerOSGi** サービスを実装する OSGi サービスを登録します。

何らかの障害が発生した場合、クレデンシャルストアバンドルは **PersistenceManager** サービスを登録しますが、特別なことはしません。何かが破損しているか、クレデンシャルストアが利用できない場合、Fuse は暗号化されていない設定値を読み取ることができるはずです。**CS:** 接頭辞で指定した暗号化された値は、元の値を覚えているか、クレデンシャルストアとその設定を復旧できる場合以外は失われます。

3. **felix.configadmin** プロセスは、新しい永続マネージャーサービスを使用してクレデンシャルストア設定をロードおよび保存します。

例

クレデンシャルストアに 2 つのエントリーがあるとします。

```
karaf@root()> credential-store:list --show-secrets
Alias       │ Reference      │ Secret value
────────────┼────────────────┼─────────────
db.password │ CS:db.password │ sec4et
http.port   │ CS:http.port   │ 8182
```

設定管理サービス設定では、実際の値ではなく、機密性の高い値のエイリアスを使用することを選択します。たとえば、Web 設定プロパティーを次のように変更します。

```
karaf@root()> config:property-list --pid org.ops4j.pax.web
  javax.servlet.context.tempdir = /data/servers/fuse-karaf-7.4.0.fuse-740060/data/pax-web-jsp
  org.ops4j.pax.web.config.file = /data/servers/fuse-karaf-7.4.0.fuse-740060/etc/undertow.xml
  org.ops4j.pax.web.session.cookie.httpOnly = true
  org.osgi.service.http.port = 8181

karaf@root()> config:property-set --pid org.ops4j.pax.web org.osgi.service.http.port CS:http.port

karaf@root()> config:property-list --pid org.ops4j.pax.web
  javax.servlet.context.tempdir = /data/servers/fuse-karaf-7.4.0.fuse-740060/data/pax-web-jsp
  org.ops4j.pax.web.config.file = /data/servers/fuse-karaf-7.4.0.fuse-740060/etc/undertow.xml
  org.ops4j.pax.web.session.cookie.httpOnly = true
  org.osgi.service.http.port = CS:http.port
```

ログでは、以下の行の最後にあるように、実際の値 **8182** が表示されます。ログに実際のテキスト値が表示されるかどうかは、暗号化された値を使用するコンポーネントによって決まります。

```
2019-03-12 15:36:25,648 INFO  {paxweb-config-2-thread-1} (ServerControllerImpl.java:458) :
Starting undertow http listener on 0.0.0.0:8182
```

上記のコマンドでは、このプロパティーには数値がありますが、2 番目の **config:property-list --pid org.ops4j.pax.web** コマンドは、**8182** ではなく **CS:http.port** を表示します。**pax-web-undertow** プロセスはこのポートで開始します。これは、OSGi フックにより、**config:property-list --pid org.ops4j.pax.web** コマンドの出力を表示する **felix.fileinstall** プロセスが、復号化された (逆参照) 値を認識できないようにするためです。これは、**etc/org.ops4j.pax.web.cfg** ファイルが復号化された (逆参照) 値を格納せず、代わりに格納する理由でもあります。次に例を示します。

```
org.osgi.service.http.port = CS:http.port

org.ops4j.pax.web.config.file = ${karaf.etc}/undertow.xml
org.ops4j.pax.web.session.cookie.httpOnly = true

javax.servlet.context.tempdir = ${karaf.data}/pax-web-jsp
```

第3章 UNDERTOW HTTP サーバーのセキュリティー保護

概要

etc/undertow.xml 設定ファイルの内容を編集して、組み込み Undertow HTTP サーバーが SSL/TLS セキュリティーを使用するように設定できます。特に、この方法で Fuse Console に SSL/TLS セキュリティーを追加できます。

3.1. UNDERTOW サーバー

Fuse コンテナーは、汎用 HTTP サーバーおよび HTTP サーブレットコンテナーとして機能する Undertow サーバーで事前設定されています。1 つの HTTP ポート (デフォルトでは **http://localhost:8181**) により、Undertow コンテナーは複数のサービスをホストできます。以下に例を示します。

- Fuse コンソール (デフォルトでは、**http://localhost:8181/hawtio**)
- Apache CXF Web サービスエンドポイント (ホストとポートがエンドポイント設定で未指定である場合)
- 一部の Apache Camel エンドポイント

すべての HTTP エンドポイントでデフォルトの Undertow サーバーを使用する場合、ここで説明する手順に従って、これらの HTTP エンドポイントに SSL/TLS セキュリティーを簡単に追加できます。

3.2. X.509 証明書と秘密鍵の作成

Undertow サーバーで SSL/TLS を有効にする前に、必ず Java キーストア形式 (JKS 形式) で、X.509 証明書と秘密鍵を作成する必要があります。署名付き証明書と秘密鍵の作成方法について、詳しくは 付録 A *証明書の管理* を参照してください。

3.3. APACHE KARAF コンテナーで UNDERTOW の SSL/TLS を有効化

以下の手順では、キーストアパスワード **StorePass** およびキーパスワード **KeyPass** で、署名済みの X.509 証明書と秘密鍵のペアがキーストアファイル **alice.ks** で作成済みであることを前提とします。

Karaf コンテナーで Undertow の SSL/TLS を有効にするには、以下を実行します。

1. Pax Web サーバーが、**etc/undertow.xml** ファイルから設定を取得するように設定されていることを確認します。**etc/org.ops4j.pax.web.cfg** ファイルの内容を確認すると、以下の設定があるはずです。

   ```
   org.ops4j.pax.web.config.file=${karaf.etc}/undertow.xml
   ```

2. テキストエディターで **etc/org.ops4j.pax.web.cfg** ファイルを開き、以下の行を追加します。

   ```
   org.osgi.service.http.port.secure=8443
   org.osgi.service.http.secure.enabled=true
   ```

   etc/org.ops4j.pax.web.cfg ファイルを保存して閉じます。

3. テキストエディターで **etc/undertow.xml** ファイルを開きます。次の手順では、インストール時以降、変更されていないデフォルトの **undertow.xml** ファイルで作業することを前提としています。

4. XML 要素 **http-listener** および **https-listener** を検索します。**http-listener** 要素をコメントアウトし (**<!--** および **-->** で囲み)、**https-listener** 要素をアンコメントします (2 行使用)。編集された XML のフラグメントは、次のようになります。

```
<!-- HTTP(S) Listener references Socket Binding (and indirectly - Interfaces) -->
<!-- http-listener name="http" socket-binding="http" /> -->
verify-client: org.xnio.SslClientAuthMode.NOT_REQUESTED,
org.xnio.SslClientAuthMode.REQUESTED, org.xnio.SslClientAuthMode.REQUIRED
<!--<https-listener name="https"
            socket-binding="https"
            security-realm="https" verify-client="NOT_REQUESTED"
            enabled="true" />
-->
<https-listener name="https"
            socket-binding="https"
            worker="default"
            buffer-pool="default"
            enabled="true"

            receive-buffer="65536"
            send-buffer="65536"
            tcp-backlog="128"
            tcp-keep-alive="false"
            read-timeout="-1"
            write-timeout="-1"
            max-connections="1000000"

            resolve-peer-address="false"
            disallowed-methods="TRACE OPTIONS"
            secure="true"

            max-post-size="10485760"
            buffer-pipelined-data="false"
            max-header-size="1048576"
            max-parameters="1000"
            max-headers="200"
            max-cookies="200"
            allow-encoded-slash="false"
            decode-url="true"
            url-charset="UTF-8"
            always-set-keep-alive="true"
            max-buffered-request-size="16384"
            record-request-start-time="true"
            allow-equals-in-cookie-value="false"
            no-request-timeout="60000"
            request-parse-timeout="60000"
            rfc6265-cookie-validation="false"
            allow-unescaped-characters-in-url="false"

            certificate-forwarding="false"
            proxy-address-forwarding="false"
```

```
            enable-http2="false"
            http2-enable-push="false"
            http2-header-table-size="4096"
            http2-initial-window-size="65535"
            http2-max-concurrent-streams="-1"
            http2-max-frame-size="16384"
            http2-max-header-list-size="-1"

            require-host-http11="false"
            proxy-protocol="false"

            security-realm="https"
            verify-client="NOT_REQUESTED"
            enabled-cipher-suites="TLS_AES_256_GCM_SHA384"
            enabled-protocols="TLSv1.3"
            ssl-session-cache-size="0"
            ssl-session-timeout="0"
/>
```

5. **w:keystore** 要素を検索します。デフォルトでは、**w:keystore** 要素は以下のように設定されます。

```
<w:keystore path="${karaf.etc}/certs/server.keystore" provider="JKS" alias="server"
        keystore-password="secret" key-password="secret"
        generate-self-signed-certificate-host="localhost" />
```

Undertow サーバーの証明書として **alice** 証明書をインストールするには、以下のように **w:keystore** 要素属性を変更します。

- **path** を、ファイルシステムにおける **alice.ks** ファイルの場所 (絶対パス) に設定します。
- **provider** を **JKS** に設定します。
- **alias** をキーストアの **alice** 証明書エイリアスに設定します。
- **keystore-password** を、キーストアをアンロックするパスワードの値に設定します。
- **key-password** を、**alice** 秘密鍵を暗号化するパスワードの値に設定します。
- **generate-self-signed-certificate-host** 属性設定を削除します。

6. たとえば、**alice.ks** キーストアをインストールした後に、変更された **w:keystore** 要素は以下のようになります。

```
<w:keystore path="${karaf.etc}/certs/alice.ks" provider="JKS" alias="alice"
        keystore-password="StorePass" key-password="KeyPass" />
```

7. セキュアな HTTPS ポートがバインドする IP アドレスを指定するために使用される **<interface name="secure">** タグを検索します。デフォルトでは、この要素は次のようにコメントアウトされます。

```
<!--<interface name="secure">-->
   <!--<w:inet-address value="127.0.0.1" />-->
<!--</interface>-->
```

要素のコメントを解除し、**value** 属性をカスタマイズして、HTTPS ポートをバインドする IP アドレスを指定します。たとえば、ワイルドカード値 **0.0.0.0** は、利用可能なすべての IP アドレスにバインドするように HTTPS を設定します。

```
<interface name="secure">
    <w:inet-address value="0.0.0.0" />
</interface>
```

8. **<socket-binding name="https"** タグを検索し、コメント解除します。このタグのコメントを解除すると、次のようになります。

```
<socket-binding name="https" interface="secure" port="${org.osgi.service.http.port.secure}"
/>
```

9. **etc/undertow.xml** ファイルを保存して閉じます。
10. 設定の変更を有効にするために、Fuse コンテナーを再起動します。

3.4. 許可された TLS プロトコルと暗号化スイートのカスタマイズ

許可される TLS プロトコルおよび暗号スイートをカスタマイズするには、**etc/undertow.xml** ファイルの **w:engine** 要素の以下の属性を変更します。

enabled-cipher-suites

許可される TLS/SSL 暗号化スイートのリストを指定します。

enabled-protocols

許可される TLS/SSL プロトコルのリストを指定します。

警告

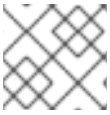
SSL プロトコルバージョンは攻撃に対して脆弱であるため、有効にしないでください。**TLS** プロトコルバージョンのみを使用します。

使用可能なプロトコルと暗号化スイートの詳細については、適切な JVM ドキュメントとセキュリティープロバイダーのドキュメントを参照してください。たとえば Java 8 の場合は、Java Cryptography Architecture Oracle Providers Documentation for JDK 8 を参照してください。

3.5. セキュアなコンソールへの接続

Pax Web 設定ファイルで Undertow サーバーの SSL セキュリティーを設定すると、次の URL を参照して Fuse コンソールを開くことができるはずです。

```
https://localhost:8443/hawtio
```



注記

この URL では、**http:** ではなく、**https:** スキームを入力するようにしてください。

最初に、ブラウザーは、信頼できない証明書を使用していることを警告します。この警告をスキップすると、Fuse Console のログイン画面が表示されます。

3.6. 高度な UNDERTOW 設定

3.6.1. IO 設定

PAXWEB-1255 以降、リスナーによって使用される XNIO ワーカーおよびバッファープールの設定を変更できます。undertow.xml テンプレートには、一部の IO 関連のパラメーターのデフォルト値を指定するセクションがあります。

```
<!-- Only "default" worker and buffer-pool are supported and can be used to override the default
values used by all listeners
    buffer-pool:
     - buffer-size defaults to:
        - when < 64MB of Xmx: 512
        - when < 128MB of Xmx: 1024
        - when >= 128MB of Xmx: 16K - 20
     - direct-buffers defaults to:
        - when < 64MB of Xmx: false
        - when >= 64MB of Xmx: true
    worker:
     - io-threads defaults to Math.max(Runtime.getRuntime().availableProcessors(), 2);
     - task-core-threads and task-max-threads default to io-threads * 8
-->

<!--
<subsystem xmlns="urn:jboss:domain:io:3.0">
    <buffer-pool name="default" buffer-size="16364" direct-buffers="true" />
    <worker name="default" io-threads="8" task-core-threads="64" task-max-threads="64" task-
keepalive="60000" />
</subsystem>
-->
```

以下の **buffer-pool** パラメーターを指定できます。

buffer-size

IO 操作に使用されるバッファーのサイズを指定します。指定のない場合は、利用可能なメモリーに応じてサイズが計算されます。

direct-buffers

java.nio.ByteBuffer#allocateDirect または java.nio.ByteBuffer#allocate を使用するかどうかを決定します。

以下の **worker** パラメーターを指定できます。

io-threads

ワーカーに作成する I/O スレッドの数。指定のない場合は、スレッドの数が CPU の数の 2 倍に設定されます。

task-core-threads

コアタスクスレッドプールのスレッド数。

task-max-threads

ワーカータスクスレッドプールの最大スレッド数。指定のない場合は、最大スレッドの数が CPU の数の 16 倍に設定されます。

3.6.2. ワーカー IO 設定

Undertow スレッドプールおよびその名前はサービスごとまたはバンドルベースで設定できるので、Hawtio コンソールからの監視やデバッグの効率が向上します。

バンドル Blueprint 設定ファイル (通常は Maven プロジェクトの **src/main/resources/OSGI-INF/blueprint** ディレクトリーに保存される) では、以下の例のように workerIOName および ThreadPool を設定できます。

例3.1 workerIOName および ThreadPool 設定の httpu:engine-factory 要素

```
<httpu:engine-factory>
    <httpu:engine port="9001">
        <httpu:threadingParameters minThreads="99" maxThreads="777" workerIOThreads="8"
workerIOName="WorkerIOTest"/>
    </httpu:engine>
</httpu:engine-factory>
```

以下の **threadingParameters** を指定できます。

minThreads

ワーカータスクスレッドプールのコアスレッドの数を指定します。通常、これは CPU コアごとに少なくとも 10 個必要です。

maxThreads

ワーカータスクスレッドプールの最大スレッド数を指定します。

以下の **worker** パラメーターを指定できます。

workerIOThreads

ワーカーに作成する I/O スレッドの数を指定します。指定されていない場合は、デフォルト値が選択されます。デフォルトとして妥当な設定は、CPU コアごとに 1 つの IO スレッドです。

workerIOName

ワーカーの名前を指定します。指定されていない場合には、デフォルトの "XNIO-1" が選択されます。

第4章 CAMEL ACTIVEMQ コンポーネントのセキュリティー保護

概要

Camel ActiveMQ コンポーネントを使用すると、Apache ActiveMQ ブローカーに接続できるルートに JMS エンドポイントを定義できます。Camel ActiveMQ エンドポイントをセキュリティー保護するには、**セキュアな** 接続ファクトリーを使用する Camel ActiveMQ コンポーネントのインスタンスを作成する必要があります。

4.1. セキュアな ACTIVEMQ 接続ファクトリー

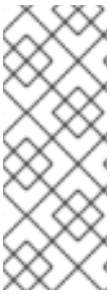
概要

Apache Camel は、ルート内の Apache ActiveMQ エンドポイントを定義するための Apache ActiveMQ コンポーネントを提供します。Apache ActiveMQ エンドポイントは事実上ブローカーの Java クライアントであり、コンシューマーエンドポイント (通常は JMS メッセージを **ポーリング** するルートの開始時に使用) またはプロデューサーエンドポイント (通常は JMS メッセージをブローカーに **送信** するためのルートの最後または途中で使用) を定義できます。

リモートブローカーがセキュアな場合 (SSL セキュリティー、JAAS セキュリティー、またはその両方)、Apache ActiveMQ コンポーネントを必要なクライアントセキュリティーを設定する必要があります。

