Red Hat build of Keycloak 24.0

サーバー開発者ガイド

Last Updated: 2024-06-04

法律上の通知

概要

このガイドには、開発者が Red Hat build of Keycloak 24.0 をカスタマイズするために必要な情報が記載されています。

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、Red Hat CTO である Chris Wright のメッセージ をご覧ください。

第1章 はじめに

サンプルのリストでは、1行に表示されるものが、利用可能なページ幅に収まらない場合があります。これらの行は、分割されています。行の末尾にある\は、ページに収まるように、改行が追加されていることを意味しています。つまり、

```
Let's pretend to have an extremely \
  long line that \
  does not fit
This one is short
```

上記は、以下を指します。

```
Let's pretend to have an extremely long line that does not fit
This one is short
```

第2章 管理 REST API

Red Hat build of Keycloak には、管理コンソールの全機能が含まれる、完全に機能する管理 REST API が同梱されています。

API を呼び出すには、適切なパーミッションでアクセストークンを取得する必要があります。必要なパーミッションは、サーバー管理ガイド で説明しています。

Red Hat build of Keycloak を使用してアプリケーションの認証を有効にすることで、トークンを取得できます。アプリケーションおよびサービスのセキュリティー保護ガイドを参照してください。直接アクセス許可を使用して、アクセストークンを取得することもできます。

2.1. CURL の使用例

2.1.1. ユーザー名とパスワードでの認証

注記

次の例では、スタートガイド のチュートリアルのとおり、パスワード **password** を使用してユーザー **admin** を **master** レルムに作成したことを前提としています。

手順

1. ユーザー名 **admin** およびパスワード **password** を使用して、**master** レルムのユーザーのアクセストークンを取得します。

```
curl \
  -d "client_id=admin-cli" \
  -d "username=admin" \
  -d "password=password" \
  -d "grant_type=password" \
  "http://localhost:8080/realms/master/protocol/openid-connect/token"
```

注記

デフォルトでは、このトークンは 1 分で有効期限が切れます。

結果は JSON 形式のドキュメントになります。

2. **access_token** プロパティーの値を抽出して、必要な API を呼び出します。
3. API へのリクエストの **Authorization** ヘッダーに値を追加して、API を呼び出します。以下の例は、master レルムの詳細を取得する方法を示しています。

```
curl \
  -H "Authorization: bearer eyJhbGciOiJSUz..." \
  "http://localhost:8080/admin/realms/master"
```

2.1.2. サービスアカウントでの認証

client_id および **client_secret** を使用し、Admin REST API に対して認証するには、この手順を実行します。

手順

1. クライアントが次のように設定されていることを確認してください。
   - **client_id** は、レルム **マスター** に所属する **機密** クライアントです。
   - **client_id** が **Service Accounts Enabled** オプションを有効にしています。
   - **client_id** には、カスタムの "Audience" マッパーがあります。
     - 含まれるクライアントオーディエンス: **security-admin-console**
2. Service Account Roles タブで **client_id** に admin というロールが割り当てられていることを確認します。

```
curl \
  -d "client_id=<YOUR_CLIENT_ID>" \
  -d "client_secret=<YOUR_CLIENT_SECRET>" \
  -d "grant_type=client_credentials" \
  "http://localhost:8080/realms/master/protocol/openid-connect/token"
```

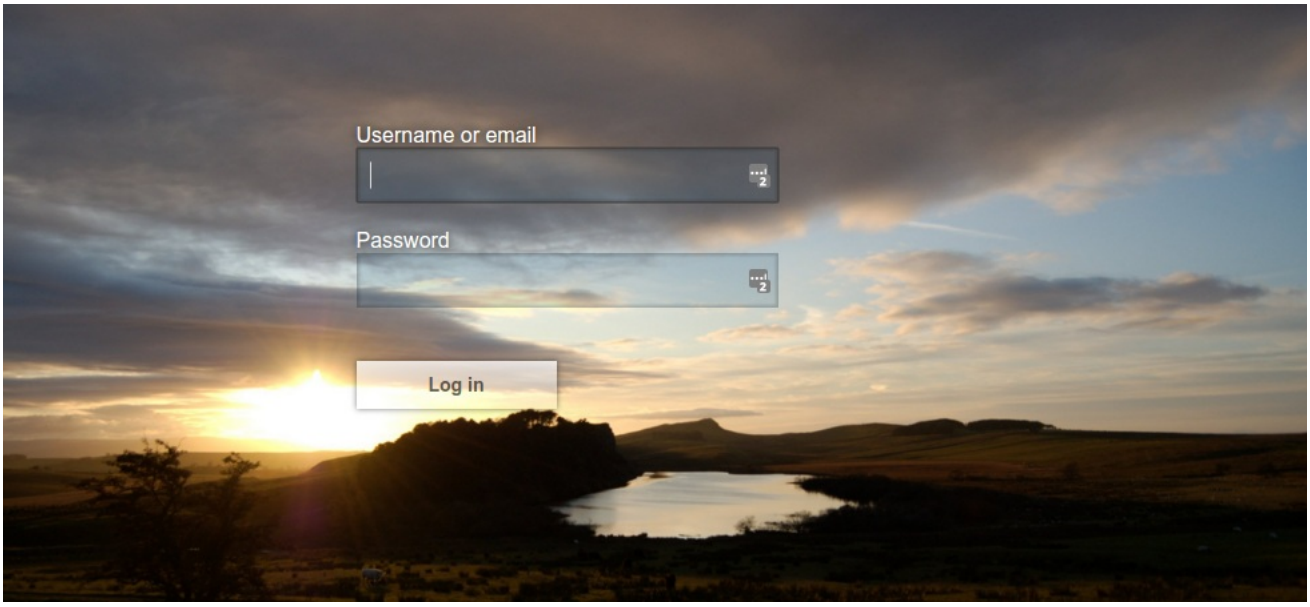
2.2. 関連情報

- サーバー管理ガイド
- アプリケーションおよびサービスの保護ガイド
- API Documentation

第3章 テーマ

Red Hat build of Keycloak は、Web ページとメールでテーマを使用できます。これにより、エンドユーザーに表示されるページのルックアンドフィールをカスタマイズでき、アプリケーションと統合できます。

図3.1 sunrise のサンプルテーマを含むログインページ



3.1. テーマのタイプ

テーマには 1 つ以上のタイプがあり、Red Hat build of Keycloak のさまざまな側面のカスタマイズに使用できます。利用可能なタイプは以下のとおりです。

- Account - アカウントコンソール
- Admin - 管理コンソール
- Email - メール
- Login - ログインフォーム
- Welcome - Welcome ページ

3.2. テーマの設定

Welcome を除くすべてのテーマタイプは、管理コンソールを介して設定されます。

手順

1. 管理コンソールにログインします。
2. 左上隅のドロップダウンボックスからレルムを選択します。
3. メニューで **Realm Settings** をクリックします。
4. **Themes** タブをクリックします。

注記

master 管理コンソールのテーマを設定するには、**master** レルムの管理コンソールテーマを設定する必要があります。

5. 管理コンソールへの変更を表示するには、ページをリフレッシュします。
6. **spi-theme-welcome-theme** オプションを使用して welcome テーマを変更します。
7. 以下に例を示します。

```
bin/kc.[sh|bat] start --spi-theme-welcome-theme=custom-theme
```

3.3. デフォルトのテーマ

Red Hat build of Keycloak は、サーバーディストリビューション内の JAR ファイル **keycloak-themes-24.0.0.redhat-00001.jar** のデフォルトテーマにバンドルされています。サーバーのルートの **themes** ディレクトリーには、デフォルトでテーマが含まれていませんが、デフォルトテーマに関する追加の詳細が記載された README ファイルが含まれています。アップグレードを簡略化するために、バンドルされているテーマを直接編集しないでください。代わりに、独自のテーマを作成して、バンドルされたテーマのいずれかを拡張します。

3.4. テーマの作成

テーマは次のもので構成されます。

- HTML テンプレート (Freemarker テンプレート)
- イメージ
- メッセージバンドル
- スタイルシート
- スクリプト
- テーマのプロパティー

すべてのページを置き換える予定がない限り、別のテーマを拡張することを推奨します。ほとんどの場合、既存のテーマを拡張することになるでしょう。または、管理コンソールまたはアカウントコンソールの独自の実装を提供する場合は、**base** テーマを拡張することを検討してください。**base** テーマはメッセージバンドルで構成されているため、メインの **index.ftl** Freemarker テンプレートの実装を含め、実装を最初から開始する必要があります。ただし、メッセージバンドルから既存の翻訳を活用することができます。