セキュリティープロパティーのプログラミング

Apache ActiveMQ では、**ActiveMQSslConnectionFactory** JMS 接続ファクトリーのインスタンスを作成および設定することで、SSL セキュリティー設定 (および JAAS セキュリティー設定) をプログラムできます。JMS 接続ファクトリーのプログラミングは、OSGi、J2EE、Tomcat などのコンテナーのコンテキストで使用する正しいアプローチです。なぜなら、これらの設定は JMS 接続ファクトリーインスタンスを使用するアプリケーションに対してローカルだからです。



注記

スタンドアロンブローカーは、**Java システムプロパティー** を使用して SSL を設定できます。ただし、コンテナーにデプロイされたクライアントの場合、この設定は OSGi コンテナー全体ではなく、個々のバンドルにのみ適用する必要があるため、これは実用的なアプローチでは **ありません**。Camel ActiveMQ エンドポイントは事実上 Apache ActiveMQ Java クライアントの一種であるため、この制限は Camel ActiveMQ エンドポイントにも適用されます。

セキュアな接続ファクトリーの定義

例4.1「セキュアな接続ファクトリー Bean の定義」 では、ブループリントでセキュアな接続ファクトリー Bean を作成し、SSL/TLS セキュリティー **および** JAAS 認証の両方を有効にする方法を示しています。

例4.1 セキュアな接続ファクトリー Bean の定義

```
<bean id="jmsConnectionFactory"
      class="org.apache.activemq.ActiveMQSslConnectionFactory">
  <property name="brokerURL" value="ssl://localhost:61617" />
  <property name="userName" value="Username"/>
  <property name="password" value="Password"/>
```

```
    <property name="trustStore" value="/conf/client.ts"/>
    <property name="trustStorePassword" value="password"/>
  </bean>
```

ActiveMQSslConnectionFactory クラスには、次のプロパティーが指定されています。

brokerURL

接続するリモートブローカーの URL。この例では、ローカルホスト上の SSL 対応 OpenWire ポートに接続します。ブローカーは、互換性のあるポート設定で対応するトランスポートコネクターも定義する必要があります。

userName および password

有効な JAAS ログインクレデンシャル、**Username** および **Password**。

trustStore

SSL 接続用の証明書トラストストアを含む Java キーストアファイルの場所。場所はクラスパスリソースとして指定されます。相対パスを指定すると、リソースの場所はクラスパスの **org/jbossfuse/example** ディレクトリーからの相対パスになります。

trustStorePassword

トラストストアを含むキーストアファイルのロックを解除するパスワード。

keyStore および **keyStorePassword** プロパティーを指定することもできますが、これらのプロパティーは、SSL 相互認証が有効になっている場合にのみ必要になります (クライアントが SSL ハンドシェイク中に X.509 証明書をブローカーに提示)。

4.2. CAMEL ACTIVEMQ コンポーネントの設定例

概要

このセクションでは、サンプルの Camel ActiveMQ コンポーネントインスタンスを初期化および設定する方法について説明します。このインスタンスを使用して、Camel ルートで ActiveMQ エンドポイントを定義できます。これにより、Camel ルートがブローカーからメッセージを送受信できるようになります。

前提条件

Camel ActiveMQ コンポーネントに必要なバンドルを定義する **camel-activemq** 機能は、デフォルトではインストール**されません**。**camel-activemq** 機能をインストールするには、以下のコンソールコマンドを入力します。

```
JBossFuse:karaf@root> features:install camel-activemq
```

Camel ActiveMQ コンポーネントのサンプル

次のブループリントのサンプルは、SSL/TLS セキュリティーと JAAS 認証の両方が有効になっている Camel ActiveMQ コンポーネントの完全な設定を示しています。Camel ActiveMQ コンポーネントインスタンスは **activemqssl** Bean ID と定義されます。つまり、Camel ルート内でエンドポイントを定義するときに使用する **activemqssl** スキームに関連付けられています。

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

```
  ...
  <!--
    Configure the activemqssl component:
  -->
  <bean id="jmsConnectionFactory"
        class="org.apache.activemq.ActiveMQSslConnectionFactory">
    <property name="brokerURL" value="ssl://localhost:61617" />
    <property name="userName" value="Username"/>
    <property name="password" value="Password"/>
    <property name="trustStore" value="/conf/client.ts"/>
    <property name="trustStorePassword" value="password"/>
  </bean>

  <bean id="pooledConnectionFactory"
        class="org.apache.activemq.pool.PooledConnectionFactory">
    <property name="maxConnections" value="8" />
    <property name="maximumActive" value="500" />
    <property name="connectionFactory" ref="jmsConnectionFactory" />
  </bean>

  <bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration">
    <property name="connectionFactory" ref="pooledConnectionFactory"/>
    <property name="transacted" value="false"/>
    <property name="concurrentConsumers" value="10"/>
  </bean>

  <bean id="activemqssl"
        class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="configuration" ref="jmsConfig"/>
  </bean>

</beans>
```

Camel ルートのサンプル

以下の Camel ルートは、前述の例で定義された Camel ActiveMQ コンポーネントを参照する **activemqssl** スキームを使用して、ブローカーの **security.test** キューにメッセージを安全に送信するサンプルエンドポイントを定義します。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
  ...
  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="timer://myTimer?fixedRate=true&period=5000"/>
      <transform><constant>Hello world!</constant></transform>
      <to uri="activemqssl:security.test"/>
    </route>
  </camelContext>
  ...
</beans>
```

第5章 CAMEL CXF コンポーネントのセキュリティー保護

概要

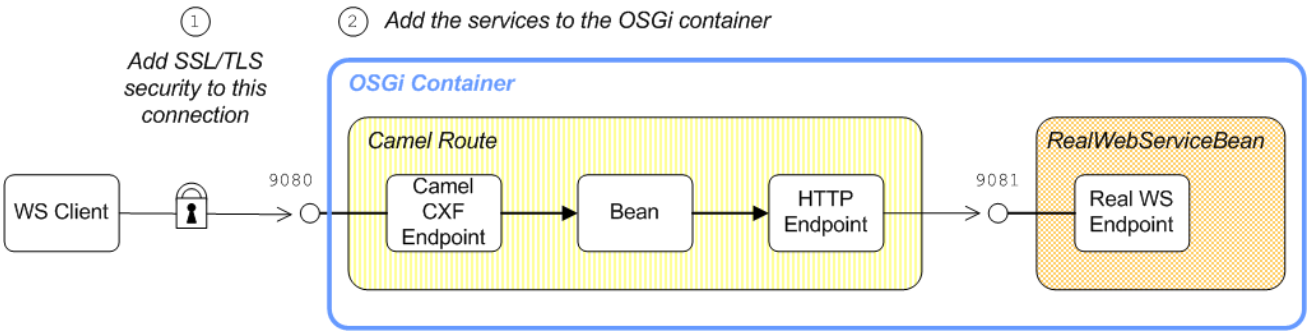
この章では、Camel CXF プロキシーのデモンストレーションを開始点として使用して、Camel CXF エンドポイントで SSL/TLS セキュリティーを有効にする方法について説明します。Camel CXF コンポーネントを使用すると、Apache CXF エンドポイントを Apache Camel ルートに追加できます。これにより、Apache Camel で Web サービスをシミュレートしたり、WS クライアントと Web サービスの間にルートを挿入して追加の処理を実行したり (ここで検討するケース) できます。

5.1. CAMEL CFX プロキシーのデモンストレーション

概要

このチュートリアルでは、OSGi で Camel CXF エンドポイントをセキュリティー保護する方法を説明するために、Apache Camel のスタンドアロンディストリビューションから使用できる例を示しています。それが **Camel CXF プロキシー** デモ です。図5.1「Camel CFX プロキシーの概要」 は、このデモンストレーションがどのように機能するかの概要を示しています。

図5.1 Camel CFX プロキシーの概要



RealWebServiceBean によって実装された**レポートインシデント** Web サービスは、インシデント (例: 交通事故) の詳細を受信し、クライアントに追跡コードを返します。ただし、WS クライアントは、要求を実際の Web サービスに直接送信する代わりに、WS クライアントと実際の Web サービスの間に置かれたエンドポイントに接続します。Apache Camel ルートは (**enrichBean** を使用して) WSDL メッセージに何らかの処理を実行した後に実際の Web サービスに転送します。

警告

SSL/TLS セキュリティーを有効にする場合は、Poodle 脆弱性 (CVE-2014-3566) に対して保護するために、SSLv3 プロトコルを明示的に無効にする必要があります。詳細は、Disabling SSLv3 in JBoss Fuse 6.x and JBoss A-MQ 6.x を参照してください。

変更

OSGi のコンテキストで Camel CXF エンドポイントの SSL/TLS を有効にする方法を示すために、この章では基本的なデモンストレーションを次のように変更する方法を説明しています。

1. SSL/TLS セキュリティーは、WS クライアントと Camel CXF エンドポイント間の接続で有効化されている。
2. Apache Camel ルートと **RealWebServiceBean** Bean の両方が OSGi コンテナーにデプロイされている。

デモンストレーションコードの取得

Camel CXF プロキシーのデモンストレーションは、**InstallDir/extras** ディレクトリーに含まれる Apache Camel のスタンドアロンディストリビューションでのみ利用可能です。標準のアーカイブユーティリティーを使用して、Camel アーカイブファイルをデプロイメントし、ファイルシステム上の便利な場所にコンテンツを抽出します。

Apache Camel を **CamelInstallDir** にインストールした場合、Camel CFX プロキシーのデモは次のディレクトリーにあります。

> **CamelInstallDir**/examples/camel-example-cxf-proxy

サンプル証明書の取得

このデモには X.509 証明書が必要です。実際のデプロイメントでは、プライベート認証局を使用してこれらの証明書を自分で生成する必要があります。ただし、このデモでは、Apache CXF の **wsdl_first_http** の例からいくつかのサンプル証明書を使用します。このデモは、**InstallDir/extras** ディレクトリーに含まれる Apache CXF のスタンドアロンディストリビューションから入手できます。標準のアーカイブユーティリティーを使用して、CFX アーカイブファイルをデプロイメントし、ファイルシステム上の便利な場所にコンテンツを抽出します。

Apache CXF を **CXFInstallDir** にインストールしている場合、以下のディレクトリーに **wsdl_first_http** デモンストレーションがあります。

> **CXFInstallDir**/samples/wsdl_first_http

WSDL コントラクトの物理部分

WSDL コントラクトの物理部分は、**wsdl:service** および **wsdl:port** 要素を参照します。これらの要素は、特定の Web サービスエンドポイントに接続するために必要なトランスポートの詳細を指定します。このデモンストレーションの目的において、これは契約の中でも最も興味深い部分です。詳細については 例5.1「ReportIncidentEndpointService WSDL サービス」 を参照してください。

例5.1 ReportIncidentEndpointService WSDL サービス

```
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    ...
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 targetNamespace="http://reportincident.example.camel.apache.org">
    ...
    <!-- Service definition -->
    <wsdl:service name="ReportIncidentEndpointService">
        <wsdl:port name="ReportIncidentEndpoint" binding="tns:ReportIncidentBinding">
            <soap:address location="http://localhost:9080/camel-example-cxf-
proxy/webservices/incident"/>
        </wsdl:port>
```

```
    </wsdl:service>

</wsdl:definitions>
```

注記

WSDL コントラクトに表示されるアドレス URL (**soap:address** 要素の **location** 属性の値) は、アプリケーションコードがアドレス URL のデフォルト値を上書きするため、ここでは重要ではありません。

WSDL アドレス指定の詳細

WS クライアントが WSDL サービスに接続するには、**WSDL サービス名**、**WSDL ポート名**、および Web サービスの **アドレス URL** の 3 つの情報が必要です。この例では、プロキシー Web サービスや実際の Web サービスとの接続に、次のアドレス指定の詳細が使用されます。

WSDL サービス名

WSDL サービスの完全な QName は次のとおりです。

```
{http://reportincident.example.camel.apache.org}ReportIncidentEndpointService
```

WSDL ポート名

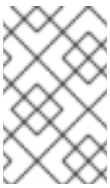
WSDL ポートの完全な QName は次のとおりです。

```
{http://reportincident.example.camel.apache.org}ReportIncidentEndpoint
```

アドレス URL

プロキシー Web サービス エンドポイント (HTTPS プロトコルを使用) のアドレス URL は次のとおりです。

```
https://localhost:9080/camel-example-cxf-proxy/webservices/incident
```

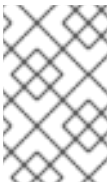


注記

上記のアドレスは、バンドルの Spring 設定ファイル **src/main/resources/META-INF/spring/camel-config.xml** で **cxf:cxfEndpoint** 要素を使用して **reportIncident** Bean が作成されると指定されます。

実際の Web サービス エンドポイント (HTTP プロトコルを使用する) のアドレス URL は次のとおりです。

```
http://localhost:9081/real-webservice
```



注記

上記のアドレスは、**realWebService** Bean がバンドルの Spring 設定ファイル (**src/main/resources/META-INF/spring/camel-config.xml**) で作成される際に指定されます。

5.2. WEB サービスプロキシーのセキュリティー保護

概要

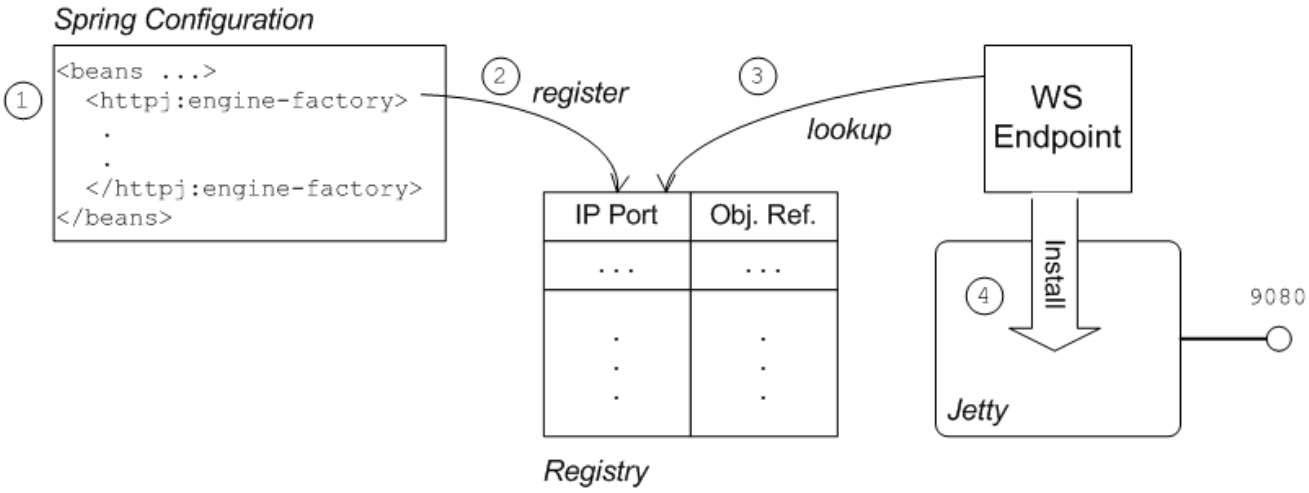
このセクションでは、実際の Web サービスのプロキシーとして機能する Camel CXF エンドポイントで SSL/TLS セキュリティーを有効にする方法について説明します。X.509 証明書をすでに利用できる場合、必要なのは設定データのブロックを Spring 設定ファイル (設定データは **httpj:engine-factory** 要素に含まれる) に追加することだけです。ただし、Camel CXF エンドポイントが SSL/TLS 設定の詳細にどのように関連付けられるかを理解する必要があるという点のみ、少し注意が必要です。

暗黙的な設定

WS エンドポイントは、Spring でエンドポイントを作成し、その Jetty コンテナーで SSL/TLS プロパティーを設定することで設定できます。しかし、設定が若干複雑になることもあります。Jetty コンテナー (Spring の **httpj:engine-factory** 要素によって設定される) は、含まれる WS エンドポイントを明示的に参照**せず**、WS エンドポイントは **Jetty コンテナーを明示的に参照しない**からです。Jetty コンテナーとそれに含まれるエンドポイント間の接続は、httpj:engine-factory によって暗黙的に設定された WS エンドポイント で示されるとおり、両方が同じ TCP ポートを使用するように設定されているという点で暗黙的に確立されます。

httpj:engine-factory によって暗黙的に設定された WS エンドポイント

Element



Web サービスエンドポイントと **httpj:engine-factory** 要素間の接続は以下のように確立されます。

1. Spring コンテナーは、**httpj:engine-factory** 要素を含むファイルを読み込み、解析します。
2. **httpj:engine-factory** Bean が作成されると、対応するエントリーがレジストリーに作成され、Bean への参照が格納されます。**httpj:engine-factory** Bean は、指定の TCP ポートをリッスンする Jetty コンテナーを初期化するために使用されます。
3. WS エンドポイントが作成されると、レジストリーをスキャンし、エンドポイントのアドレス URL の TCP ポートと同じ TCP ポートを持つ **httpj:engine-factory** Bean を検索できるかどうかを確認します。
4. Bean の 1 つがエンドポイントの TCP ポートと一致する場合、WS エンドポイントはそれ自体を対応する Jetty コンテナーにインストールします。Jetty コンテナーで SSL/TLS が有効になっている場合、WS エンドポイントはそれらのセキュリティー設定を共有します。

Jetty コンテナーに SSL/TLS セキュリティーを追加する手順

Jetty コンテナーに SSL/TLS セキュリティーを追加して、WS プロキシーエンドポイントをセキュリティー保護するには、次の手順を実行します。

1. 「バンドルリソースへの証明書の追加」。
2. 「POM を変更してリソースフィルタリングを無効化」。
3. 「CXF バスのインスタンス化」。
4. 「Spring への httpj:engine-factory 要素の追加」。
5. 「接頭辞 cxfcore:、sec:、httpj: の定義」。
6. 「プロキシーアドレス URL を変更して HTTPS の使用を有効化」。

バンドルリソースへの証明書の追加

このデモンストレーションで使用される証明書は、Apache CXF 3.3.6.fuse-7_13_0-00015-redhat-00001 製品のサンプルから取得されます。スタンドアロンバージョンの Apache CXF (**InstallDir/extras/** ディレクトリーで利用可能) をインストールする場合は、**CXFInstallDir/samples/wsdl_first_https/src/main/config** ディレクトリーにサンプル証明書があります。

clientKeystore.jks および **serviceKeystore.jks** キーストアを **CXFInstallDir/samples/wsdl_first_https/src/main/config** ディレクトリーから **CamelInstallDir/examples/camel-example-cxf-proxy/src/main/resources/certs** ディレクトリーにコピーします (最初に **certs** サブディレクトリーを作成する必要があります)。

POM を変更してリソースフィルタリングを無効化

証明書をリソースとしてバンドルに直接含めることは、証明書をデプロイするための最も便利な方法です。ただし、証明書を Maven プロジェクトのリソースとしてデプロイする場合は、バイナリーファイルを破損する Maven リソースフィルタリングを無効にすることを忘れないでください。

Maven で **.jks** ファイルのフィルターを無効にするには、プロジェクト POM ファイル、**CamelInstallDir/examples/camel-example-cxf-proxy/pom.xml** を開き、テキストエディターで以下の **resources** 要素を **build** 要素の子として追加します。

```
<?xml version="1.0" encoding="UTF-8"?>
...
<project ...>
  ...
  <build>
    <plugins>
      ...
    </plugins>

    <resources> <resource> <directory>src/main/resources</directory> <filtering>true</filtering>
<excludes> <exclude>/.jks</exclude> </excludes> </resource> <resource>
<directory>src/main/resources</directory> <filtering>false</filtering> <includes>
<include>/.jks</include> </includes> </resource> </resources>
  </build>

</project>
```

-

CXF バスのインスタンス化

Spring XML で明示的に CXF バスをインスタンス化する必要があります (これは、次の手順の **httpj:engine-factory** 要素によってインスタンス化される Jetty コンテナーが利用できるようにするためです)。**src/main/resources/META-INF/spring** ディレクトリーの **camel-config.xml** ファイルを編集し、**cxfcore:bus** 要素を **beans** 要素の子として追加します。

```
<beans ... >
    ...
    <cxfcore:bus/>
    ...
</beans>
```

注記

cxfcore: namespace の接頭辞は、後のステップで定義します。

Spring への httpj:engine-factory 要素の追加

configuration

TCP ポート 9080 でリッスンする Jetty コンテナーを設定するには、**src/main/resources/META-INF/spring** ディレクトリーの **camel-config.xml** ファイルを編集し、例5.2「SSL/TLS が有効になっている httpj:engine-factory 要素」 に示されているように **httpj:engine-factory** 要素を追加します。

この例では、**sec:clientAuthentication** 要素の **required** 属性は **false** に設定されています。つまり、SSL/TLS ハンドシェイク中に、サーバーに X.509 証明書を提示する必要は**ありません** (ただし、証明書がある場合は提示できます)。

例5.2 SSL/TLS が有効になっている httpj:engine-factory 要素

```
<beans ... >
    ...
    <httpj:engine-factory bus="cxf">
      <httpj:engine port="${proxy.port}">
        <httpj:tlsServerParameters secureSocketProtocol="TLSv1">
          <sec:keyManagers keyPassword="skpass">
            <sec:keyStore resource="certs/serviceKeystore.jks" password="sspass" type="JKS"/>
          </sec:keyManagers>
          <sec:trustManagers>
            <sec:keyStore resource="certs/serviceKeystore.jks" password="sspass" type="JKS"/>
          </sec:trustManagers>
          <sec:cipherSuitesFilter>
            <sec:include>.*_WITH_3DES_.*</sec:include>
            <sec:include>.*_WITH_DES_.*</sec:include>
            <sec:exclude>.*_WITH_NULL_.*</sec:exclude>
            <sec:exclude>.*_DH_anon_.*</sec:exclude>
          </sec:cipherSuitesFilter>
          <sec:clientAuthentication want="true" required="false"/>
        </httpj:tlsServerParameters>
      </httpj:engine>
```

```
    </httpj:engine-factory>

</beans>
```

重要

Poodle 脆弱性 (CVE-2014-3566) から保護するために、サーバー側で secureSocketProtocol を **TLSv1** に設定する必要があります。

接頭辞 cxfcore:、sec:、httpj: の定義

camel-config.xml ファイルの **beans** 要素に次の強調表示された行を追加して、**cxfcore:bus** 要素と **httpj:engine-factory** 要素の定義に表示される **cxfcore:**、**sec:**、および **httpj:** 名前空間接頭辞を定義します。

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:camel="http://camel.apache.org/schema/spring"
      xmlns:cxf="http://camel.apache.org/schema/cxf"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:cxfcore="http://cxf.apache.org/core"
      xmlns:sec="http://cxf.apache.org/configuration/security"
      xmlns:httpj="http://cxf.apache.org/transports/http-jetty/configuration"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
      http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd
      http://camel.apache.org/schema/cxf http://camel.apache.org/schema/cxf/camel-cxf.xsd
      http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
      http://cxf.apache.org/core http://cxf.apache.org/schemas/core.xsd
      http://cxf.apache.org/configuration/security
http://cxf.apache.org/schemas/configuration/security.xsd
      http://cxf.apache.org/transports/http-jetty/configuration
http://cxf.apache.org/schemas/configuration/http-jetty.xsd
      ">
```

注記

http://cxf.apache.org/configuration/security スキーマと **http://cxf.apache.org/transports/http-jetty/configuration** スキーマの場所を **xsi:schemaLocation** 属性に指定することが 必須 です。これらは、OSGi コンテナーで自動的に提供されません。

プロキシーアドレス URL を変更して HTTPS の使用を有効化

Apache Camel ルート先頭のプロキシーエンドポイントは、**camel-config.xml** ファイルの **cxf:cxfEndpoint** 要素によって設定されます。デフォルトでは、このプロキシーエンドポイントは HTTP プロトコルを使用するように設定されています。ただし、代わりにセキュアな HTTPS プロトコルを使用するには、アドレス URL を変更する必要があります。**camel-config.xml** ファイルで、以下のフラグメントが示すように、**cxf:cxfEndpoint** 要素の address 属性を編集し、**http:** 接頭辞を **https:** 接頭辞に置き換えます。

-

```
<beans ...>
  ...
  <cxf:cxfEndpoint id="reportIncident"
                   address="https://localhost:${proxy.port}/camel-example-cxf-proxy/webservices/incident"
                   endpointName="s:ReportIncidentEndpoint"
                   serviceName="s:ReportIncidentEndpointService"
                   wsdlURL="etc/report_incident.wsdl"
                   xmlns:s="http://reportincident.example.camel.apache.org"/>
  ...
</beans>
```

また、アドレス URL は TCP ポート **${proxy.port}** を使用するように設定されています (デフォルト値は **9080**)。この TCP ポート値は Jetty コンテナーに設定された値と同じであるため (**http:engine-factory** 要素によって設定)、このエンドポイントが Jetty コンテナーにデプロイされるようにします。**cxf:cxfEndpoint** の属性は、「WSDL アドレス指定の詳細」 で説明されているように WSDL アドレスの詳細を指定します。

serviceName

WSDL サービス名を指定します。

endpointName

WSDL ポート名を指定します。

address

プロキシー Web サービスのアドレス URL を指定します。

5.3. APACHE CAMEL ルートのデプロイ

概要

基本的な Camelcamel プロキシーデモんストレーションの MavenPOM ファイルは、OSGi バンドルを生成するようにすでに設定されています。そのため、Maven を使用してデモンストレーションをビルドした後、Apache Camel ルートと **RealWebServicesBean** Bean が含まれるデモンストレーションバンドルを OSGi コンテナーにデプロイする準備が整います。

前提条件

Apache Camel ルートを OSGi コンテナーにデプロイする前に、前のセクション (「Web サービスプロキシーのセキュリティー保護」) で説明したように、SSL/TLS セキュリティーを使用するようにプロキシー Web サービスを設定する必要があります。

Camel ルートをデプロイメントする手順

Web サービスプロキシーのデモンストレーションを OSGi コンテナーにデプロイするには、以下の手順を実行します。

1. 「デモンストレーションのビルド」.
2. 「OSGi コンテナーの起動」.
3. 「必要な機能のインストール」.
4. 「バンドルのデプロイ」.
5. 「コンソール出力の確認」.

デモンストレーションのビルド

Maven を使用して、デモンストレーションを OSGi バンドルとしてビルドおよびインストールします。コマンドプロンプトを開き、現在のディレクトリーを **CamelInstallDir/examples/camel-example-cxf-proxy** に切り替え、以下のコマンドを入力します。

```
mvn install -Dmaven.test.skip=true
```

OSGi コンテナーの起動

まだ行っていない場合は、新しいコマンドプロンプトで次のコマンドを入力して、Karaf コンソール (およびコンテナーインスタンス) を起動します。

```
./fuse
```

必要な機能のインストール

Camel/CXF コンポーネントに必要なバンドルを定義する **camel-cxf** 機能は、デフォルトではインストール**されません**。**camel-cxf** 機能をインストールするには、以下のコンソールコマンドを入力します。

```
JBossFuse:karaf@root> features:install camel-cxf
```

また、Camel/HTTP コンポーネントに必要なバンドルを定義する **camel-http** 機能も必要です。**camel-http** 機能をインストールするには、以下のコンソールコマンドを入力します。

```
JBossFuse:karaf@root> features:install camel-http
```

バンドルのデプロイ

以下のコンソールコマンドを入力して **camel-example-cxf-proxy** バンドルをデプロイします。

```
JBossFuse:karaf@root> install -s mvn:org.apache.camel/camel-example-cxf-proxy/2.23.2.fuse-7_13_0-00013-redhat-00001
```

注記

この場合、コンソール画面でバンドル出力を確認できるように、ホットデプロイではなく、**install** を使用してバンドルを直接デプロイすることが推奨されます。

mvn URL ハンドラーの使用に問題がある場合は、設定方法の詳細を olink: esbosguide/urlHandlers-Maven で確認してください。

コンソール出力の確認

バンドルが正常にデプロイされると、コンソールウィンドウに次のような出力が表示されます。

```
JBossFuse:karaf@root> Starting real web service...
Started real web service at: http://localhost:9081/real-webservice
```

5.4. WEB サービスクライアントのセキュリティー保護

概要

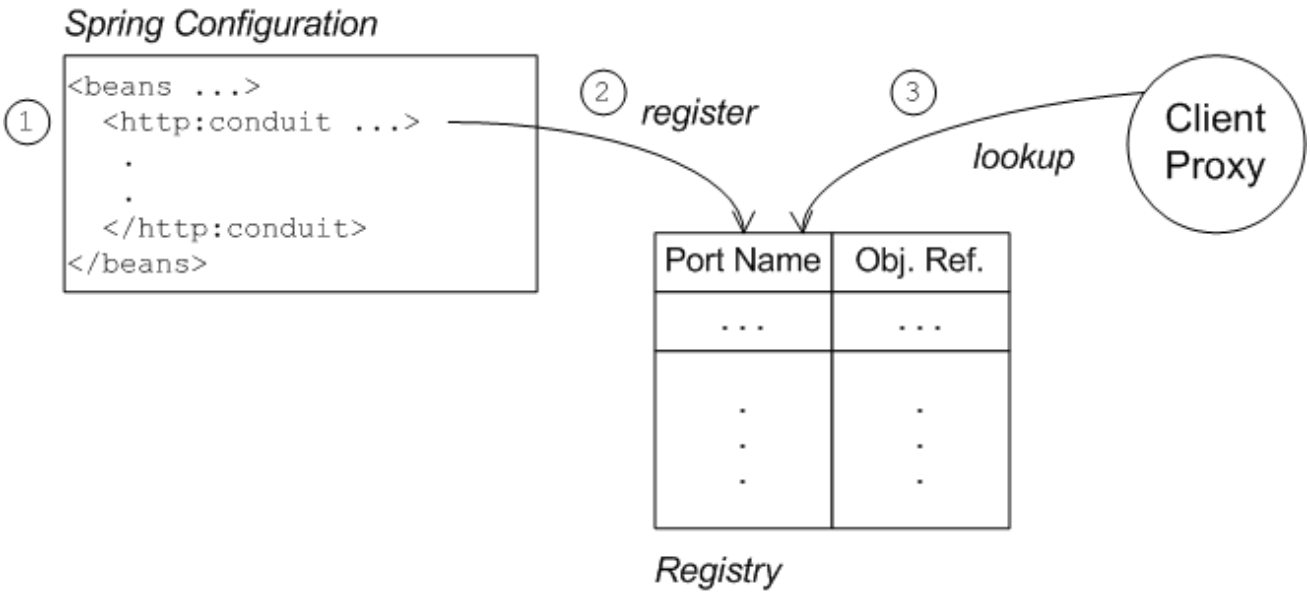
基本的な Camel CXF プロキシーのデモンストレーションでは、Web サービスクライアントは実際には **src/test** ディレクトリーに JUnit テストとして実装されます。そのため、クライアントは Maven コマンド **mvn test** を使用して簡単に実行できます。クライアントで SSL/TLS セキュリティーを有効にするため、テストクライアントの Java 実装が完全に置き換えられ、SSL/TLS 設定を含む Spring ファイルが **src/test/resources/META-INF/spring** ディレクトリーに追加されます。クライアントを設定するために実行する必要のある手順を説明する前に、このセクションでクライアントの Java コードと Spring 設定の詳細について説明します。

暗黙的な設定

エンドポイントアドレスの URL スキームを **https:** に変更する以外に、クライアントプロキシーで SSL/TLS セキュリティーを有効にする設定のほとんどは、Spring 設定の **http:conduit** 要素に含まれます。ただし、この設定をクライアントプロキシーに適用する方法は、混乱を招く可能性があります。それは、**http:conduit** 要素によって**クライアントプロキシーが明示的に参照されず**、クライアントプロキシーは **http:conduit 要素を明示的に参照しない**ためです。**http:conduit** 要素とクライアントプロキシー間の接続は暗黙的に確立され、http:conduit で暗黙的に設定されたクライアントプロキシー が示すように、いずれも同じ WSDL ポートを参照します。

http:conduit で暗黙的に設定されたクライアントプロキシー

Element



以下のように、クライアントプロキシーと **http:conduit** 要素間の接続が確立されます。

1. クライアントは **http:conduit** 要素が含まれる Spring 設定ファイルをロードおよび解析します。
2. **http:conduit** Bean が作成されると、対応するエントリーがレジストリーに作成されます。これは、指定の WSDL ポート名の下に Bean への参照を保存します (名前は QName 形式で保存されます)。