テーマを拡張する場合は、個別のリソース (テンプレート、スタイルシートなど) をオーバーライドできます。HTML テンプレートの上書きをすることにした場合は、新しいリリースにアップグレードする際にカスタムテンプレートを更新する必要があることに注意してください。

テーマを作成する際に、Red Hat build of Keycloak を再起動せずに **themes** ディレクトリーから直接テーマリソースを編集できるため、キャッシングを無効にするとを推奨します。

手順

1. 以下のオプションを指定して Keycloak を実行します。

```
bin/kc.[sh|bat] start --spi-theme-static-max-age=-1 --spi-theme-cache-themes=false --spi-theme-cache-templates=false
```

2. **themes** ディレクトリーにディレクトリーを作成します。
ディレクトリーの名前はテーマの名前になります。たとえば、**mytheme** というテーマを作成するには、**themes/mytheme** ディレクトリーを作成します。

3. theme ディレクトリーには、テーマが提供するタイプごとにディレクトリーを作成します。
たとえば、ログインタイプを **mytheme** テーマに追加するには、**themes/mytheme/login** ディレクトリーを作成します。

4. それぞれのタイプに対して、テーマの設定を可能にする **theme.properties** ファイルを作成します。
たとえば、テーマ **themes/mytheme/login** を設定して **base** テーマを拡張し、いくつかの共通リソースをインポートするには、次の内容を含むファイル **themes/mytheme/login/theme.properties** を作成します。

```
parent=base
import=common/keycloak
```

これで、ログインタイプのサポートのあるテーマが作成されました。

5. 管理コンソールにログインし、新しいテーマをチェックアウトします。

6. レルムを選択します。

7. メニューで **Realm Settings** をクリックします。

8. **Themes** タブをクリックします。

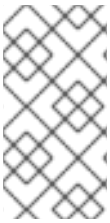
9. **Login Theme** については **mytheme** を選択し、**Save** をクリックします。

10. レルムのログインページを開きます。
これは、アプリケーションからログインするか、アカウントコンソール (**/realms/{realm name}/account**) を開くことによって実行できます。

11. 親テーマの変更の効果を確認するには、**parent=keycloak** in **theme.properties** に設定し、ログインページを更新します。

注記

パフォーマンスに大きく影響するため、必ず実稼働環境でキャッシュを再度有効にしてください。



注記

テーマキャッシュのコンテンツを手動で削除する場合は、サーバーディストリビューションの **data/tmp/kc-gzip-cache** ディレクトリーを削除することで実行できます。これは、たとえば、以前のサーバー実行の際にテーマのキャッシュを無効にせずにカスタムプロバイダーまたはカスタムテーマを再デプロイした場合に便利です。

3.4.1. テーマのプロパティー

テーマプロパティーは、テーマディレクトリーの **<THEME TYPE>/theme.properties** ファイルで設定されます。

- parent - 拡張する親テーマ
- import - 別のテーマからのリソースのインポート
- common - 共通リソースパスをオーバーライドします。指定しない場合のデフォルト値は **common/keycloak** です。この値は、**${url.resourcesCommonPath}** の接尾辞の値として、通常、freemarker テンプレートで使用されます (**${url.resoucesCommonPath}** 値の接頭辞はテーマのルート URI です)。
- styles - 追加するスタイルのスペース区切りリスト
- locales - サポートされるロケールのコンマ区切りリスト

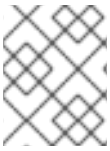
特定の要素タイプに使用される css クラスを変更するために使用できるプロパティーのリストがあります。これらのプロパティーのリストは、keycloak テーマ (**themes/keycloak/<THEME TYPE>/theme.properties**) の対応するタイプの theme.properties ファイルを参照してください。

独自のカスタムプロパティーを追加して、カスタムテンプレートから使用することもできます。

これを行うには、次の形式を使用してシステムプロパティーまたは環境変数を置き換えることができます。

- **${some.system.property}** - システムプロパティーの場合。
- **${env.ENV_VAR}** - 環境変数の場合。

システムプロパティーまたは環境変数が **${foo:defaultValue}** で見つからない場合は、デフォルト値を指定することもできます。



注記

デフォルト値が指定されておらず、対応するシステムプロパティーまたは環境変数がない場合は、何も置き換えられず、テンプレートの形式が使用されることになります。

以下は、実現できる例の一部です。

```
javaVersion=${java.version}

unixHome=${env.HOME:Unix home not found}
windowsHome=${env.HOMEPATH:Windows home not found}
```

3.4.2. テーマにスタイルシートを追加

テーマに 1 つ以上のスタイルシートを追加できます。

手順

1. テーマの **<THEME TYPE>/resources/css** ディレクトリーにファイルを作成します。
2. それを **theme.properties** の **styles** プロパティーに追加します。
   たとえば、**styles.css** を **mytheme** に追加するには、以下の内容を含む **themes/mytheme/login/resources/css/styles.css** を作成します。

```
.login-pf body {
    background: DimGrey none;
}
```

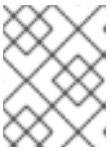
3. **themes/mytheme/login/theme.properties** を編集し、以下を追加します。

```
styles=css/styles.css
```

4. 変更を確認するには、レルムのログインページを開きます。
適用されるスタイルは、カスタムのスタイルシートからのものだけであることに注目してください。

5. 親テーマのスタイルを含めるには、そのテーマのスタイルを読み込みます。**themes/mytheme/login/theme.properties** を編集し、**styles** を次のように変更します。

```
styles=css/login.css css/styles.css
```



注記

親スタイルシートのスタイルをオーバーライドするには、スタイルシートが最後にリストされていることを確認してください。

3.4.3. テーマにスクリプトを追加

1 つ以上のスクリプトをテーマに追加できます。

手順

1. テーマの **<THEMETYPE>/resources/js** ディレクトリーにファイルを作成します。

2. ファイルを **theme.properties** の **scripts** プロパティーに追加します。
たとえば、**script.js** を **mytheme** に追加するには、以下の内容で **themes/mytheme/login/resources/js/script.js** を作成します。

```
alert('Hello');
```

次に、**themes/mytheme/login/theme.properties** を編集し、以下を追加します。

```
scripts=js/script.js
```

3.4.4. テーマにイメージを追加

テーマがイメージを使用できるようにするには、テーマの **<THEME TYPE>/resources/img** ディレクトリーにイメージを追加します。これらはスタイルシート内から使用することも、HTML テンプレートで直接使用することもできます。

たとえば、イメージを **mytheme** に追加するには、イメージを **themes/mytheme/login/resources/img/image.jpg** にイメージをコピーします。

次に、以下を使用して、カスタムスタイルシート内からこのイメージを使用できます。

```
body {
```

```
    background-image: url('../img/image.jpg');
    background-size: cover;
}
```

または、HTML テンプレートで直接使用するには、以下をカスタム HTML テンプレートに追加します。

```
<img src="${url.resourcesPath}/img/image.jpg" alt="My image description">
```

3.4.5. メールテーマへの画像の追加

テーマで画像を使用できるようにするには、テーマの **<THEME TYPE>/email/resources/img** ディレクトリーに画像を追加します。この画像は HTML テンプレート内で直接使用できます。

たとえば、**mytheme** に画像を追加するには、画像を **themes/mytheme/email/resources/img/logo.jpg** にコピーします。

HTML テンプレートで直接使用するには、カスタム HTML テンプレートに以下を追加します。

```
<img src="${url.resourcesUrl}/img/image.jpg" alt="My image description">
```

3.4.6. メッセージ

テンプレートのテキストは、メッセージバンドルからロードされます。別のテーマを拡張するテーマは、親のメッセージバンドルからすべてのメッセージを継承し、**<THEME TYPE>/messages/messages_en.properties** をテーマに追加して個別のメッセージを上書きできます。

たとえば、ログインフォーム上の **Username** を、**mytheme** の **Your Username** に置き換えるには、以下の内容で **themes/mytheme/login/messages/messages_en.properties** ファイルを作成します。

```
usernameOrEmail=Your Username
```

メッセージの使用時に **{0}** や **{1}** などのメッセージ値内では引数に置き換えられます。たとえば、**Log in to {0}** の {0} はレルム名に置き換えられます。

これらのメッセージバンドルのテキストは、レルム固有の値で上書きすることができます。レルム固有の値は、UI や API で管理できます。

3.4.7. レルムに言語を追加

前提条件

- レルムの国際化を有効にするには、サーバー管理ガイド を参照してください。

手順

1. テーマのディレクトリーに **<THEME TYPE>/messages/messages_<LOCALE>.properties** ファイルを作成します。
2. このファイルを **<THEME TYPE>/theme.properties** の **locales** プロパティーに追加します。言語をユーザーが利用できるようにするには、レルムの **login**、**account** および **email** テーマがその言語をサポートしていなければならないため、これらのテーマタイプに言語を追加する必

要があります。
たとえば、**mytheme** テーマにノルウェー語の翻訳を追加するには、以下の内容で **themes/mytheme/login/messages/messages_no.properties** ファイルを作成します。

```
usernameOrEmail=Brukernavn
password=Passord
```

メッセージの翻訳を省略すると、英語が使用されます。

3. **themes/mytheme/login/theme.properties** を編集し、以下を追加します。

```
locales=en,no
```

4. **account** と **email** のテーマタイプにも同じものを追加します。そのためには、**themes/mytheme/account/messages/messages_no.properties** および **themes/mytheme/email/messages/messages_no.properties** を作成します。これらのファイルを空のままにすると、英語のメッセージが使用されます。

5. **themes/mytheme/login/theme.properties** を **themes/mytheme/account/theme.properties** および **themes/mytheme/email/theme.properties** にコピーします。

6. 言語セレクターの翻訳を追加します。これには、英語の翻訳にメッセージを追加します。これには、以下を **themes/mytheme/account/messages/messages_en.properties** および **themes/mytheme/login/messages/messages_en.properties** に追加します。

```
locale_no=Norsk
```

デフォルトでは、メッセージプロパティーファイルは UTF-8 を使用してエンコードする必要があります。Keycloak は、内容を UTF-8 として読み取れない場合、ISO-8859-1 処理にフォールバックします。Java の PropertyResourceBundle のドキュメントで説明されているように、Unicode 文字をエスケープできます。以前のバージョンの Keycloak では、**# encoding: UTF-8** のようなコメントを使用して最初の行にエンコーディングを指定することがサポートされていました。これはサポートされなくなりました。

関連情報

- 現在のロケールの選択方法の詳細については、ロケールセレクター を参照してください。

3.4.8. カスタムアイデンティティープロバイダーのアイコンの追加

Red Hat build of Keycloak は、ログイン画面に表示されるカスタムアイデンティティープロバイダーのアイコンの追加に対応しています。

手順

1. ログイン **theme.properties** ファイル (**themes/mytheme/login/theme.properties** など) の中で、キーパターン **kcLogoIdP-<alias>** でアイコンクラスを定義します。

2. **myProvider** というエイリアスを持つ Identity プロバイダーの場合は、カスタムテーマの **theme.properties** ファイルに行を追加できます。以下に例を示します。

```
kcLogoIdP-myProvider = fa fa-lock
```

すべてのアイコンは、PatternFly4 の公式 Web サイトで公開されています。ソーシャルプロバイダー用のアイコンは、**base** ログインテーマのプロパティー (**themes/keycloak/login/theme.properties**) ですでに定義されているため、それを参考にすることができます。

3.4.9. カスタム HTML テンプレートの作成

Red Hat build of Keycloak は、Apache Freemarker テンプレートを使用して HTML を生成し、ページをレンダリングします。

カスタムテンプレートを作成してページのレンダリング方法を完全に変更することも可能ですが、組み込みテンプレートをできるだけ活用することを推奨します。理由は次のとおりです。

- アップグレード時に、新しいバージョンから最新の更新を取得するために、カスタムテンプレートを更新する必要がある場合があります。
- テーマに CSS スタイルを設定 すると、UI をご自身の UI 設計標準およびガイドラインに合わせて調整できます。
- ユーザープロファイルを使用 すると、カスタムユーザー属性をサポートし、そのレンダリング方法を設定できます。

ほとんどの場合、Red Hat build of Keycloak をニーズに合わせて調整するためにテンプレートを変更する必要はありません。ただし、**<THEME TYPE>/<TEMPLATE>.ftl** を作成することで、独自のテーマ内の個々のテンプレートをオーバーライドできます。管理者およびアカウントコンソールは、アプリケーションのレンダリングに単一のテンプレート **index.ftl** を使用します。

他のテーマタイプのテンプレートのリストについては、**$KEYCLOAK_HOME/lib/lib/main/org.keycloak.keycloak-themes-<VERSION>.jar** にある JAR ファイル内の **theme/base/<THEME_TYPE>** ディレクトリーを参照してください。

手順

1. テンプレートをベーステーマから独自のテーマにコピーします。
2. 必要な変更を適用します。
   たとえば、**mytheme** テーマのカスタムログインフォームを作成するには、**themes/base/login/login.ftl** を **themes/mytheme/login** にコピーし、エディターで開きます。

   最初の行 (<#import ...>) の後に、**<h1>HELLO WORLD!</h1>** を以下のように追加します。

   ```
   <#import "template.ftl" as layout>
   <h1>HELLO WORLD!</h1>
   ...
   ```

3. 変更したテンプレートをバックアップします。Red Hat build of Keycloak の新しいバージョンにアップグレードする場合は、必要に応じて元のテンプレートに変更を適用できるようにカスタムテンプレートを更新する必要がある可能性があります。

関連情報

- テンプレートの編集方法の詳細については、FreeMarker Manual を参照してください。

3.4.10. 電子メール

パスワードリカバリーメールなどのメールの件名および内容を編集するには、テーマの **email** タイプにメッセージバンドルをを追加します。メールごとに 3 つのメッセージがあります。このサブジェクトの 1 つ。1 つはプレーンテキストの本文用に、もう 1 つは html 本文用です。

利用可能なすべてのメールを確認するには、**themes/base/email/messages/messages_en.properties** を確認します。

たとえば、**mytheme** テーマのパスワードリカバリーメールを変更するには、以下の内容で **themes/mytheme/email/messages/messages_en.properties** を作成します。

```
passwordResetSubject=My password recovery
passwordResetBody=Reset password link: {0}
passwordResetBodyHtml=<a href="{0}">Reset password</a>
```

3.5. テーマのデプロイ

テーマは、テーマディレクトリーを **themes** にコピーして Red Hat build of Keycloak にデプロイするか、アーカイブとしてデプロイできます。開発中にテーマを **themes** のディレクトリーにコピーできますが、実稼働環境では **archive** の使用を検討してください。**archive** を使用すると、特にクラスタリングなどにより Red Hat build of Keycloak のインスタンスが複数ある場合に、バージョン管理されたテーマのコピーを簡単に作成できます。

手順

1. テーマをアーカイブとしてデプロイするには、テーマリソースを使用して JAR アーカイブを作成します。
2. **META-INF/keycloak-themes.json** ファイルをアーカイブで利用可能なテーマをリスト表示するアーカイブに追加して、各テーマが提供するタイプを追加します。
   たとえば、**mytheme** テーマでは、以下の内容を含む **mytheme.jar** を作成します。
   - META-INF/keycloak-themes.json
   - theme/mytheme/login/theme.properties
   - theme/mytheme/login/login.ftl
   - theme/mytheme/login/resources/css/styles.css
   - theme/mytheme/login/resources/img/image.png
   - theme/mytheme/login/messages/messages_en.properties
   - theme/mytheme/email/messages/messages_en.properties
     この場合、**META-INF/keycloak-themes.json** の内容は以下のようになります。

     ```
     {
        "themes": [{
           "name" : "mytheme",
           "types": [ "login", "email" ]
        }]
     }
     ```

     1 つのアーカイブに複数のテーマを含めることができ、各テーマでは 1 つまたは複数のタイプをサポートできます。

Red Hat build of Keycloak にアーカイブをデプロイするには、それを Red Hat build of Keycloak の **providers/** ディレクトリーに追加し、サーバーがすでに実行されている場合は再起動します。

3.6. テーマに関する関連情報

- 参考例については、Red Hat build of Keycloak にバンドルされている デフォルトのテーマ を参照してください。
- Red Hat build of Keycloak Quickstarts Repository - クイックスタートリポジトリーのディレクトリー **extension** に、テーマのサンプルがいくつか含まれています。このサンプルは参考例としても使用できます。=== テーマセレクター

デフォルトでは、レルムに設定されたテーマが使用されます。ただし、クライアントがログインテーマをオーバーライドできる場合を除きます。この動作は Theme Selector SPI で変更できます。