3. JAX-WS クライアントプロキシーが作成されると、レジストリーをスキャンし、プロキシーの WSDL ポート名に関連付けられた **http:conduit** Bean を見つけられるかどうかを確認します。そのような Bean が見つかると、設定の詳細がプロキシーに自動的に挿入されます。

クライアント側で必要な証明書

クライアントは、**src/main/resources/certs** ディレクトリーからの以下の **clientKeystore.jks** キーストアファイルで設定されます。このキーストアには、次の 2 つのエントリーが含まれています。

信頼できる証明書エントリー

サーバー証明書とクライアント証明書の両方を発行して署名した CA 証明書を含む信頼できる証明書エントリー。

秘密鍵の入力

クライアント自身の X.509 証明書と秘密鍵を含む秘密鍵エントリー。実際、サーバーでは TLS ハンドシェイク中にクライアントによる証明書の送信が必要ないため、現在の例を実行するためにこの証明書が必ず必要なわけではありません (例5.2「SSL/TLS が有効になっている httpj:engine-factory 要素」 を参照)。

Spring 定義をクライアントにロード

サンプルクライアントは Spring コンテナーに直接デプロイされていませんが、セキュアな HTTP コンジットを定義するためにいくつかの Spring 定義が必要です。では、Spring コンテナーなしで Spring 定義を作成するにはどうすればよいでしょうか。**org.apache.cxf.bus.spring.SpringBusFactory** クラスを使用すると、Spring 定義を Java ベースのクライアントに簡単に読み取ることができます。

以下のコードは、**META-INF/spring/cxf-client.xml** ファイルから Spring 定義を読み取り、これらの定義を取り入れた Apache CXF Bus オブジェクトを作成する方法を示しています。

```
// Java
import org.apache.cxf.bus.spring.SpringBusFactory;
...
protected void startCxfBus() throws Exception {
    bf = new SpringBusFactory();
    Bus bus = bf.createBus("META-INF/spring/cxf-client.xml");
    bf.setDefaultBus(bus);
}
```

クライアントプロキシーの作成

原則として、WSDL プロキシーの作成にはいくつかの方法があります。JAX-WS API を使用して WSDL ファイルの内容に基づいてプロキシーを作成したり、WS-WS API を使用して WSDL ファイル**なし**でプロキシーを作成できます。また、Apache CXF 固有のクラス **JaxWsProxyFactoryBean** を使用してプロキシーを作成できます。

この SSL/TLS クライアントの場合、次の Java サンプルに示すように、最も便利なアプローチは、JAX-WS API を使用して WSDL ファイルを使用せずにプロキシーを作成することです。

```
// Java
import javax.xml.ws.Service;
import org.apache.camel.example.reportincident.ReportIncidentEndpoint;
...
// create the webservice client and send the request
Service s = Service.create(SERVICE_NAME);
```

```
s.addPort(
    PORT_NAME,
    "http://schemas.xmlsoap.org/soap/",
    ADDRESS_URL
  );
ReportIncidentEndpoint client =
  s.getPort(PORT_NAME, ReportIncidentEndpoint.class);
```

注記

この例では、**JaxWsProxyFactoryBean** のアプローチを使用してプロキシーを作成**できません**。これは、このように作成されたプロキシーは Spring 設定ファイルで指定された HTTP コンジット設定を見つけることができないためです。

SERVICE_NAME および **PORT_NAME** 定数は、例5.1「ReportIncidentEndpointService WSDL サービス」 で定義されているように、それぞれ WSDL サービスおよび WSDL ポートの QNames です。**ADDRESS_URL** 文字列には、プロキシー Web サービスアドレスと同じ値があり、以下のように定義されます。

```
private static final String ADDRESS_URL =
  "https://localhost:9080/camel-example-cxf-proxy/webservices/incident";
```

特に、このアドレスは URL スキーム **https** で定義し、SSL/TLS 経由の HTTP を選択する必要があることに**注意してください**。

クライアントへの SSL/TLS セキュリティー追加手順

SSL/TLS セキュリティーが有効になっている JAX-WS クライアントを定義するには、次の手順を実行します。

1. 「テストケースとして Java クライアントを作成」。
2. 「Spring 設定への http:conduit 要素の追加」。
3. 「クライアントの実行」。

テストケースとして Java クライアントを作成

例5.3「ReportIncidentRoutesTest Java クライアント」 は、JUnit テストケースとして実装された Java クライアントの完全なコードを示しています。このクライアントは、**examples/camel-example-cxf-proxy** デモンストレーションの **src/test/java/org/apache/camel/example/reportincident** サブディレクトリーに、既存のテストである **ReportIncidentRoutesTest.java** を置き換えます。

クライアントを **CamelInstallDir/examples/camel-example-cxf-proxy** デモに追加するには、**src/test/java/org/apache/camel/example/reportincident** サブディレクトリーに移動し、既存の **ReportIncidentRoutesTest.java** ファイルをバックアップの場所に移動し、新しい **ReportIncidentRoutesTest.java** ファイルを作成して、例5.3「ReportIncidentRoutesTest Java クライアント」 のコードをこのファイルに貼り付けます。

例5.3 ReportIncidentRoutesTest Java クライアント

```
// Java
package org.apache.camel.example.reportincident;
```

```
import org.apache.camel.spring.Main;
import org.apache.cxf.jaxws.JaxWsProxyFactoryBean;
import org.junit.Test;

import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;

import org.apache.cxf.Bus;
import org.apache.cxf.bus.spring.SpringBusFactory;
import org.apache.camel.example.reportincident.ReportIncidentEndpoint;
import org.apache.camel.example.reportincident.ReportIncidentEndpointService;

import static org.junit.Assert.assertEquals;

/**
 * Unit test of our routes
 */
public class ReportIncidentRoutesTest {

    private static final QName SERVICE_NAME
        = new QName("http://reportincident.example.camel.apache.org",
"ReportIncidentEndpointService");

    private static final QName PORT_NAME =
        new QName("http://reportincident.example.camel.apache.org", "ReportIncidentEndpoint");

    private static final String WSDL_URL = "file:src/main/resources/etc/report_incident.wsdl";

    // should be the same address as we have in our route
    private static final String ADDRESS_URL = "https://localhost:9080/camel-example-cxf-
proxy/webservices/incident";

    protected SpringBusFactory bf;

    protected void startCxfBus() throws Exception {
        bf = new SpringBusFactory();
        Bus bus = bf.createBus("META-INF/spring/cxf-client.xml");
        bf.setDefaultBus(bus);
    }

    @Test
    public void testRendportIncident() throws Exception {
        startCxfBus();
        runTest();
    }

    protected void runTest() throws Exception {

        // create input parameter
        InputReportIncident input = new InputReportIncident();
        input.setIncidentId("123");
        input.setIncidentDate("2008-08-18");
        input.setGivenName("Claus");
        input.setFamilyName("Ibsen");
        input.setSummary("Bla");
```

```
        input.setDetails("Bla bla");
        input.setEmail("davsclaus@apache.org");
        input.setPhone("0045 2962 7576");

        // create the webservice client and send the request
        Service s = Service.create(SERVICE_NAME);
        s.addPort(PORT_NAME, "http://schemas.xmlsoap.org/soap/", ADDRESS_URL);
        ReportIncidentEndpoint client = s.getPort(PORT_NAME, ReportIncidentEndpoint.class);

        OutputReportIncident out = client.reportIncident(input);

        // assert we got a OK back
        assertEquals("OK;456", out.getCode());
    }
}
```

Spring 設定への http:conduit 要素の追加

例5.4「SSL/TLS が有効になっている http:conduit 要素」 は、**ReportIncidentEndpoint** WSDL ポートの **http:conduit** 要素を定義する Spring 設定を示しています。**http:conduit** 要素は、指定の WSDL ポートを使用するクライアントプロキシーの SSL/TLS セキュリティーを有効にするように設定されます。

クライアントテストケースに Spring 設定を追加するには、**src/test/resources/META-INF/spring** サブディレクトリーを作成し、お気に入りのテキストエディターを使用してファイル **cxf-client.xml** を作成し、例5.4「SSL/TLS が有効になっている http:conduit 要素」 の内容をそのファイルに貼り付けます。

例5.4 SSL/TLS が有効になっている http:conduit 要素

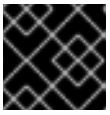
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:cxf="http://camel.apache.org/schema/cxf"
       xmlns:sec="http://cxf.apache.org/configuration/security"
       xmlns:http="http://cxf.apache.org/transports/http/configuration"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
       http://camel.apache.org/schema/cxf http://camel.apache.org/schema/cxf/camel-cxf.xsd
       http://cxf.apache.org/configuration/security
http://cxf.apache.org/schemas/configuration/security.xsd
       http://cxf.apache.org/transports/http/configuration
http://cxf.apache.org/schemas/configuration/http-conf.xsd
       ">

  <http:conduit name="
{http://reportincident.example.camel.apache.org}ReportIncidentEndpoint.http-conduit">
    <http:tlsClientParameters disableCNCheck="true" secureSocketProtocol="TLSv1">
      <sec:keyManagers keyPassword="ckpass">
        <sec:keyStore password="cspass" type="JKS"
        resource="certs/clientKeystore.jks" />
      </sec:keyManagers>
      <sec:trustManagers>
        <sec:keyStore password="cspass" type="JKS"
        resource="certs/clientKeystore.jks" />
```

```
    </sec:trustManagers>
    <sec:cipherSuitesFilter>
      <sec:include>.*_WITH_3DES_.*</sec:include>
      <sec:include>.*_WITH_DES_.*</sec:include>
      <sec:exclude>.*WITH_NULL.</sec:exclude>*
      <sec:exclude>.*DH_anon.</sec:exclude>*
    </sec:cipherSuitesFilter>
  </http:tlsClientParameters>
 </http:conduit>

</beans>
```

上記の設定については、次の点に注意してください。

- **http:conduit** 要素を定義するのに、**http:** および **sec:** 名前空間の接頭辞が必要です。**xsi:schemaLocation** 要素では、対応する **http://cxf.apache.org/configuration/security** および **http://cxf.apache.org/transports/http/configuration** 名前空間の場所を指定する必要もあります。

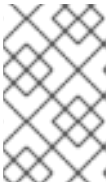
- **http:tlsClientParameters** 要素の **disableCNCheck** 属性は **true** に設定されます。これは、サーバーの X.509 証明書のコモンネームがサーバーのホスト名と一致するかどうかをチェック**しない** ことを意味します。詳細は 付録A *証明書の管理* を参照してください。

  重要

  実稼働環境では、CN チェックを無効にすることは推奨 **しません**。

- **sec:keystore** 要素では、クラスパス上の証明書を見つける **resource** 属性を使用して証明書の場所が指定されます。Maven がテストを実行すると、証明書が **src/main/resources/certs** ディレクトリーから読み取りできるように、クラスパスで **src/main/resources** の内容を自動的に利用できるようにします。

  注記

  また、ファイルシステム内を検索する **file** 属性を使用して、証明書の場所を指定するオプションもあります。ただし、**resource** 属性は、バンドルにパッケージ化されたアプリケーションでの使用に適しています。

- **sec:cipherSuitesFilter** 要素は、**.*WITH_NULL.*** および **.*DH_anon.*** に一致する暗号スイートを除外するように設定されています。これらの暗号化スイートは事実上不完全であり、通常の使用を目的としたものでは **ありません**。

  重要

  .*WITH_NULL.* および **.*DH_anon.*** に一致する暗号を常に**除外**することが推奨されます。

- **secureSocketProtocol** 属性は、サーバーのプロトコルと一致するように TLSv1 に設定し、SSLv3 プロトコルが使用されないようにする必要があります (POODLE セキュリティー脆弱性 (CVE-2014-3566))。

クライアントの実行

クライアントはテストケースとして定義されているため、標準の Maven テスト目標を使用してクライアントを実行できます。クライアントを実行するには、新しいコマンドウィンドウを開き、**CamelInstallDir/examples/camel-example-cxf-proxy** ディレクトリーに移動し、以下の Maven コマンドを入力します。

```
mvn test
```

テストが正常に実行されると、OSGi コンソールウィンドウに次の出力が表示されます。

```
Incident was 123, changed to 456

Invoked real web service: id=456 by Claus Ibsen
```

第6章 FUSE CONSOLE のセキュア化

スタンドアロンデプロイメントで Fuse コンテナーを保護するために以下のセキュリティー機能を実装する方法の詳細は、Fuse on Karaf スタンドアロンの管理 を参照してください。

- HTTPS を必須プロトコルとして設定する
- 公開鍵を使用して応答をセキュアにする
- SSL/TLS セキュリティーを有効にする
- ユーザーのアクセスを制御する

デフォルトでは、Fuse Console にリモートでアクセスすることはできません。Fuse Console にリモートでアクセスする方法については、Fuse on Karaf スタンドアロンの管理 ガイドの「Fuse Console のアンロック」セクションを参照してください。

第7章 RED HAT SINGLE SIGN-ON とのインテグレーション

Red Hat Single Sign-On (RH-SSO) オプションは JAAS と連携して機能し、Fuse および Fuse 管理サービス (SSH、JMX、および Fuse 管理コンソール) 内で実行される特定の Web クライアントアプリケーションおよびサービスにエンタープライズセキュリティーを提供します。

Red Hat Fuse には、次のタイプのコンテナー用のアダプターが用意されています。

- 「Spring Boot コンテナー用アダプター」
- 「Apache Karaf コンテナー用アダプター」
- 「JBoss EAP コンテナー用アダプター」

7.1. SPRING BOOT コンテナー用アダプター

Spring Boot コンテナーのアダプターは、次の組み込み Web コンテナーをサポートしています。

- Undertow
- Jetty
- Tomcat

Spring Boot コンテナー用の Red Hat Single Sign-On アダプターのインストールと使用の詳細については、Red Hat Single Sign-On **Securing Applications and Services Guide** の Spring Boot Adapter を参照してください。

7.2. APACHE KARAF コンテナー用アダプター

Apache Karaf コンテナーのアダプターは、次のコンポーネントをセキュリティー保護できます。

- Pax Web War Extender を使用して Fuse にデプロイされた Classic WAR アプリケーション
- Pax Web Whiteboard Extender で Fuse に OSGI サービスとしてデプロイされたサーブレットと、標準の OSGi Enterprise HTTP Service である **org.osgi.service.http.HttpService#registerServlet()`** 経由で登録された追加サーブレット
- Camel Undertow コンポーネントで実行している Apache Camel Undertow エンドポイント
- 独自の個別の Undertow エンジンで実行される Apache CXF エンドポイント
- CXF サーブレットによって提供されるデフォルトのエンジンで実行される Apache CXF エンドポイント
- SSH および JMX 管理者アクセス
- Hawtio 管理コンソール

Apache Karaf コンテナー用の Red Hat Single Sign-On アダプターのインストールと使用の詳細については、Red Hat Single Sign-On **Securing Applications and Services Guide** の JBoss Fuse 7 Adapter を参照してください。

7.3. JBOSS EAP コンテナー用アダプター

JBoss Enterprise Application Platform (EAP) コンテナーのアダプターは、WAR のセキュリティーを提供します。これにより、URL にロールベースのセキュリティー制約を定義できます。

JBoss EAP コンテナー用の Red Hat Single Sign-On アダプターのインストールと使用の詳細については、Red Hat Single Sign-On **Securing Applications and Services Guide** の JBoss EAP Adapter を参照してください。

第8章 LDAP 認証チュートリアル

概要

このチュートリアルでは、X.500 ディレクトリーサーバーを設定し、LDAP 認証を使用するように OSGi コンテナーを設定する方法について説明します。

8.1. チュートリアルの概要

ゴール

このチュートリアルでは、以下を行います。

- 389 Directory Server をインストールする
- LDAP サーバーにユーザーエントリーを追加する
- グループを追加してセキュリティーロールを管理する
- LDAP 認証を使用するように Fuse を設定する
- 認可にロールを使用するように Fuse を設定する
- LDAP サーバーへの SSL/TLS 接続を設定する

8.2. DIRECTORY SERVER とコンソールの設定

チュートリアルのこの段階では、Fedora 389 Directory Server プロジェクトから X.500 Directory Server と管理コンソールをインストールする方法について説明します。389 Directory Server インスタンスにすでにアクセスできる場合は、389 Directory Server をインストールする手順をスキップして、代わりに 389 管理コンソールをインストールできます。

前提条件

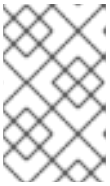
Red Hat Enterprise Linux プラットフォームにインストールする場合は、まず Extra Packages for Enterprise Linux (EPEL) をインストールする必要があります。**fedoraproject.org** サイトの RHEL/Cent OS/ EPEL (RHEL 6, RHEL 7, Cent OS 6, Cent OSý7) にあるインストールノートを参照してください。

389 Directory Server をインストールする

既存の **389 Directory Server** インスタンスにアクセスできない場合は、次のように **389 Directory Server** をローカルマシンにインストールできます。

1. Red Hat Enterprise Linux および Fedora プラットフォームでは、標準の **dnf** パッケージ管理ユーティリティーを使用して **389 Directory Server** をインストールします。コマンドプロンプトで次のコマンドを入力します (マシンの管理者権限が必要です)。