これは、たとえばユーザーエージェントヘッダーを確認して、デスクトップおよびモバイルデバイスの異なるテーマを選択するために使用できます。

カスタムテーマセレクターを作成するには、**ThemeSelectorProviderFactory** および **ThemeSelectorProvider** を実装する必要があります。

3.7. テーマリソース

Red Hat build of Keycloak でカスタムプロバイダーを実装するのであれば、多くの場合はテンプレート、リソース、およびメッセージバンドルをさらに追加する必要があります。

追加のテーマリソースを読み込む最も簡単な方法は、**theme-resources/resources** の **theme-resources/templates** リソース内に JAR を作成し、 **theme-resources/messages** のメッセージバンドルを作成することです。

テンプレートとリソースをさらに柔軟に読み込む方法が必要な場合には、ThemeResourceSPI で実現できます。**ThemeResourceProviderFactory** および **ThemeResourceProvider** を実装することで、テンプレートおよびリソースを読み込む方法を正確に決定できます。

3.8. ロケールセレクター

デフォルトでは、**LocaleSelectorProvider** インターフェイスを実装する **DefaultLocaleSelectorProvider** を使用してロケールが選択されます。国際化が無効になっている場合は、英語がデフォルト言語です。

国際化を有効にした場合、ロケールは サーバー管理ガイド に記載されているロジックに従って解決されます。

この動作は、**LocaleSelectorProvider** および **LocaleSelectorProviderFactory** を実装して、**LocaleSelectorSPI** を介して変更できます。

LocaleSelectorProvider インターフェイスには単一のメソッド **resolveLocale** があり、**RealmModel** および Null が可能な **UserModel** を指定したロケールを返す必要があります。実際のリクエストは **KeycloakSession#getContext** メソッドから利用できます。

カスタム実装は、デフォルトの動作の一部を再利用できるように **DefaultLocaleSelectorProvider** を拡張できます。たとえば、**Accept-Language** リクエストヘッダーを無視する場合、カスタム実装はデフォルトのプロバイダーを拡張し、その **getAcceptLanguageHeaderLocale** をオーバーライドして Null 値を返すことができます。その結果、ロケールの選択はレルムのデフォルト言語にフォールバックします。

3.9. ロケールセレクターに関する関連情報

- カスタムプロバイダーの作成とデプロイの詳細については、サービスプロバイダーインターフェイス を参照してください。

第4章 アイデンティティーブローカー API

Red Hat build of Keycloak は、親 IDP にログイン認証を委譲できます。典型的な例としては、Facebook や Google などのソーシャルプロバイダーを使用してユーザーがログインできるようにするケースがあります。既存のアカウントをブローカー化された IDP にリンクすることもできます。このセクションでは、アイデンティティーブローカーに関連するので、アプリケーションが使用する API についても一部説明します。

4.1. 外部 IDP トークンの取得

Red Hat build of Keycloak を使用すると、外部 IDP を使用して認証プロセスからのトークンと応答を保存できます。これには、IDP の設定ページで **Store Token** 設定オプションを使用できます。

アプリケーションコードは、これらのトークンを取得し、追加のユーザー情報でプルしたり、外部 IDP でリクエストを安全に呼び出すことを行うことができます。たとえば、アプリケーションは Google トークンを使用して他の Google サービスおよび REST API で呼び出す場合があります。特定のアイデンティティープロバイダーのトークンを取得するには、以下のようにリクエストを送信する必要があります。

```
GET /realms/{realm}/broker/{provider_alias}/token HTTP/1.1
Host: localhost:8080
Authorization: Bearer <KEYCLOAK ACCESS TOKEN>
```

アプリケーションは、Red Hat build of Keycloak で認証され、アクセストークンを受け取っている必要があります。このアクセストークンには、**ブローカー** のクライアントレベルのロール **read-token** を設定する必要があります。そのため、ユーザーにはこのロールのロールマッピングが必要で、クライアントアプリケーションにそのスコープ内にそのロールがなければなりません。この場合、Red Hat build of Keycloak で保護されたサービスにアクセスする場合には、ユーザー認証時に Red Hat build of Keycloak が発行するアクセストークンを送信する必要があります。ブローカー設定ページでは、**Stored Tokens Readable** スイッチを有効にして、新たにインポートしたユーザーにこのロールを自動的に割り当てることができます。

これらの外部トークンについては、プロバイダー経由で再度ログインするか、クライアントが開始したアカウントリンク API を使用して再度確立できます。

4.2. クライアント開始アカウントリンク

アプリケーションによっては、Facebook などのソーシャルプロバイダーと統合しても、このようなソーシャルプロバイダー経由でログインするオプションを提供しないようにする場合もあります。Red Hat build of Keycloak は、アプリケーションが既存のユーザーアカウントを特定の外部 IDP にリンクするために使用できるブラウザーベースの API を提供します。これは、クライアント開始アカウントリンク と呼ばれます。アカウントリンクは OIDC アプリケーションによってのみ開始できます。

アプリケーションがユーザーのブラウザーを Red Hat build of Keycloak サーバーの URL に転送する方法は、ユーザーのアカウントを特定の外部プロバイダー (Facebook など) にリンクするように要求することです。サーバーは外部プロバイダーでログインを開始します。ブラウザーは外部プロバイダーにログインし、サーバーにリダイレクトされます。サーバーはリンクを確立し、確認でアプリケーションにリダイレクトします。

このプロトコルを開始する前に、クライアントアプリケーションが満たさなければならない前提条件があります。

- 必要なアイデンティティープロバイダーは、管理コンソールでユーザーのレルムに対して設定して有効にする必要があります。

- ユーザーアカウントは、OIDC プロトコルを使用して既存のユーザーとしてログインしている必要があります。
- ユーザーに **account.manage-account** または **account.manage-account-links** ロールマッピングが必要です。
- アプリケーションには、アクセストークン内で上記のロールのスコープが付与される必要があります。
- アプリケーションは、リダイレクト URL 生成にその情報を必要とするため、アクセストークンへのアクセスが必要です。

ログインを開始するには、アプリケーションは URL を作成し、ユーザーのブラウザーをこの URL にリダイレクトする必要があります。URL は以下のようになります。

```
/{auth-server-root}/realms/{realm}/broker/{provider}/link?client_id={id}&redirect_uri={uri}&nonce={nonce}&hash={hash}
```

以下は、各パスおよびクエリーパラメーターの説明です。

provider

これは、管理コンソールの **Identity Provider** セクションで定義した外部 IDP のプロバイダーエイリアスです。

client_id

これは、アプリケーションの OIDC クライアント ID です。アプリケーションを管理コンソールでクライアントとして登録する場合は、このクライアント ID を指定する必要があります。

redirect_uri

これは、アカウントリンクの確立後にリダイレクトするアプリケーションのコールバック URL です。有効なクライアントのリダイレクト URI パターンである必要があります。つまり、管理コンソールでクライアント登録時に定義した有効な URL パターンのいずれかと一致する必要があります。

nonce

これは、アプリケーションが生成する必要のあるランダムな文字列です。

hash

これは、Base64 URL でエンコードされたハッシュです。このハッシュは、Base64 URL で **nonce** + **token.getSessionState()** + **token.getIssuedFor()** + **provider** の SHA_256 ハッシュをエンコードすることで生成されます。トークン変数は OIDC アクセストークンから取得されます。基本的に、アクセスするアイデンティティープロバイダーエイリアス、nonce、ユーザーセッション ID、クライアント ID、およびアイデンティティープロバイダーエイリアスを無作為にハッシュ化します。

以下は、アカウントリンクを確立するために URL を生成する Java Servlet コードの例になります。

```
  KeycloakSecurityContext session = (KeycloakSecurityContext)
httpServletRequest.getAttribute(KeycloakSecurityContext.class.getName());
  AccessToken token = session.getToken();
  String clientId = token.getIssuedFor();
  String nonce = UUID.randomUUID().toString();
  MessageDigest md = null;
  try {
    md = MessageDigest.getInstance("SHA-256");
  } catch (NoSuchAlgorithmException e) {
    throw new RuntimeException(e);
  }
```

```
String input = nonce + token.getSessionState() + clientId + provider;
byte[] check = md.digest(input.getBytes(StandardCharsets.UTF_8));
String hash = Base64Url.encode(check);
request.getSession().setAttribute("hash", hash);
String redirectUri = ...;
String accountLinkUrl = KeycloakUriBuilder.fromUri(authServerRootUrl)
                .path("/realms/{realm}/broker/{provider}/link")
                .queryParam("nonce", nonce)
                .queryParam("hash", hash)
                .queryParam("client_id", clientId)
                .queryParam("redirect_uri", redirectUri).build(realm, provider).toString();
```

このハッシュを含める理由認証サーバーで、クライアントアプリケーションが要求を開始したことを認識し、その他の不正なアプリが、特定のプロバイダーにリンクされるユーザーアカウントを無作為に要求しないように、ハッシュを含めます。認証サーバーは、最初にログイン時に設定された SSO クッキーをチェックしてユーザーがログインしているかどうかを確認します。その後、現在のログインに基づいてハッシュを再生成し、アプリケーションによって送信されるハッシュにマッチします。

アカウントのリンク後、認証サーバーは **redirect_uri** にリダイレクトします。リンク要求の処理に問題がある場合は、認証サーバーが **redirect_uri** にリダイレクトされない場合があります。ブラウザーは、アプリケーションにリダイレクトされるのではなく、エラーページで終了できます。エラー条件があり、認証サーバーがクライアントアプリケーションにリダイレクトするのに十分な安全でない場合には、追加の **error** クエリーパラメーターが **redirect_uri** に追加されます。

警告

この API は、アプリケーションで確実に要求を開始されるようにしますが、この操作に対する CSRF 攻撃を完全に防ぐことはできません。このアプリケーションは、CSRF 攻撃ターゲットに対して自己防衛します。

4.2.1. 外部トークンのリフレッシュ

プロバイダーにログインして生成される外部トークン (つまり Facebook または GitHub トークン) を使用している場合は、アカウントリンク API を再度初期化してこのトークンを更新できます。

第5章 SPI (サービスプロバイダーインターフェイス)

Red Hat build of Keycloak は、カスタムコードを必要とせずにほとんどのユースケースに対応するように設計されていますが、カスタマイズ性も望まれています。そのため、Red Hat build of Keycloak には、独自プロバイダーの実装を可能にする数多くの SPI (サービスプロバイダーインターフェイス) があります。

5.1. SPI の実装

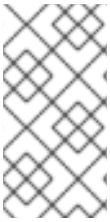
SPI を実装するには、その ProviderFactory および Provider インターフェイスを実装する必要があります。サービス設定ファイルも作成する必要があります。

たとえば、Theme Selector SPI を実装するには、ThemeSelectorProviderFactory および ThemeSelectorProvider を実装し、さらに **META-INF/services/org.keycloak.theme.ThemeSelectorProviderFactory** ファイルも指定する必要があります。

ThemeSelectorProviderFactory の例:

```
package org.acme.provider;

import ...

public class MyThemeSelectorProviderFactory implements ThemeSelectorProviderFactory {

    @Override
    public ThemeSelectorProvider create(KeycloakSession session) {
        return new MyThemeSelectorProvider(session);
    }

    @Override
    public void init(Config.Scope config) {
    }

    @Override
    public void postInit(KeycloakSessionFactory factory) {
    }

    @Override
    public void close() {
    }

    @Override
    public String getId() {
        return "myThemeSelector";
    }
}
```

プロバイダーファクトリーの実装では、**getId()** メソッドによって一意の ID を返すことを推奨します。ただし、以下の プロバイダーのオーバーライド セクションで説明するように、このルールには例外が存在する場合があります。



注記

Red Hat build of Keycloak はプロバイダーファクトリーの単一のインスタンスを作成するので、複数の要求の状態を保存できます。プロバイダーのインスタンスは、各要求のファクトリーで create を呼び出すことで作成されるので、軽量オブジェクトでなければなりません。

ThemeSelectorProvider の例:

```
package org.acme.provider;

import ...

public class MyThemeSelectorProvider implements ThemeSelectorProvider {

    public MyThemeSelectorProvider(KeycloakSession session) {
    }


    @Override
    public String getThemeName(Theme.Type type) {
        return "my-theme";
    }

    @Override
    public void close() {
    }
}
```

サービス設定ファイルの例 (**META-INF/services/org.keycloak.theme.ThemeSelectorProviderFactory**):

```
org.acme.provider.MyThemeSelectorProviderFactory
```

プロバイダーを設定するには、プロバイダーの設定 の章を参照してください。

たとえば、プロバイダーを設定するには、次のようにオプションを設定できます。

```
bin/kc.[sh|bat] --spi-theme-selector-my-theme-selector-enabled=true --spi-theme-selector-my-theme-selector-theme=my-theme
```

その後、init メソッド **ProviderFactory** で設定を取得できます。

```
public void init(Config.Scope config) {
    String themeName = config.get("theme");
}
```

必要に応じて、プロバイダーは他のプロバイダーを検索することもできます。以下に例を示します。

```
public class MyThemeSelectorProvider implements ThemeSelectorProvider {

    private KeycloakSession session;

    public MyThemeSelectorProvider(KeycloakSession session) {
```

```
        this.session = session;
    }

    @Override
    public String getThemeName(Theme.Type type) {
        return session.getContext().getRealm().getLoginTheme();
    }
}
```

5.1.1. ビルトインプロバイダーのオーバーライド

前述したように、**ProviderFactory** の実装では一意の ID を使用することを推奨します。ただし、同時に、Red Hat build of Keycloak のビルトインプロバイダーの 1 つをオーバーライドすると便利な場合があります。この場合に推奨される方法は、一意の ID を使用して ProviderFactory を実装し、たとえばプロバイダーの設定 の章で指定されているようにデフォルトのプロバイダーを設定することです。一方で、これが常に可能であるとは限りません。

たとえば、デフォルトの OpenID Connect プロトコルの動作にいくつかのカスタマイズが必要で、**OIDCLoginProtocolFactory** のデフォルトの Red Hat build of Keycloak 実装をオーバーライドする場合は、同じ providerId を保持する必要があります。たとえば、管理コンソール、OIDC プロトコルの既知のエンドポイント、およびその他のさまざまなものは、プロトコルファクトリーの ID が **openid-connect** であることを前提としています。

この場合、カスタム実装のメソッド **order()** を実装し、その順序をビルトインの実装よりも上位にすることを強く推奨します。

```
public class CustomOIDCLoginProtocolFactory extends OIDCLoginProtocolFactory {

    // Some customizations here

    @Override
    public int order() {
        return 1;
    }
}
```

同じプロバイダー ID を持つ実装が複数ある場合、最上位の実装のみが Red Hat build of Keycloak ランタイムによって使用されます。

5.1.2. 管理コンソールで SPI 実装からの情報を表示

プロバイダーに関する追加情報を Red Hat build of Keycloak 管理者に表示すると便利な場合があります。プロバイダーのビルド時間情報 (現在インストールされているカスタムプロバイダーのバージョンなど)、プロバイダーの現在の設定 (プロバイダーが通信するリモートシステムの URL)、または一部の運用情報 (プロバイダーが対話するリモートシステムからの応答時間) を表示できます。Red Hat build of Keycloak 管理コンソールには、そのような情報を表示するサーバー情報ページがあります。

プロバイダーからの情報を表示するには、**ProviderFactory** に **org.keycloak.provider.ServerInfoAwareProviderFactory** インターフェイスを実装するだけで十分です。

前の例の **MyThemeSelectorProviderFactory** の実装例:

```
package org.acme.provider;
```

```
import ...

public class MyThemeSelectorProviderFactory implements ThemeSelectorProviderFactory,
ServerInfoAwareProviderFactory {
    ...

    @Override
    public Map<String, String> getOperationalInfo() {
        Map<String, String> ret = new LinkedHashMap<>();
        ret.put("theme-name", "my-theme");
        return ret;
    }
}
```

5.2. 利用可能なプロバイダーの利用

プロバイダーの実装では、Red Hat build of Keycloak で利用可能な他のプロバイダーを使用できます。既存のプロバイダーは通常、**KeycloakSession** を使用して取得できます。これは、セクション SPI の実装 で説明されているように、プロバイダーで利用できます。

Red Hat build of Keycloak には 2 つのプロバイダータイプがあります。

- **単一実装のプロバイダータイプ** - Red Hat build of Keycloak ランタイムで、1 つのプロバイダータイプのみアクティブに実装できます。
  たとえば **HostnameProvider** は、Red Hat build of Keycloak で使用されるホスト名を指定し、そのホスト名が Red Hat build of Keycloak サーバー全体で共有されます。そのため、Red Hat build of Keycloak サーバーでアクティブになるこのプロバイダーの実装は 1 つだけです。サーバーランタイムに複数のプロバイダー実装が利用できる場合は、それらのいずれかをデフォルトとして指定する必要があります。