```
sudo dnf install 389-ds
```



注記

必要な **389-ds** および **389-console** RPM パッケージは、Fedora、RHEL6+EPEL、および CentOS7+EPEL のプラットフォームで利用できます。執筆時点では、**389-console** パッケージは RHEL 7 では利用できません。

2. 389 Directory Server パッケージをインストールした後、次のコマンドを入力して Directory Server を設定します。

```
sudo setup-ds-admin.pl
```

スクリプトは対話型であり、389 Directory Server の基本的な設定を指定するように求められます。スクリプトが完了すると、バックグラウンドで 389 Directory Server が自動的に起動されます。

3. **389 Directory Server** のインストール方法について、詳細は ダウンロード ページを参照してください。

389 管理コンソールのインストール

389 Directory Server インスタンスにすでにアクセスできる場合は、389 管理コンソールをインストールするだけで、サーバーにログインしてリモートで管理できます。389 管理コンソールのインストール方法は以下のとおりです。

- **Red Hat Enterprise Linux および Fedora プラットフォームの場合**– 標準の **dnf** パッケージ管理ユーティリティーを使用して、389 Management Console をインストールします。コマンドプロンプトで次のコマンドを入力します (マシンの管理者権限が必要です)。

```
sudo dnf install 389-console
```

- **Windows プラットフォームの場合** - **fedoraproject.org** で Windows Console のダウンロード手順を参照してください。

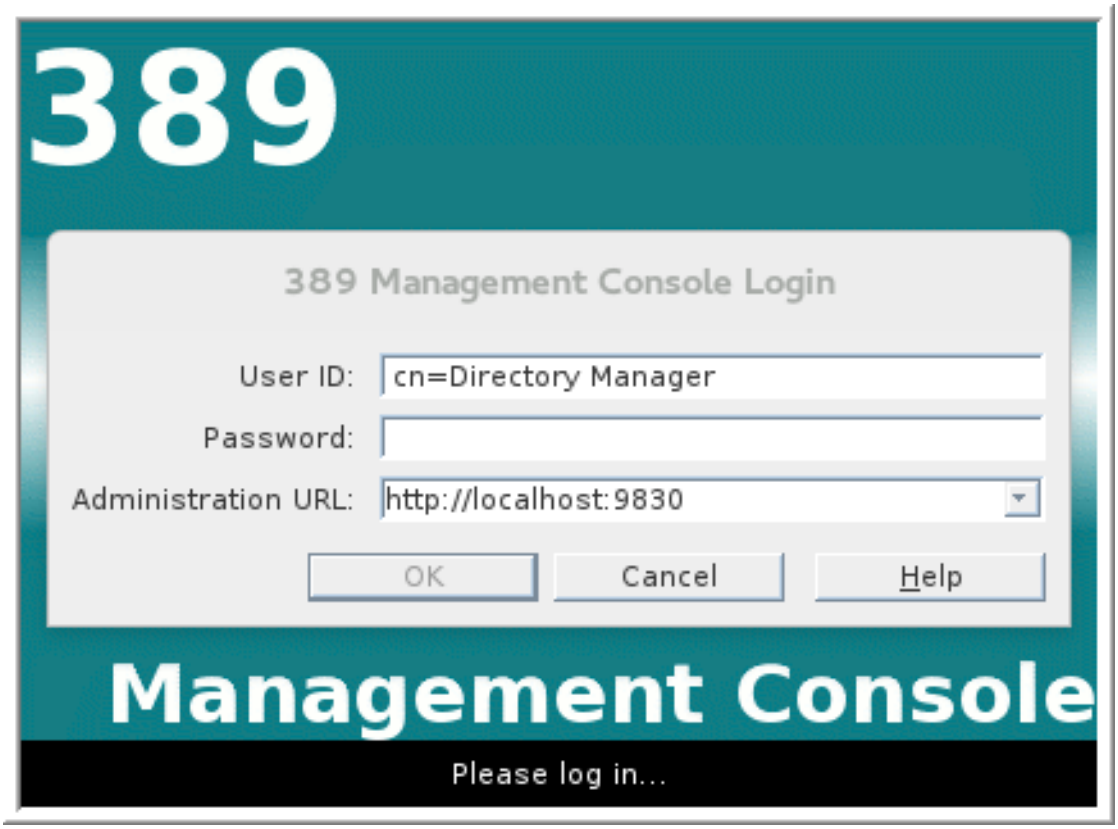
コンソールとサーバーの接続

389 Directory Server コンソールを LDAP サーバーに接続するには、以下を実行します。

1. 次のコマンドを入力して、389 管理コンソールを起動します。

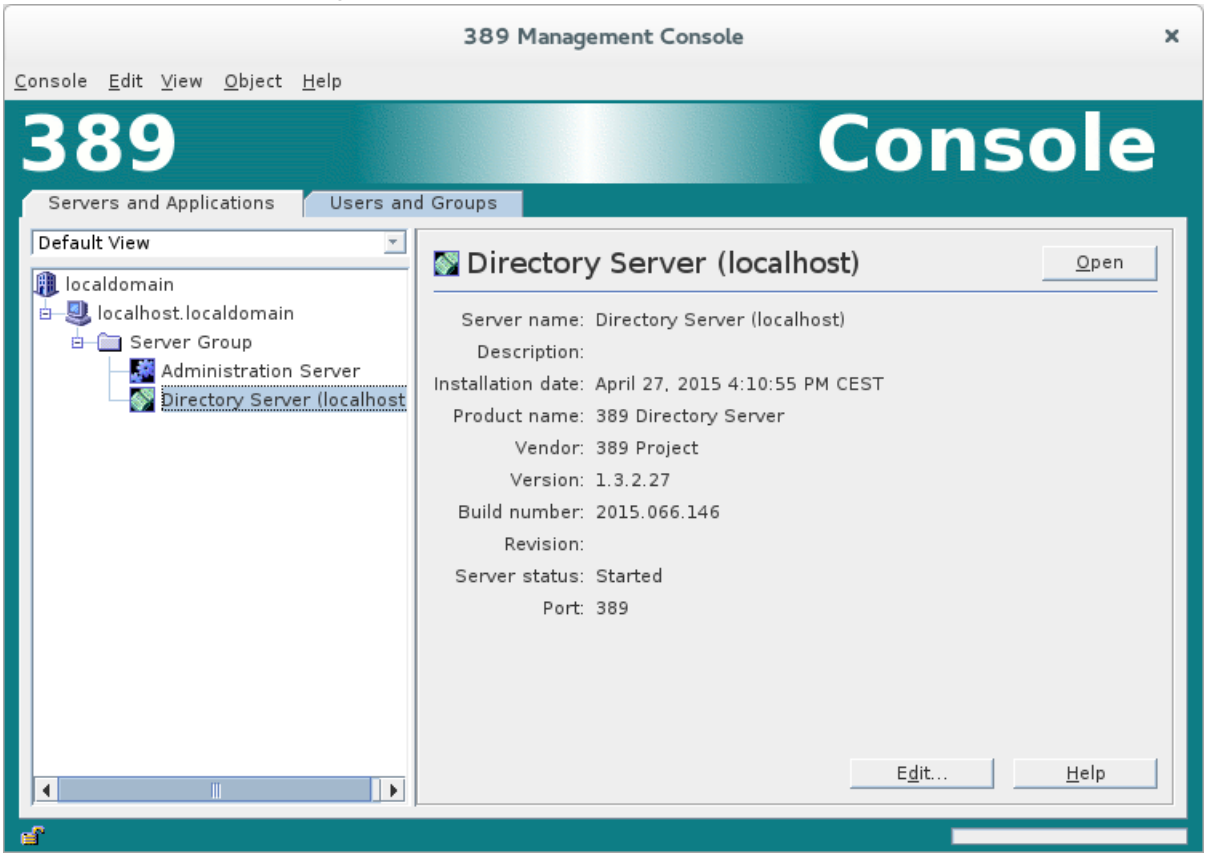
```
389-console
```

2. ログインダイアログが表示されます。**User ID** および **Password** フィールドに LDAP ログインクレデンシャルを入力し、**Administration URL** フィールドのホスト名をカスタマイズして 389 管理サーバーインスタンスに接続します (ポート **9830** は 389 管理サーバーインスタンスのデフォルトポートです)。



3. 389 管理コンソールウィンドウが表示されます。**Servers and Applications** タブを選択します。

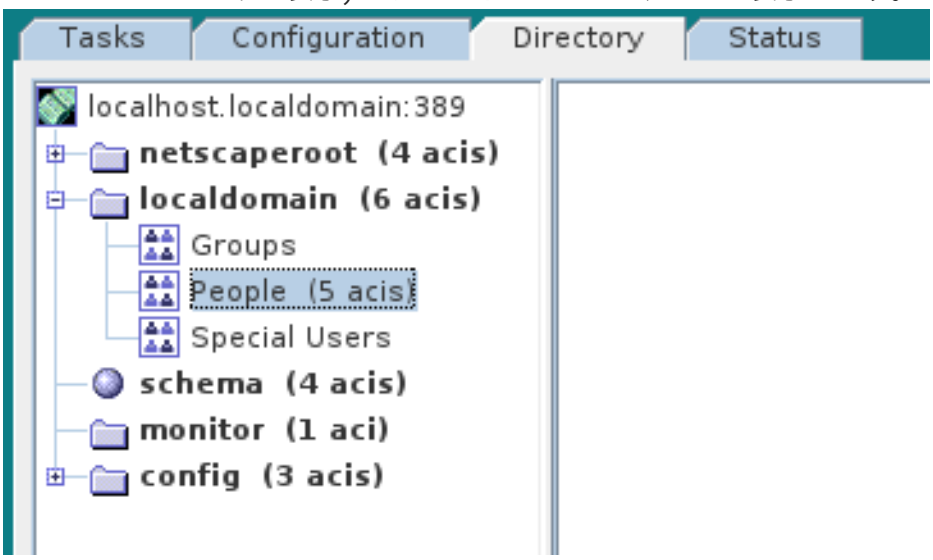
4. 左側のペインで、**Directory Server** アイコンを見つけます。



5. 左側のペインで **Directory Server** アイコンを選択し、**Open** をクリックして、**389 Directory Server Console** を開きます。

6. **389 Directory Server Console** コンソールで、**Directory** タブをクリックして Directory Information Tree (DIT) を表示します。

7. ルートノード **YourDomain** (通常はホスト名にちなんで命名され、次のスクリーンショットの **localdomain** のように表示) をデプロイメントして、DIT を表示します。



8.3. DIRECTORY SERVER へのユーザーエントリーの追加

OSGi コンテナーで LDAP 認証を使用するための基本的な前提条件として、X.500 Directory Server を実行してユーザーエントリーのコレクションで設定する必要があります。多くのユースケースでは、ユーザーのロールを管理するためにいくつかのグループを設定することもできます。

ユーザーエントリーの追加に替わる方法

LDAP サーバーにユーザーエントリーおよびグループがすでに定義されている場合は、新規エントリーを作成する代わりに、**LDAPLoginModule** 設定の **roles.mapping** プロパティーを使用して既存の LDAP グループを JAAS ロールにマッピングした方が好ましい場合があります。詳細は、「JAAS LDAP ログインモジュール」 を参照してください。

ゴール

チュートリアルのこの部分では、以下について説明します。

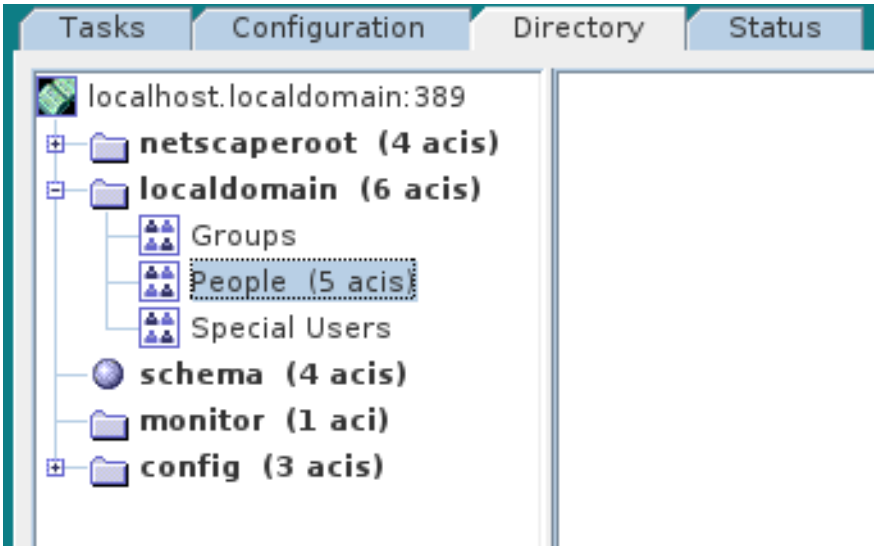
- LDAP サーバーに 3 つのユーザーエントリーを追加する方法
- LDAP サーバーに 4 つのグループを追加する方法

ユーザーエントリーの追加

次の手順を実行して、Directory Server にユーザーエントリーを追加します。

1. LDAP サーバーとコンソールが実行されていることを確認します。「Directory Server とコンソールの設定」 を参照してください。

2. **Directory Server Console** コンソールで、**Directory** タブをクリックして、**YourDomain** ノード下の **People** ノードにドリルダウンします (以下のスクリーンショットでは **YourDomain** が **localdomain** と表示されます)。



3. **People** ノードを右クリックし、コンテキストメニューから menu:[> New > > **User** >] と選択して **Create New User** ダイアログを開きます。

4. **Create New User** ダイアログの左側のペインで **User** タブを選択します。

5. 以下のように **User** タブのフィールドを入力します。

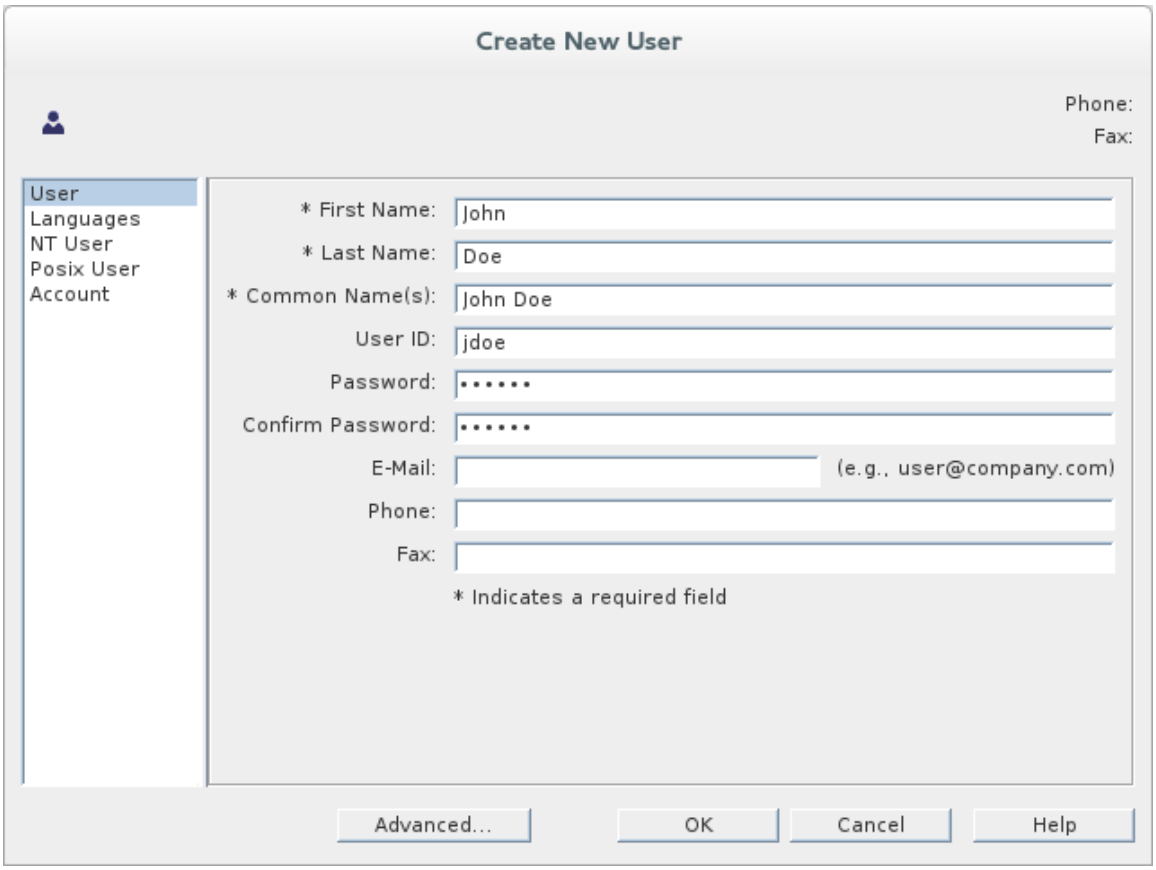
   a. **First Name** フィールドを **John** に設定します。

   b. **Last Name** フィールドを **Doe** に設定します。

   c. **User ID** フィールドを **jdoe** に設定します。

   d. **Password** フィールドに、パスワード **secret** を入力します。

   e. **Confirm Password** フィールドに、パスワード **secret** を入力します。



6. 6 で **OK** をクリックします。

7. ステップ 3 から ステップ に従って、ユーザー **Jane Doe** を追加します。
   ステップ 5.e で、新規ユーザーの **User ID** に **janedoe** を使用し、パスワードフィールドにはパスワード **secret** を使用します。

8. ステップ 3 から ステップ に従って、ユーザー **Camel Rider** を追加します。
   ステップ 5.e で、新規ユーザーの **User ID** に **crider** を使用し、パスワードフィールドにはパスワード **secret** を使用します。

グループにロールを追加

ロールを定義するグループを追加するには、以下を実行します。

1. **Directory Server Console** コンソールの **Directory** タブで、**YourDomain** ノードの下で、**Groups** ノードにドリルダウンします。

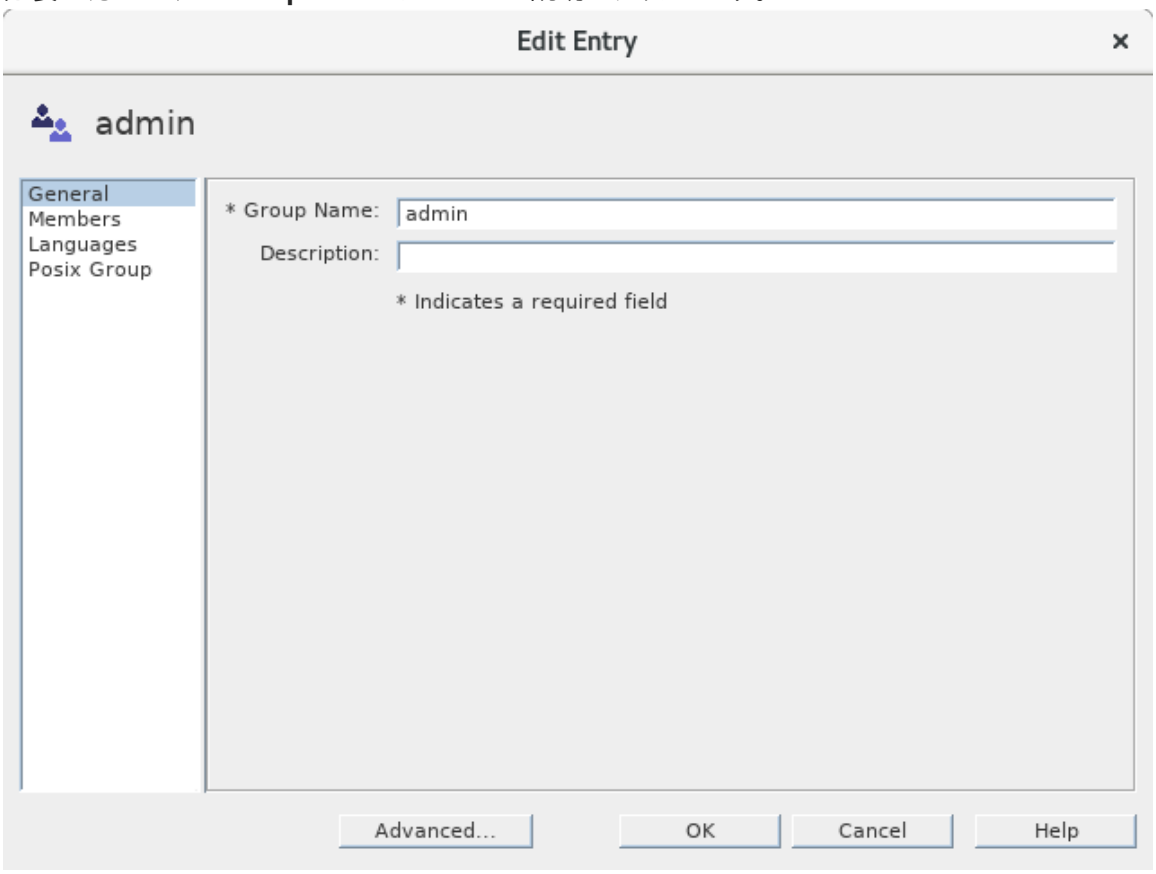
2. **Groups** ノードを右クリックし、コンテキストメニューから menu:[> New > > **Group** >] と選択して **Create New Group** ダイアログを開きます。

3. **Create New Group** ダイアログの左側のペインで **General** タブを選択します。

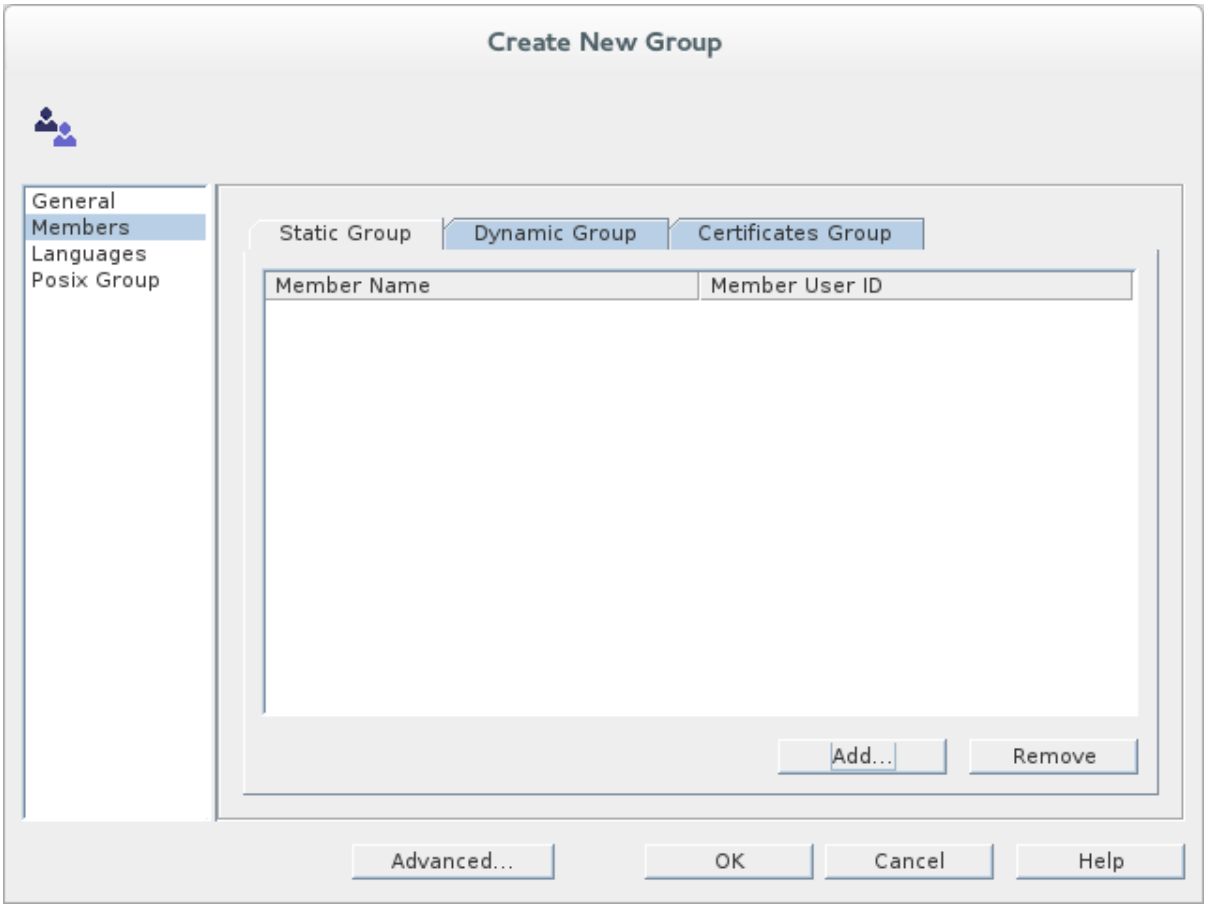
4. 以下のように **General** タブのフィールドを入力します。

   a. **Group Name** フィールドを **admin** に設定します。

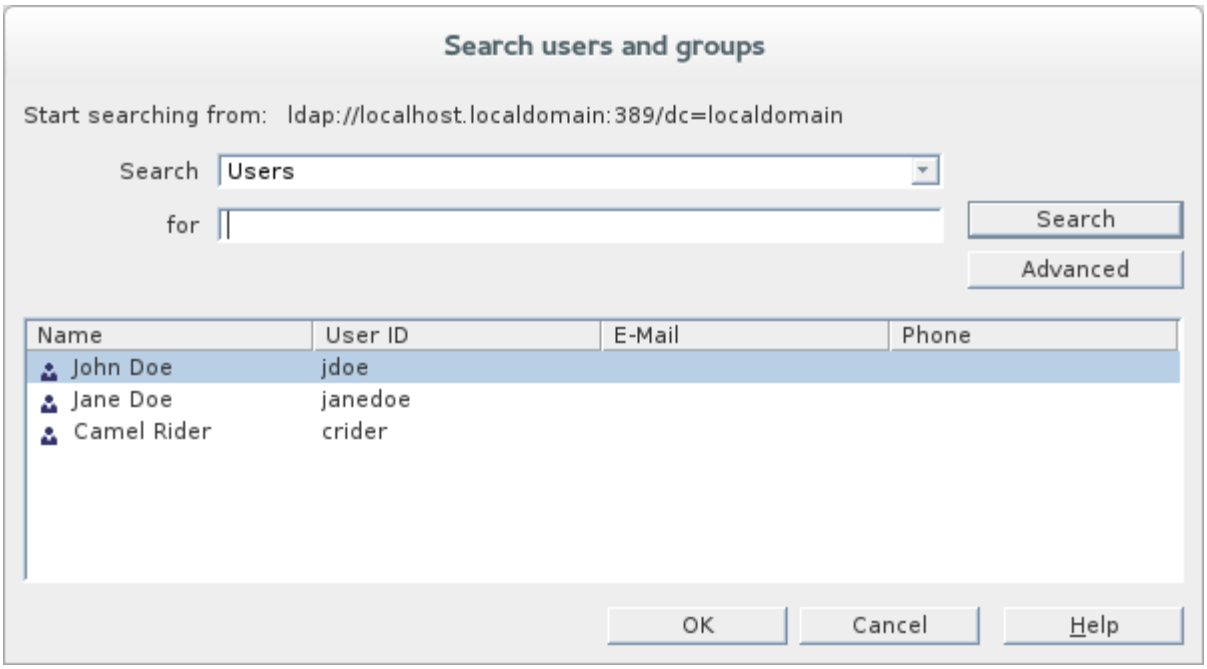
   b. 必要に応じて、**Description** フィールドに説明を入力します。



5. **Create New Group** ダイアログの左側のペインで **Members** タブを選択します。



6. **Add** をクリックして **Search users and groups** ダイアログを開きます。
7. **Search** フィールドで、ドロップダウンメニューから **Users** を選択し、**Search** ボタンをクリックします。



8. 表示されているユーザーのリストから、**John Doe** を選択します。
9. **OK** をクリックして、**Search users and groups** ダイアログを閉じます。
10. **OK** をクリックし、**Create New Group** ダイアログを閉じます。
11. ステップ 2 から ステップ 10 に従って、**manager** ロールを追加します。

ステップ 4 で、**Group Name** フィールドに **manager** と入力します。

ステップ 8 で、**Jane Doe** を選択します。

12. ステップ 2 から ステップ 10 に従って、**viewer** ロールを追加します。
ステップ 4 で、**Group Name** フィールドに **viewer** と入力します。

ステップ 8 で **Camel Rider** を選択します。

13. ステップ 2 から ステップ 10 に従って、**ssh** ロールを追加します。
ステップ 4 で、**Group Name** フィールドに **ssh** と入力します。

ステップ 8 で、**すべて** のユーザー **John Doe**、**Jane Doe**、および **Camel Rider** を選択します。

8.4. OSGI コンテナーでの LDAP 認証の有効化

このセクションでは、OSGi コンテナーで LDAP レルムを設定する方法について説明します。新しいレルムはデフォルトの **karaf** レルムを上書きするため、コンテナーは X.500 ディレクトリーサーバーに保存されているユーザーエントリーを基にしてクレデンシャルを認証します。

参考資料

以下は、LDAP 認証に関する詳細なドキュメントです。

- **LDAPLoginModule オプション** - 詳細については、「JAAS LDAP ログインモジュール」 を参照してください。
- **その他のディレクトリーサーバーの設定** - 本チュートリアルは 389-DS のみを取り上げます。Microsoft Active Directory などの他の Directory Server を設定する方法について、詳細は「さまざまなディレクトリーサーバーのフィルター設定」 を参照してください。

スタンドアロン OSGi コンテナーの手順

スタンドアロン OSGi コンテナーで LDAP 認証を有効にするには、以下を実行します。

1. X.500 Directory Server が実行されていることを確認します。
2. ターミナルウィンドウで次のコマンドを入力し、Karaf コンテナーを起動します。