たとえば、以下のようなものです。

```
bin/kc.[sh|bat] build --spi-hostname-provider=default
```

default-provider の値として使用される **default** は、特定のプロバイダーファクトリー実装の **ProviderFactory.getId()** が返す ID と一致する必要があります。コードで は、**keycloakSession.getProvider(HostnameProvider.class)** のようにプロバイダーを取得します。

- **複数実装のプロバイダータイプ** - Red Hat build of Keycloak ランタイムで複数の実装を利用し、一緒に動作させることができるプロバイダータイプです。
  たとえば、**EventListener** プロバイダーは、利用可能で登録された複数の実装を持つことができます。これは、特定のイベントがすべてのリスナー (jboss-logging、sysout など) に送信できることを意味します。コードでは、たとえば **session.getProvider(EventListener.class, "jboss-logging")** のように、プロバイダーの指定したインスタンスを取得することができます。前述のように、このプロバイダータイプのインスタンスは複数存在する可能性があるため、第 2 引数にプロバイダーの **provider_id** を指定する必要があります。

  プロバイダー ID は、特定のプロバイダーファクトリー実装の **ProviderFactory.getId()** が返す ID と一致する必要があります。プロバイダーの種類によっては、第 2 引数に **ComponentModel** を指定することで取得できるものもありますし、**KeycloakSessionFactory** を指定しなければならないものもあります (**Authenticator** など)。将来的に非推奨となる可能性があるため、この方法で独自のプロバイダーを実装することは推奨できません。

5.3. プロバイダー実装の登録

プロバイダーは、JAR ファイルを **providers** ディレクトリーにコピーするだけでサーバーに登録されます。

Keycloak が提供していない別の依存関係が追加で必要な場合は、それを **providers** ディレクトリーにコピーします。

新しいプロバイダーまたは依存関係を登録した後、Keycloak を、最適化なしの start コマンドまたは **kc.[sh|bat] build** コマンドで再構築する必要があります。

注記

プロバイダーの JAR は、分離されたクラスローダーにロードされないため、組み込みのリソースまたはクラスと、競合するリソースまたはクラスをプロバイダーの JAR に含めないでください。特に、application.properties ファイルを含めるか、commons-lang3 依存関係をオーバーライドすると、プロバイダーの JAR が削除された場合に自動ビルドが失敗します。競合するクラスが含まれている場合、サーバーの起動ログにパッケージ分離の警告が表示されることがあります。すべての組み込みの lib jar がパッケージ分離の警告ロジックによってチェックされるわけではありません。そのため、推移的な依存関係をバンドルまたは含める前に、lib ディレクトリーの JAR をチェックする必要があります。競合が発生した場合は、問題のあるクラスを削除するか再パッケージ化することで解決できます。

競合するリソースファイルがあっても、警告は表示されません。JAR のリソースファイルのパス名が、そのプロバイダーに固有のものであることを確認してください。または、次のような方法で、**"install root"/lib/lib/main** ディレクトリー配下の JAR の内容に **some.file** が存在するかどうかを確認してください。

```
find . -type f -name "*.jar" -exec unzip -l {} \; | grep some.file
```

削除されたプロバイダーの JAR に関連する **NoSuchFileException** エラーが原因でサーバーが起動しない場合は、次を実行します。

```
./kc.sh -Dquarkus.launch.rebuild=true
```

これにより、Quarkus がクラスローディング関連のインデックスファイルを強制的に再構築します。その後、最適化なしの start または build を例外を出さずに実行できるはずです。

5.3.1. プロバイダーの無効化

プロバイダーを無効化するには、プロバイダーの enabled 属性を false に設定します。たとえば、Infinispan ユーザーキャッシュプロバイダーを無効にするには、以下を使用します。

```
bin/kc.[sh|bat] build --spi-user-cache-infinispan-enabled=false
```

5.4. JAVASCRIPT プロバイダー

Red Hat build of Keycloak には、管理者が特定の機能をカスタマイズできるように、ランタイム時にスクリプトを実行する機能があります。

- オーセンティケーター
- JavaScript ポリシー

- OpenID Connect Protocol Mapper
- SAML プロトコルマッパー

5.4.1. オーセンティケーター

認証スクリプトは、**Authenticator#action(AuthenticationFlowContext)** から呼び出される **Authenticator#authenticate(AuthenticationFlowContext) action(..)** から呼び出される **authenticate(..)** 関数を少なくとも 1 つ提供する必要があります。

カスタム **Authenticator** は、最低でも **authenticate(..)** 関数を提供する必要があります。コード内で **javax.script.Bindings** スクリプトを使用できます。

script
スクリプトメタデータにアクセスするための **ScriptModel**

realm
RealmModel

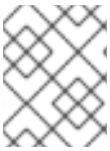
user
現在の **UserModel**。**user** を使用できるのは、別のオーセンティケーターがユーザーアイデンティティーの確立に成功し、ユーザーを認証セッションに設定した後にトリガーされるように、スクリプトオーセンティケーターが認証フロー内で設定されている場合です。

session
アクティブな **KeycloakSession**

authenticationSession
現在の **AuthenticationSessionModel**

httpRequest
the current **org.jboss.resteasy.spi.HttpRequest**

LOG
ScriptBasedAuthenticator にスコープ指定された **org.jboss.logging.Logger**



注記

authenticate(context) action(context) 関数に渡される **context** 引数から追加のコンテキスト情報を抽出できます。

```
AuthenticationFlowError = Java.type("org.keycloak.authentication.AuthenticationFlowError");

function authenticate(context) {

  LOG.info(script.name + " --> trace auth for: " + user.username);

  if (   user.username === "tester"
     && user.getAttribute("someAttribute")
     && user.getAttribute("someAttribute").contains("someValue")) {

     context.failure(AuthenticationFlowError.INVALID_USER);
     return;
  }
```

```
  context.success();
}
```

5.4.1.1. スクリプトオーセンティケーターを追加する場所

スクリプトオーセンティケーターの使用方法としては、認証の最後にいくつかのチェックを実行することが考えられます。スクリプトオーセンティケーターを (たとえば、アイデンティティー Cookie による SSO 再認証中であっても) 常にトリガーする場合、認証フローの最後にスクリプトオーセンティケーターを REQUIRED として追加し、既存のスクリプトオーセンティケーターを別の REQUIRED の認証サブフローにカプセル化する必要がある場合があります。これが必要なのは、REQUIRED と ALTERNATIVE の実行を同じレベルにすることができないためです。たとえば、認証フロー設定は次のようになります。

```
- User-authentication-subflow REQUIRED
-- Cookie ALTERNATIVE
-- Identity-provider-redirect ALTERNATIVE
...
- Your-Script-Authenticator REQUIRED
```

5.4.2. OpenID Connect Protocol Mapper

OpenID Connect Protocol Mapper スクリプトは、ID トークンやアクセストークンの内容を変更できる JavaScript スクリプトです。

コード内で **javax.script.Bindings** スクリプトを使用できます。

user
: 現在の **UserModel**

realm
: **RealmModel**

token
: 現在の **IDToken**。ID トークン用にマッパーが設定されている場合にのみ使用できます。

tokenResponse
: 現在の **AccessTokenResponse**。アクセストークン用にマッパーが設定されている場合にのみ使用できます。

userSession
: アクティブな **UserSessionModel**

keycloakSession
: アクティブな **KeycloakSession**

スクリプトのエクスポートが、トークンクレームの値として使用されます。

```
// prints can be used to log information for debug purpose.
print("STARTING CUSTOM MAPPER");

var inputRequest = keycloakSession.getContext().getHttpRequest();
var params = inputRequest.getDecodedFormParameters();
var output = params.getFirst("user_input");
exports = output;
```

上記のスクリプトを使用すると、認証リクエストから **user_input** を取得できます。これは、マッパーで設定された **Token Claim Name** にマッピングするために使用できます。

5.4.3. スクリプトで JAR を作成してデプロイ

注記

JAR ファイルは、**.jar** 拡張子を持つ通常の ZIP ファイルです。

Red Hat build of Keycloak でスクリプトを利用できるようにするには、サーバーにスクリプトをデプロイする必要があります。そのため、以下の構造で **JAR** ファイルを作成します。

```
META-INF/keycloak-scripts.json

my-script-authenticator.js
my-script-policy.js
my-script-mapper.js
```