```
./bin/fuse
```

3. **ldap-module.xml** という名前のファイルを作成します。
4. 例8.1「スタンドアロン用の JAAS レルム」 を **ldap-module.xml** にコピーします。

例8.1 スタンドアロン用の JAAS レルム

```
<?xml version="2.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0">

  <jaas:config name="karaf" rank="200">
    <jaas:module className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
```

```
            flags="required">
    initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
    connection.url=ldap://localhost:389
    connection.username=cn=Directory Manager
    connection.password=DIRECTORY_MANAGER_PASSWORD
    connection.protocol=
    user.base.dn=ou=People,dc=localdomain
    user.filter=(&amp;(objectClass=inetOrgPerson)(uid=%u))
    user.search.subtree=true
    role.base.dn=ou=Groups,dc=localdomain
    role.name.attribute=cn
    role.filter=(uniquemember=%fqdn)
    role.search.subtree=true
    authentication=simple
  </jaas:module>
 </jaas:config>
</blueprint>
```

ldap-module.xml ファイルで以下の設定をカスタマイズする必要があります。

connection.url

この URL を Directory Server インスタンスの実際の場所に設定します。通常、この URL の形式は **ldap://Hostname:Port** です。たとえば、389 Directory Server のデフォルトポートは IP ポート **389** です。

connection.username

Directory Server への接続を認証するために使用されるユーザー名を指定します。389 Directory Server の場合、デフォルトは通常 **cn=Directory Manager** です。

connection.password

Directory Server への接続に使用するクレデンシャルのパスワード部分を指定します。

認証

認証プロトコルには、次のいずれかの選択肢を指定できます。

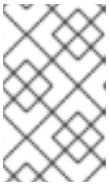
- **simple** の場合、ユーザークレデンシャルが提供され、**connection.username** オプションおよび **connection.password** オプションを設定する義務があることを意味します。
- **none** の場合、認証が**行われない**ことを意味します。この場合、**connection.username** および **connection.password** オプションを設定しないでください。
  このログインモジュールは、**karaf** という名前の JAAS レルムを作成します。これは、Fuse によって使用されるデフォルトの JAAS レルムと同じ名前です。**0** より大きい **rank** 属性の値でこのレルムを再定義すると、ランク **0** を持つ標準 **karaf** レルムがオーバーライドされます。

  LDAP を使用するように Fuse を設定する方法について、詳細は 「JAAS LDAP ログインモジュール」 を参照してください。

重要

上記の JAAS プロパティーを設定するときは、プロパティー値を二重引用符で **囲まない** でください。

5. 新しい LDAP モジュールをデプロイするには、**ldap-module.xml** を Karaf コンテナーの **deploy**/ ディレクトリー (ホットデプロイ) にコピーします。
LDAP モジュールは自動的にアクティブ化されます。



注記

その後、LDAP モジュールをアンデプロイする必要がある場合は、**Karaf コンテナーの実行中** に **ldap-module.xml** ファイルを **deploy**/ ディレクトリーから削除することで実行できます。

LDAP 認証のテスト

以下のように Karaf **client** ユーティリティーを使用して実行中のコンテナーに接続し、新しい LDAP レルムをテストします。

1. 新しいコマンドプロンプトを開きます。
2. Karaf **InstallDir**/**bin** ディレクトリーに移動します。
3. 以下のコマンドを入力し、ID **jdoe** を使用して実行中のコンテナーインスタンスにログインします。