META-INF/keycloak-scripts.json は、デプロイするスクリプトに関するメタデータ情報を提供するファイル記述子です。これは、次の構造が含まれる JSON ファイルです。

```
{
    "authenticators": [
        {
            "name": "My Authenticator",
            "fileName": "my-script-authenticator.js",
            "description": "My Authenticator from a JS file"
        }
    ],
    "policies": [
        {
            "name": "My Policy",
            "fileName": "my-script-policy.js",
            "description": "My Policy from a JS file"
        }
    ],
    "mappers": [
        {
            "name": "My Mapper",
            "fileName": "my-script-mapper.js",
            "description": "My Mapper from a JS file"
        }
    ],
    "saml-mappers": [
        {
            "name": "My Mapper",
            "fileName": "my-script-mapper.js",
            "description": "My Mapper from a JS file"
        }
    ]
}
```

このファイルは、デプロイする別のタイプのスクリプトプロバイダーを参照する必要があります。

- **authenticators**
  OpenID Connect Script Authenticator 用。同じ JAR ファイルに 1 つまたは複数のオーセンティケーターを設定できます。

- **policies**
  Red Hat build of Keycloak Authorization Services を使用する場合の JavaScript ポリシー用。同じ JAR ファイルに 1 つまたは複数のポリシーを設定できます。

- **mappers**
  OpenID Connect Script Protocol Mapper 用。同じ JAR ファイルに 1 つまたは複数のマッパーを設定できます。

- **saml-mappers**
  SAML Script Protocol Mapper 用。同じ JAR ファイルに 1 つまたは複数のマッパーを設定できます。

JAR ファイルのスクリプトファイルごとに、スクリプトファイルを特定のプロバイダータイプにマッピングする **META-INF/keycloak-scripts.json** に対応するエントリーが必要です。そのためには、各エントリーに以下のプロパティーを指定する必要があります。

- **name**
  Red Hat build of Keycloak 管理コンソールでのスクリプト表示に使用するための分かりやすい名前。指定がない場合は、代わりにスクリプトファイルを使用します。

- **description**
  スクリプトファイルの意図をより適切に説明する任意のテキスト

- **fileName**
  スクリプトファイルの名前。このプロパティーは **必須** であり、JAR 内のファイルにマップする必要があります。

5.4.4. スクリプト JAR のデプロイ

記述子とデプロイするスクリプトを含む JAR ファイルを作成したら、その JAR を Red Hat build of Keycloak の **providers/** ディレクトリーにコピーして、**bin/kc.[sh|bat] build** を実行します。**scripts** 機能を有効にする必要もあることに注意してください。

5.5. 利用可能な SPI

実行時に利用可能なすべての SPI のリストを表示する場合は、管理コンソール セクションで説明されているように、管理コンソールの **Server Info** ページを確認できます。

第6章 ユーザーストレージ SPI

ユーザーストレージ SPI を使用して拡張機能を Red Hat build of Keycloak に書き込み、外部ユーザーデータベースおよび認証情報ストアに接続できます。組み込み LDAP および ActiveDirectory サポートは、この SPI 動作を実装したものです。Red Hat build of Keycloak は、追加設定なしでローカルデータベースを使用して、ユーザーを作成、更新、検索し、クレデンシャルを検証します。ただし、多くの場合、組織には Red Hat build of Keycloak のデータモデルに移行できない既存の外部プロプライエタリーユーザーデータベースがあります。このような状況では、アプリケーション開発者は User Storage SPI の実装を記述して、Red Hat build of Keycloak がユーザーのログインと管理に使用する外部ユーザーストアと内部ユーザーオブジェクトモデルを橋渡しできます。

ユーザーログイン時などに Red Hat build of Keycloak ランタイムがユーザーを検索する必要がある場合は、複数の手順を実行してユーザーを特定します。まず、ユーザーキャッシュにユーザーが含まれているかどうかを確認します。ユーザーが見つかった場合にはユーザーがそのインメモリー表現を使用しているかを確認します。次に、Red Hat build of Keycloak ローカルデータベース内でユーザーを検索します。ユーザーが見つからない場合は、ランタイムが探しているユーザーをいずれかの実装が返すまで、User Storage SPI プロバイダー実装全体をループして、ユーザークエリーを実行し続けます。プロバイダーは、ユーザーの外部ユーザーストアをクエリーし、ユーザーの外部データ表現を Red Hat build of Keycloak のユーザーメタモデルにマップします。

User Storage SPI プロバイダー実装は、複雑な基準クエリーの実行、ユーザーへの CRUD 操作の実行、認証情報の検証および管理、または多数のユーザーの一括更新を実行することもできます。これは、外部ストアの機能により異なります。

User Storage SPI プロバイダー実装は、Jakarta EE コンポーネントと同様にパッケージ化およびデプロイされます。これらはデフォルトでは有効になっていませんが、管理コンソールの **User Federation** タブでレルムごとに有効および設定する必要があります。

警告

ユーザープロバイダーの実装で、ユーザー属性をユーザーアイデンティティーのリンク/確立のためのメタデータ属性として使用している場合は、ユーザーが属性を編集できないようにし、対応する属性が読み取り専用になっていることを確認してください。これは **LDAP_ID** 属性の例で、ビルトインの Red Hat build of Keycloak LDAP プロバイダーが LDAP サーバー側でユーザーの ID を保存するために使用しています。詳細は、脅威モデルの軽減策 の章を参照してください。

Red Hat build of Keycloak Quickstarts Repository に 2 つのサンプルプロジェクトがあります。各クイックスタートには **README** ファイルがあり、サンプルアプリケーションをビルド、デプロイ、およびテストする方法が記載されています。次の表は、利用可能な User Storage SPI クイックスタートの簡単な説明を示しています。

表6.1 User Storage SPI クイックスタート

名前	説明
user-storage-jpa	EJB と JPA を使用してユーザーストレージプロバイダーを実装する方法を説明しています。

名前	説明
user-storage-simple	ユーザー名とパスワードのキーペアが含まれる単純なプロパティーファイルを使用して、ユーザーストレージプロバイダーを実装する方法を説明しています。

6.1. プロバイダーインターフェイス

User Storage SPI の実装を構築する場合は、プロバイダークラスとプロバイダーファクトリーを定義する必要があります。プロバイダークラスインスタンスは、プロバイダーファクトリーによってトランザクションごとに作成されます。プロバイダークラスは、ユーザー検索やその他のユーザー操作で負荷が大きい作業をすべて行います。**org.keycloak.storage.UserStorageProvider** インターフェイスを実装する必要があります。

```
package org.keycloak.storage;

public interface UserStorageProvider extends Provider {


    /**
     * Callback when a realm is removed.  Implement this if, for example, you want to do some
     * cleanup in your user storage when a realm is removed
     *
     * @param realm
     */
    default
    void preRemove(RealmModel realm) {

    }

    /**
     * Callback when a group is removed.  Allows you to do things like remove a user
     * group mapping in your external store if appropriate
     *
     * @param realm
     * @param group
     */
    default
    void preRemove(RealmModel realm, GroupModel group) {

    }

    /**
     * Callback when a role is removed.  Allows you to do things like remove a user
     * role mapping in your external store if appropriate

     * @param realm
     * @param role
     */
    default
    void preRemove(RealmModel realm, RoleModel role) {
```

```
    }

}
```

UserStorageProvider インターフェイスはほんのわずかであると思われるかもしれません。本章では後で、ユーザー統合のサポート向けにプロバイダーが実装する可能性のあるさまざまなインターフェイスが他にあることを説明します。

UserStorageProvider インスタンスは、トランザクションごとに 1 度作成されます。トランザクションが完了すると、**UserStorageProvider.close()** メソッドが呼び出され、インスタンスはガベージコレクションされます。インスタンスはプロバイダーファクトリーによって作成されます。プロバイダーファクトリーは **org.keycloak.storage.UserStorageProviderFactory** インターフェイスを実装します。

```
package org.keycloak.storage;

/**
 * @author <a href="mailto:bill@burkecentral.com">Bill Burke</a>
 * @version $Revision: 1 $
 */
public interface UserStorageProviderFactory<T extends UserStorageProvider> extends
ComponentFactory<T, UserStorageProvider> {

    /**
     * This is the name of the provider and will be shown in the admin console as an option.
     *
     * @return
     */
    @Override
    String getId();

    /**
     * called per Keycloak transaction.
     *
     * @param session
     * @param model
     * @return
     */
    T create(KeycloakSession session, ComponentModel model);
...
}
```