```
./client -u jdoe -p secret
```

コンテナーのリモートコンソールに正常にログインする必要があります。コマンドコンソールで、**jaas:** と入力した後に [Tab] キーを押します (コンテンツ補完を使用)。

```
jdoe@root()> jaas:
Display all 31 possibilities? (31 lines)?
jaas:cancel
jaas:group-add
...
jaas:whoami
```

jdoe がすべての **jaas** コマンド (**admin** と一致) にアクセスできることが確認できるはずです。

4. **logout** コマンドを入力して、リモートコンソールからログアウトします。
5. 以下のコマンドを入力し、ID **janedoe** を使用して実行中のコンテナーインスタンスにログインします。

```
./client -u janedoe -p secret
```

コンテナーのリモートコンソールに正常にログインする必要があります。コマンドコンソールで、**jaas:** と入力した後に [Tab] キーを押します (コンテンツ補完を使用)。

```
janedoe@root()> jaas:
Display all 25 possibilities? (25 lines)?
jaas:cancel
jaas:group-add
...
jaas:users
```

janedoe がほぼすべての **jaas** コマンド (**manager** と一致) にアクセスできることが確認できるはずです。

6. **logout** コマンドを入力して、リモートコンソールからログアウトします。
7. 以下のコマンドを入力し、ID **crider** を使用して実行中のコンテナーインスタンスにログインします。

```
./client -u crider -p secret
```

コンテナーのリモートコンソールに正常にログインする必要があります。コマンドコンソールで、**jaas:** と入力した後に [Tab] キーを押します (コンテンツ補完を使用)。

```
crider@root()> jaas:
jaas:manage
jaas:realm-list
jaas:realm-manage
jaas:realms
jaas:user-list
jaas:users
```

crider は 5 つの **jaas** コマンド (**viewer** ロールと一致) のみにアクセスできることが確認できるはずです。

8. **logout** コマンドを入力して、リモートコンソールからログアウトします。

トラブルシューティング

LDAP 接続のテスト中に問題が発生した場合は、ログレベルを **DEBUG** に引き上げ、LDAP サーバーへの接続で何が起こっているかを詳細にトレースします。

以下の手順を実行します。

1. Karaf コンソールから以下のコマンドを入力し、ログレベルを **DEBUG** に引き上げます。

```
log:set DEBUG
```

2. Karaf ログをリアルタイムで観察します。

```
log:tail
```

ログリストからエスケープするには、Ctrl-C を入力します。

付録A 証明書の管理

概要

TLS 認証は、アプリケーションオブジェクトを認証する一般的なセキュアで信頼性の高い X.509 証明書を使用します。Red Hat Fuse アプリケーションを識別する X.509 証明書を作成できます。

A.1. X.509 証明書とは

証明書のロール

X.509 証明書は、名前を公開鍵の値にバインドします。証明書のロールは、公開鍵を X.509 証明書に含まれる ID に関連付けることです。

公開鍵の整合性

セキュアなアプリケーションの認証は、アプリケーションの証明書の公開鍵値の整合性によって異なります。公開鍵を独自の公開鍵に置き換える場合は、true アプリケーションの権限を借用し、セキュアなデータにアクセスできます。

この種の攻撃を防ぐには、すべての証明書を **認証局 (CA)** で署名する必要があります。CA は、証明書の公開鍵値の整合性を確認する信頼できるノードです。

デジタル署名

CA は、**デジタル署名**を証明書に追加して証明書に署名 します。デジタル署名は、CA の秘密鍵でエンコードされたメッセージです。CA の公開鍵は、CA の証明書を配布することでアプリケーションで利用できます。アプリケーションは、CA の公開鍵を使用して CA のデジタル署名をデコードして、証明書が有効で署名されていることを確認します。

警告

提供されるデモ証明書は自己署名証明書です。これらの証明書は、すべてのユーザーが秘密鍵にアクセスできるため、安全ではありません。システムのセキュリティーを保護するには、信頼される CA によって署名された新しい証明書を作成する必要があります。

X.509 証明書の内容

X.509 証明書には、証明書のサブジェクトと証明書の発行者 (証明書を発行した CA) に関する情報が含まれています。証明書は、ネットワーク上で送受信できるメッセージを記述するための標準構文である Abstract Syntax Notation One (ASN.1) でエンコードされます。

証明書のロールは、アイデンティティーを公開鍵の値に関連付けることです。詳細は、証明書には以下が含まれます。

- 証明書の所有者を識別する **サブジェクト識別名 (DN)**
- 発行先に関連付けられた **公開鍵**。

- X.509 バージョン情報。
- 証明書を一意に識別する **シリアル番号**
- 証明書を発行した CA を識別する**発行者 DN**。
- 発行者のデジタル署名。
- 証明書の署名に使用されるアルゴリズムに関する情報。
- オプションの X.509 v.3 拡張の一部。たとえば、CA 証明書とエンドエンティティー証明書を区別する拡張機能が存在します。

識別名

DN は、セキュリティーのコンテキストで使用される汎用 X.500 識別子です。

DN の詳細は、付録B *ASN.1 および識別名* を参照してください。

A.2. 認証局

A.2.1. 認証局の概要

CA は、証明書を生成および管理するツールセットと、生成されたすべての証明書が含まれるデータベースで設定されます。システムを設定する際には、要件に十分に安全である適切な CA を選択することが重要です。

使用できる CA には、以下の 2 つのタイプがあります。

- 商用 CA は、多くのシステムの証明書に署名する企業です。
- プライベート CA は、システムで証明書をセットアップし、使用する信頼されたノードです。

A.2.2. 商用認証局

署名証明書

利用可能な商用 CA が複数存在します。商用 CA を使用して証明書に署名するメカニズムは、選択した CA によって異なります。

商用 CA の利点

商用 CA の利点は、多くの場合、多数のユーザーが信頼していることです。お使いのアプリケーションが組織外のシステムで使用できるように設計されている場合は、商用 CA を使用して証明書に署名します。アプリケーションが内部ネットワーク内で使用されている場合には、プライベート CA が適切である可能性があります。

CA を選択するための基準

商用 CA を選択する前に、以下の基準を考慮してください。

- 商用 CA の証明書署名ポリシーとは
- アプリケーションは内部ネットワークでのみ利用できるように設計されているか？

- プライベート CA の設定にかかる時間は、商用 CA にサブスクライブするコストと比較してどうでしょうか。

A.2.3. プライベート認証局

CA ソフトウェアパッケージの選択

システムの証明書の署名を行う責任がある場合は、プライベート CA を設定します。プライベート CA を設定するには、証明書を作成および署名するためのユーティリティーを提供するソフトウェアパッケージへのアクセスが必要です。このタイプのパッケージが複数利用可能です。

OpenSSL ソフトウェアパッケージ

プライベート CA の設定を可能にするソフトウェアパッケージの 1 つが OpenSSL http://www.openssl.org です。OpenSSL パッケージには、証明書を生成および署名するための基本的なコマンドラインユーティリティーが含まれています。OpenSSL コマンドラインユーティリティーの完全なドキュメントは、http://www.openssl.org/docs から入手できます。

OpenSSL を使用したプライベート CA の設定

プライベート CA を設定するには、「独自の証明書の作成」 の手順を参照してください。

プライベート認証局のホストの選択

プライベート CA を設定する上では、ホストの選択が重要な手順になります。CA ホストに関連付けられるセキュリティーのレベルは、CA によって署名された証明書に関連する信頼レベルを決定します。

Red Hat Fuse アプリケーションの開発およびテストで使用する CA を設定する場合は、アプリケーション開発者がアクセスできるホストを使用します。ただし、CA 証明書と秘密鍵を作成する場合は、セキュリティーが重要なアプリケーションを実行するホストで CA 秘密鍵を利用できるようにしないでください。

セキュリティーの予防措置

デプロイするアプリケーションの証明書に署名するために CA を設定する場合は、CA ホストをできるだけセキュアにします。たとえば、CA を保護するには、以下の点に注意してください。

- CA をネットワークに接続しないでください。
- CA へのアクセスを、信頼できるユーザーの制限されたセットに制限します。
- RF-shield を使用して、ラジオボタン頻度から CA を保護します。

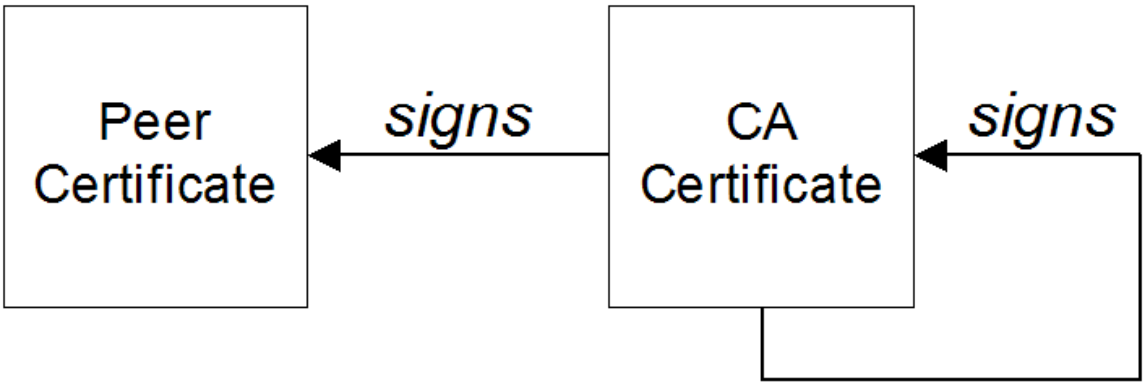
A.3. 証明書チェーン

証明書チェーン

証明書チェーン は証明書のシーケンスであり、チェーンの各証明書は後続の証明書で署名されます。

図A.1「深さ 2 の証明書チェーン」 は、簡単な証明書チェーンの例を示しています。

図A.1 深さ 2 の証明書チェーン



自己署名証明書

チェーンの最後の証明書は通常、自身を署名する **自己署名** です。

信頼チェーン

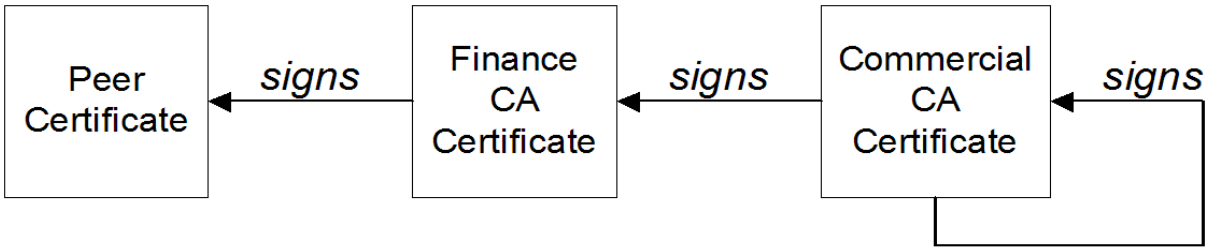
証明書チェーンの目的は、ピア証明書から信頼できる CA 証明書への信頼チェーンを確立することです。CA は、ピア証明書に署名することにより、その ID を保証します。CA が信頼できる CA である場合 (ルート証明書ディレクトリーに CA 証明書のコピーが存在することで示されます)、これは署名されたピア証明書も信頼できることを意味します。

複数の CA で署名された証明書

CA 証明書は別の CA で署名できます。たとえば、Progress Software の財務部門の CA がアプリケーション証明書に署名し、Progress Software が自己署名の商用 CA によって署名される場合があります。

図A.2「深さ 3 の証明書チェーン」 は、証明書チェーンがどのように見えるかを示します。

図A.2 深さ 3 の証明書チェーン



信頼できる CA

アプリケーションは、署名チェーン内の CA 証明書の少なくとも 1 つを信頼する場合、ピア証明書を受け入れることができます。

A.4. HTTPS 証明書の特別な要件

概要

HTTPS 仕様では、HTTPS クライアントがサーバーの ID を検証できる必要があることが義務付けられ

ています。これにより、X.509 証明書の生成方法に影響を及ぼす可能性があります。サーバー ID を検証するメカニズムは、クライアントのタイプによって異なります。一部のクライアントは、特定の信頼できる CA によって署名されたサーバー証明書のみを受け入れることによってサーバー ID を確認する場合があります。さらに、クライアントはサーバー証明書の内容を検査し、特定の制約を満たす証明書のみを受け入れることができます。

アプリケーション固有のメカニズムがない場合、HTTPS 仕様では、サーバー ID を検証するための **HTTPS URL 整合性チェック** と呼ばれる一般的なメカニズムが定義されています。これは、Web ブラウザーが使用する標準メカニズムです。

HTTPS URL 整合性チェック

URL 整合性チェックの基本的な概念は、サーバー証明書のアイデンティティーがサーバーのホスト名と一致する必要があることです。この整合性チェックは、HTTPS 用の X.509 証明書の生成方法に重要な影響を及ぼします。**証明書 ID(通常は証明書サブジェクト DN の共通名) は、HTTPS サーバーがデプロイメントされているホスト名と一致する必要があります**。

URL 整合性チェックは、**中間者攻撃** を防ぐように設計されています。

参照

HTTPS URL 整合性チェックは RFC 2818 で指定され、http://www.ietf.org/rfc/rfc2818.txt の Internet Engineering Task Force (IETF) により公開されます。

証明書アイデンティティーの指定方法

URL 整合性チェックで使用される証明書アイデンティティーは、以下のいずれかの方法で指定できます。

- commonName の使用
- subectAltName の使用

commonName の使用

(URL 整合性チェックの目的で) 証明書 ID を指定する通常の方法は、証明書のサブジェクト DN の共通名 (CN) を使用することです。

たとえば、サーバーが以下の URL でセキュアな TLS 接続をサポートする場合:

```
https://www.redhat.com/secure
```

対応するサーバー証明書には、以下のサブジェクト DN があります。

```
C=IE,ST=Co. Dublin,L=Dublin,O=RedHat,
OU=System,CN=www.redhat.com
```

ここで、CN はホスト名 **www.redhat.com** に設定されています。

新しい証明書にサブジェクト DN を設定する方法は、「独自の証明書の作成」 を参照してください。

subjectAltName の使用 (マルチホームホスト)

証明書 ID にサブジェクト DN の共通名を使用すると、一度に **1つの** ホスト名しか指定できないという

欠点があります。ただし、マルチホームホストに証明書をデプロイメントする場合は、**任意** のマルチホームホスト名で証明書を使用できるようにすることが実用的である場合があります。この場合、複数の代替 ID を使用して証明書を定義する必要があります。これは、**subjectAltName** 証明書エクステンションを使用する場合に限り可能です。

たとえば、次のいずれかのホスト名への接続をサポートするマルチホームホストがある場合:

```
www.redhat.com
www.jboss.org
```

次に、これらの DNS ホスト名の両方を明示的にリスト表示する **subjectAltName** を定義できます。**openssl** ユーティリティーを使用して証明書を生成する場合は、以下のように **openssl.cnf** 設定ファイルに関連する行を編集し、**subjectAltName** エクステンションの値を指定します。

```
subjectAltName=DNS:www.redhat.com,DNS:www.jboss.org
```

ここで、HTTPS プロトコルは、**subjectAltName** にリスト表示されている DNS ホスト名のいずれかに対してサーバーホスト名が一致します (**subjectAltName** は共通名よりも優先されます)。

HTTPS プロトコルは、ホスト名のワイルドカード文字 (*) もサポートしています。たとえば、以下のように **subjectAltName** を定義できます。

```
subjectAltName=DNS:*.jboss.org
```

この証明書 ID は、ドメイン **jboss.org** 内の任意の 3 コンポーネントホスト名と一致します。

警告

ドメイン名にワイルドカード文字を **使用しないでください** (ドメイン名の前にドット **.** の区切り文字を入力し忘れて、誤って使用しないように注意する必要があります)。たとえば、***jboss.org** を指定した場合、証明書は **jboss** 文字で終わる *任意* のドメインで使用できます。

A.5. 独自の証明書の作成

概要

この章では、独自のプライベート認証局 (CA) を設定し、この CA を使用して独自の証明書を生成および署名するための手法と手順について説明します。

警告

独自の証明書を作成および管理するには、セキュリティーに関する専門知識が必要です。この章で説明する手順を使用すると、デモンストレーションおよびテスト環境用に独自の証明書を手軽に生成できますが、実稼働環境でこれらの証明書を使用することは **推奨しません**。

A.5.1. OpenSSL ユーティリティーのインストール

RHEL および Fedora プラットフォームへの OpenSSL のインストール

Red Hat Enterprise Linux (RHEL) 5 と 6、および Fedora プラットフォームでは、RPM パッケージとして利用できます。OpenSSL をインストールするには、次のコマンドを入力します (管理者権限で実行)。

```
yum install openssl
```

ソースコードのディストリビューション

OpenSSL のソースディストリビューションは、http://www.openssl.org/docs にあります。OpenSSL プロジェクトは、ソースコードのディストリビューション **のみ** て提供しています。OpenSSL Web サイトから OpenSSL ユーティリティーのバイナリーインストールをダウンロードすることはできません。

A.5.2. プライベート認証局の設定

概要

プライベート CA を使用する場合は、アプリケーションで使用する独自の証明書を生成する必要があります。OpenSSL プロジェクトは、プライベート CA を設定し、署名付き証明書を作成し、CA を Java キーストアに追加するための無料コマンドラインユーティリティーを提供します。

警告

実稼働環境用にプライベート CA を設定するには、高度な専門知識が必要であり、外部の脅威から証明書ストアを保護するために特別な注意を払う必要があります。

プライベート認証局の設定手順

独自のプライベート認証局を設定するには、以下を実行します。

1. 次のように、CA のディレクトリー構造を作成します。

```
X509CA/demoCA
```

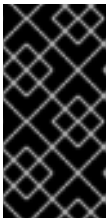
X509CA/demoCA/private
X509CA/demoCA/certs
X509CA/demoCA/newcerts
X509CA/demoCA/crl

2. テキストエディターを使用して、**X509CA/openssl.cfg** ファイルを作成し、以下の内容をこのファイルに追加します。

例A.1 OpenSSL 設定

```
#
# SSLeay example configuration file.
# This is mostly being used for generation of certificate requests.
#

RANDFILE            = ./.rnd

####################################################################
[ req ]
default_bits        = 2048
default_keyfile     = keySS.pem
distinguished_name  = req_distinguished_name
encrypt_rsa_key     = yes
default_md          = sha1

[ req_distinguished_name ]
countryName         = Country Name (2 letter code)

organizationName    = Organization Name (eg, company)

commonName          = Common Name (eg, YOUR name)

####################################################################
[ ca ]
default_ca          = CA_default        # The default ca section

####################################################################
[ CA_default ]

dir                 = ./demoCA              # Where everything is kept
certs               = $dir/certs            # Where the issued certs are kept
crl_dir             = $dir/crl              # Where the issued crl are kept
database            = $dir/index.txt        # database index file.
#unique_subject     = no                    # Set to 'no' to allow creation of
                                            # several certificates with same subject.
new_certs_dir       = $dir/newcerts         # default place for new certs.

certificate         = $dir/cacert.pem       # The CA certificate
serial              = $dir/serial           # The current serial number
crl                 = $dir/crl.pem          # The current CRL
private_key         = $dir/private/cakey.pem# The private key
RANDFILE            = $dir/private/.rand    # private random number file

name_opt            = ca_default            # Subject Name options
cert_opt            = ca_default            # Certificate field options
```

```
default_days      = 365                # how long to certify for
default_crl_days  = 30                 # how long before next CRL
default_md        = md5                # which md to use.
preserve          = no                 # keep passed DN ordering

policy            = policy_anything

[ policy_anything ]
countryName            = optional
stateOrProvinceName    = optional
localityName           = optional
organizationName       = optional
organizationalUnitName = optional
commonName             = supplied
emailAddress           = optional
```



重要

前述の **openssl.cfg** 設定ファイルは、**デモンストレーションとしてのみ**提供されます。実稼働環境では、この設定ファイルは、高度なセキュリティー専門知識を持つエンジニアが慎重に作成し、進化するセキュリティーの脅威から保護するために積極的に保守する必要があります。

3. **demoCA/serial** ファイルを初期化します。このファイルには、初期のコンテンツ **01** が必要です。以下のコマンドを入力します。

```
echo 01 > demoCA/serial
```

4. **demoCA/index.txt** を初期化します。最初は空である**必要があります**。以下のコマンドを入力します。

```
touch demoCA/index.txt
```

5. 以下のコマンドを使用して、新しい自己署名 CA 証明書と秘密鍵を作成します。

```
openssl req -x509 -new -config openssl.cfg -days 365 -out demoCA/cacert.pem -keyout
demoCA/private/cakey.pem
```

例A.2「CA 証明書の作成」 に示されるとおり、CA 秘密鍵のパスフレーズと CA 識別名の詳細の入力を求められます。

例A.2 CA 証明書の作成

```
Generating a 2048 bit RSA private key
.....................................................................+++
.................+++
writing new private key to 'demoCA/private/cakey.pem'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
```

```
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) []:DE
Organization Name (eg, company) []:Red Hat
Common Name (eg, YOUR name) []:Scooby Doo
```

注記

CA のセキュリティーは、秘密鍵ファイルと、この手順で使用する秘密鍵のパスフレーズによって異なります。

CA 証明書と秘密鍵 (**cacert.pem** および **cakey.pem**) のファイル名と場所が、**openssl.cfg** で指定した値と同じであることを確認してください (前述の手順を参照)。

A.5.3. CA トラストストアファイルの作成

概要

サーバーの ID を確認するために、SSL/TLS 接続のクライアント側でトラストストアファイルが一般的に必要です。トラストストアファイルを使用して、デジタル署名を確認することもできます (たとえば、トラストストアファイル内の信頼できる証明書の 1 つに対応する秘密鍵を使用して署名が作成されたことを確認する)。

CA トラストストアの作成手順

1 つ以上の CA 証明書をトラストストアファイルに追加するには、以下を実行します。

1. デプロイする信頼される CA 証明書のコレクションをアセンブルします。
   信頼できる CA 証明書は、パブリック CA またはプライベート CA から取得できます。信頼できる CA 証明書は、Java **keystore** ユーティリティー (例: PEM 形式) と互換性のある任意の形式にすることができます。必要なのは証明書だけです。秘密鍵とパスワードは **必要ありません**。

2. **keytool -import** コマンドを使用して、CA 証明書をトラストストアに追加します。
   以下のコマンドを入力して、PEM 形式の CA 証明書 **cacert.pem** を JKS トラストストアに追加します。

   ```
   keytool -import -file cacert.pem -alias CAAlias -keystore truststore.ts -storepass StorePass
   ```

   truststore.ts は、CA 証明書が含まれるキーストアファイルです。このファイルが存在しない場合は、**keytool** コマンドにより作成されます。**CAAlias** は、インポートされた CA 証明書の便利な識別子です。**StorePass** はキーストアファイルへのアクセスに必要なパスワードです。

3. 前の手順を繰り返して、すべての CA 証明書をトラストストアに追加します。

A.5.4. 新しい証明書の生成と署名

概要

証明書を実用するためには、証明書の信頼性を保証する CA によって署名されている必要があります。これにより、単一の CA 証明書を使用して多数の証明書を検証できるため、証明書検証のスケーラブルなソリューションが容易になります。

新しい証明書の生成および署名手順

独自のプライベート CA を使用して新しい証明書を生成し、署名するには、次の手順を実行します。

1. 以下のように、**keytool -genkeypair** コマンドを使用して、証明書と秘密鍵のペアを生成します。

   ```
   keytool -genkeypair -keyalg RSA -dname "CN=Alice, OU=Engineering, O=Red Hat,
   ST=Dublin, C=IE" -validity 365 -alias alice -keypass KeyPass -keystore alice.ks -storepass
   StorePass
   ```

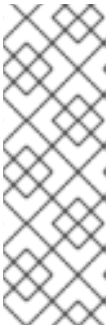
   指定のキーストア **alice.ks** はコマンドを実行する前に存在していなかったため、暗黙的に新しいキーストアが作成され、パスワードが **StorePass** に設定されます。

   -dname および **-validity** フラグは、新たに作成された X.509 証明書の内容を定義します。

   注記

   証明書の識別名 (**-dname** パラメーターを使用した) を指定する場合は、**openssl.cfg** ファイルで指定されたポリシー制約を必ず守る必要があります。これらのポリシー制約に従わない場合は、(次の手順で) CA を使用して証明書に署名することはできません。

   注記

   -keyalg RSA オプション (または同様の強度の鍵アルゴリズム) を使用してキーペアを生成することが不可欠です。デフォルトのキーアルゴリズムは、DSA 暗号化と SHA-1 署名の組み合わせを使用します。しかし、SHA-1 アルゴリズムは十分にセキュアであるとはみなされなくなっており、最新の Web ブラウザーは SHA-1 を使用して署名された証明書を拒否します。RSA 鍵アルゴリズムを選択すると、**keytool** ユーティリティーは代わりに SHA-2 アルゴリズムを使用します。

2. **keystore -certreq** コマンドを使用して、証明書署名要求を作成します。
   以下のように、**alice.ks** 証明書の新規の証明書署名要求を作成し、**alice_csr.pem** ファイルにエクスポートします。

   ```
   keytool -certreq -alias alice -file alice_csr.pem -keypass KeyPass -keystore alice.ks -
   storepass StorePass
   ```

3. **openssl ca** コマンドを使用して CSR に署名します。
   次のように、プライベート CA を使用して Alice 証明書の CSR に署名します。

   ```
   openssl ca -config openssl.cfg -days 365 -in alice_csr.pem -out alice_signed.pem
   ```

   「プライベート認証局の設定手順」 で CA 作成した際に使用した CA 秘密鍵パスフレーズを入力するように求められます。

   openssl ca コマンドについての詳細は、http://www.openssl.org/docs/apps/ca.html# を参照してください。

4. **-outform** オプションが **PEM** に設定された **openssl x509** コマンドを使用して、署名付き証明書を PEM 専用形式に変換します。以下のコマンドを入力します。

```
openssl x509 -in alice_signed.pem -out alice_signed.pem -outform PEM
```

5. CA 証明書ファイルと変換された署名付き証明書ファイルを連結し、証明書チェーンを形成します。たとえば、Linux および UNIX プラットフォームでは、以下のように CA 証明書ファイルと署名済み Alice 証明書 **alice_signed.pem** を連結できます。

```
cat demoCA/cacert.pem alice_signed.pem > alice.chain
```

6. **keytool -import** コマンドを使用して、新しい証明書の完全な証明書チェーンを Java キーストアにインポートします。以下のコマンドを入力します。

```
keytool -import -file alice.chain -keypass KeyPass -keystore alice.ks -storepass StorePass
```

付録B ASN.1 および識別名

概要

OSI Abstract Syntax Notation One (ASN.1) および X.500 Distinguished Names は、X.509 証明書および LDAP ディレクトリーを定義するセキュリティー標準で重要なロールを果たします。

B.1. ASN.1

概要

Abstract Syntax Notation One (ASN.1) は、特定のマシンハードウェアやプログラミング言語に依存しないデータ型と構造を定義する方法を提供するために、1980 年代初頭に OSI 標準化団体によって定義されました。多くの点で、ASN.1 は、プラットフォームに依存しないデータ型の定義に関係する OMG の IDL や WSDL などの最新のインターフェイス定義言語の先駆けと見なすことができます。

ASN.1 は、標準 (SNMP、X.509、LDAP など) の定義で広く使用されているため、重要です。特に、ASN.1 はセキュリティー標準の分野で広く普及しています。X.509 証明書と識別名の正式な定義は、ASN.1 構文を使用して記述されています。これらのセキュリティー標準を使用するために ASN.1 構文の詳細な知識は必要ありませんが、ほとんどのセキュリティー関連データ型の基本的な定義には ASN.1 が使用されることに注意する必要があります。

BER

OSI の基本符号化規則 (BER) は、ASN.1 データ型をオクテットのシーケンス (バイナリー表現) に変換する方法を定義します。したがって、ASN.1 に関して BER が果たすロールは、OMGIDL に関して GIOP が果たすロールと同様です。

DER

OSI の Distinguished Encoding Rules (DER) は、BER に特化したものです。DER は、BER と、エンコーディングが一意であることを保証するためのいくつかの追加ルールで設定されています (BER エンコーディングは一意ではありません)。

参考資料

ASN.1 の詳細については、次の標準ドキュメントを参照してください。

- ASN.1 は X.208 で定義されています。
- BER は X.209 で定義されています。

B.2. 識別名

概要

歴史的に、識別名 (DN) は、X.500 ディレクトリー構造のプライマリーキーとして定義されています。ただし、DN は、他の多くのコンテキストで汎用識別子として使用されるようになりました。Apache CXF では、DN は次のコンテキストで発生します。

- X.509 証明書 - たとえば、証明書内の DN の 1 つは、証明書の所有者 (セキュリティープリンシパル) を識別します。

- LDAP: DN は、LDAP ディレクトリーツリー内のオブジェクトを見つけるために使用されます。

DN の文字列表現

DN は ASN.1 で正式に定義されていますが、DN の UTF-8 文字列表現を定義する LDAP 標準もあります (**RFC 2253** を参照)。文字列表現は、DN の構造を記述するための便利な基礎を提供します。

注記

DN の文字列表現は、DER でエンコードされた DN の一意の表現を提供**しません**。そのため、文字列形式から DER 形式に戻される DN は、常に元の DER エンコーディングを復元する訳ではありません。

DN 文字列の例

以下の文字列は、DN の典型的な例です。

```
C=US,O=IONA Technologies,OU=Engineering,CN=A. N. Other
```

DN 文字列の構造

DN 文字列は、以下の基本要素から構築されます。

- OID
- 属性タイプ
- AVA
- RDN

OID

OBJECT IDENTIFIER (OID) は、ASN.1 の文法構造を一意に識別するバイトのシーケンスです。

属性タイプ

DN に表示される可能性のあるさまざまな属性タイプは、理論的には制限がありませんが、実際には、属性タイプのごく一部のみが使用されます。表B.1「一般的に使用される属性のタイプ」 に遭遇する可能性が最も高い属性タイプの選択を示します。

表B.1 一般的に使用される属性のタイプ

文字列表現	X.500 属性タイプ
データのサイズ	同等の OID
C	countryName
2	2.5.4.6

文字列表現	X.500 属性タイプ
O	organizationName
1...64	2.5.4.10
OU	organizationalUnitName
1...64	2.5.4.11
CN	commonName
1...64	2.5.4.3
ST	stateOrProvinceName
1...64	2.5.4.8
L	localityName
1...64	2.5.4.7
PIDGIN	streetAddress
DC	domainComponent
UID	userid

AVA

属性値のアサーション (AVA) は属性値を属性型に割り当てます。文字列表現では、以下の構文があります。

> **<attr-type>=<attr-value>**

以下に例を示します。

> CN=A. N. Other

または、同等の OID を使用して、文字列表現の属性タイプを特定できます (表B.1「一般的に使用される属性のタイプ」 を参照してください)。以下に例を示します。

```
2.5.4.3=A. N. Other
```

RDN

相対識別名 (RDN) は、DN の単一ノード (文字列表現のコンマ間で表示されるビット) を表します。技術的には、RDN には複数の AVA が含まれる場合があります (これは AVA のセットとして正式に定義されます)。ただし、これはほとんど行われていません。文字列表現では、RDN の構文は以下のようになります。

```
<attr-type>=<attr-value>[+<attr-type>=<attr-value> ...]
```

以下は、(非常にまれな) 複数値 RDN の冷です。

```
OU=Eng1+OU=Eng2+OU=Eng3
```

以下は、単一値 RDN の例です。

```
OU=Engineering
```