プロバイダーファクトリークラスは、**UserStorageProviderFactory** を実装する際に、具体的なプロバイダークラスをテンプレートパラメーターとして指定する必要があります。これはランタイムがこのクラスをイントロスペクションしてその機能 (実装する他のインターフェイス) をスキャンするため必要です。たとえば、プロバイダークラスの名前が **FileProvider** の場合、ファクトリークラスは以下のようになります。

```
public class FileProviderFactory implements UserStorageProviderFactory<FileProvider> {

    public String getId() { return "file-provider"; }

    public FileProvider create(KeycloakSession session, ComponentModel model) {
       ...
    }
```

getId() メソッドは、User Storage プロバイダーの名前を返します。この ID は、特定のレルムのプロバイダーを有効にする際に管理コンソールのユーザーフェデレーションページに表示されます。

create() メソッドは、プロバイダークラスのインスタンスを割り当てます。**org.keycloak.models.KeycloakSession** パラメーターを取ります。このオブジェクトは、その他の情報およびメタデータを検索し、ランタイム内の他のコンポーネントへのアクセスを提供するために使用できます。**ComponentModel** パラメーターは、プロバイダーが特定のレルム内でどのように有効および設定されたかを表します。これには、有効化されたプロバイダーのインスタンス ID と、管理コンソールで有効にした場合に指定できる設定が含まれます。

UserStorageProviderFactory には他の機能があり、この後、本章で取り上げます。

6.2. プロバイダー機能のインターフェイス

UserStorageProvider インターフェイスを詳しく調べると、ユーザーを検索または管理するための方法を定義しないことが分かります。これらの方法は、外部ユーザーストアが提供および実行可能な機能の範囲に応じて、実際に他の **機能インターフェイス** で定義されます。たとえば、外部ストアの一部は読み取り専用で、単純なクエリーおよび認証情報の検証のみを実行できます。可能な機能の **機能インターフェイス** を実装することのみが必要になります。これらのインターフェイスを実装することができます。

SPI	説明
org.keycloak.storage.user.UserLookupProvider	この外部ストアのユーザーを使用してログインできるようにする場合には、このインターフェイスが必要です。大半 (またはすべて) プロバイダーはこのインターフェイスを実装します。
org.keycloak.storage.user.UserQueryMethodsProvider	1つ以上のユーザーの特定に使用する複雑なクエリーを定義します。管理コンソールからユーザーを表示および管理する場合は、このインターフェイスを実装する必要があります。
org.keycloak.storage.user.UserCountMethodsProvider	プロバイダーがクエリーのカウントをサポートしている場合は、このインターフェイスを実装します。
org.keycloak.storage.user.UserQueryProvider	このインターフェイスは、**UserQueryMethodsProvider** と **UserCountMethodsProvider** の機能を組み合わせたものです。
org.keycloak.storage.user.UserRegistrationProvider	プロバイダーがユーザーの追加および削除に対応している場合は、このインターフェイスを実装します。
org.keycloak.storage.user.UserBulkUpdateProvider	プロバイダーがユーザーセットの一括更新をサポートする場合は、このインターフェイスを実装します。

SPI	説明
org.keycloak.credential.CredentialInputValidator	プロバイダーが 1 つ以上の異なる認証情報タイプを検証できる場合は、このインターフェイスを実装します (たとえば、プロバイダーがパスワードを検証できる場合など)。
org.keycloak.credential.CredentialInputUpdater	プロバイダーが 1 つ以上の異なる認証情報タイプの更新をサポートする場合は、このインターフェイスを実装します。

6.3. モデルインターフェイス

機能 インターフェイス で定義されるメソッドのほとんどは、ユーザーの表現を返すか、渡されます。これらの表現は、**org.keycloak.models.UserModel** インターフェイスで定義されます。アプリケーション開発者は、このインターフェイスを実装する必要があります。Red Hat build of Keycloak が使用する外部ユーザーストアとユーザーメタモデル間でマッピングを行います。

```
package org.keycloak.models;

public interface UserModel extends RoleMapperModel {
    String getId();

    String getUsername();
    void setUsername(String username);

    String getFirstName();
    void setFirstName(String firstName);

    String getLastName();
    void setLastName(String lastName);

    String getEmail();
    void setEmail(String email);
...
}
```

UserModel 実装は、ユーザー名、名前、電子メール、ロール、グループマッピング、その他の任意の属性などのユーザーについてのメタデータの読み取りおよび更新へのアクセスを提供します。

org.keycloak.models パッケージには、Red Hat build of Keycloak メタモデルの他の部分を表す他のモデルクラスがあります (**RealmModel**、**RoleModel**、**GroupModel**、および **ClientModel**)。

6.3.1. ストレージ ID

UserModel の重要なメソッドの 1 つとして、**getId()** メソッドがあります。**UserModel** の開発者は、ユーザー ID 形式を認識する必要があります。形式は以下のとおりです。

```
"f:" + component id + ":" + external id
```

Red Hat build of Keycloak ランタイムは、多くの場合、ユーザー ID でユーザーを検索する必要があります。ユーザー ID には十分な情報が含まれているため、ランタイムは、システム内のすべての **UserStorageProvider** を紹介して、ユーザーを見つける必要はありません。

コンポーネント ID は、**ComponentModel.getId()** から返される ID です。**ComponentModel** はプロバイダークラスの作成時にパラメーターとして渡され、そこから取得できるようにします。外部 ID は、プロバイダークラスが外部ストアでのユーザー検索に必要な情報です。通常これはユーザー名または uid です。たとえば、以下のようになります。

```
f:332a234e31234:wburke
```

ランタイムが ID でルックアップを行う場合、ID はコンポーネント ID を取得するよう解析されます。コンポーネント ID は、ユーザーの読み込みに最初に使用された **UserStorageProvider** を見つけるために使用されます。その後、そのプロバイダーには ID が渡されます。プロバイダーが再び ID を解析して外部 ID を取得し、外部ユーザーストレージでのユーザー検索に使用されます。

6.4. パッケージおよびデプロイメント

Red Hat build of Keycloak がプロバイダーを認識できるようにするには、ファイルを JAR (**META-INF/services/org.keycloak.storage.UserStorageProviderFactory**) に追加する必要があります。このファイルには、**UserStorageProviderFactory** 実装の完全修飾クラス名の行区切りリストが含まれている必要があります。

```
org.keycloak.examples.federation.properties.ClasspathPropertiesStorageFactory
org.keycloak.examples.federation.properties.FilePropertiesStorageFactory
```

この jar をデプロイするには、**providers/** ディレクトリーにコピーしてから **bin/kc.[sh|bat] build** を実行します。

6.5. シンプルな読み取り専用、ルックアップの例

User Storage SPI を実装する際の基本事項について、簡単な例を追って説明します。本章では、簡単なプロパティーファイルでユーザーを検索する簡単な **UserStorageProvider** の実装を説明します。プロパティーファイルにはユーザー名とパスワードの定義が含まれ、クラスパス上の特定の場所にハードコーディングされます。プロバイダーは、ID およびユーザー名でユーザーを検索でき、パスワードを検証することもできます。このプロバイダーから発信されるユーザーは読み取り専用です。

6.5.1. プロバイダークラス

最初に説明するのは **UserStorageProvider** クラスです。

```
public class PropertyFileUserStorageProvider implements
        UserStorageProvider,
        UserLookupProvider,
        CredentialInputValidator,
        CredentialInputUpdater
{
...
}
```

プロバイダークラス **PropertyFileUserStorageProvider** は多くのインターフェイスを実装します。SPI のベース要件であるように **UserStorageProvider** を実装します。このプロバイダーが保存したユーザーでログインできるため、**UserLookupProvider** インターフェイスを実装します。ログイン画面で入

力されたパスワードを検証する必要があるため、**CredentialInputValidator** インターフェイスを実装します。プロパティーファイルは読み取り専用です。ユーザーがパスワードを更新しようとするとエラー条件を投稿するため、**CredentialInputUpdater** を実装します。

```
    protected KeycloakSession session;
    protected Properties properties;
    protected ComponentModel model;
    // map of loaded users in this transaction
    protected Map<String, UserModel> loadedUsers = new HashMap<>();

    public PropertyFileUserStorageProvider(KeycloakSession session, ComponentModel model,
Properties properties) {
        this.session = session;
        this.model = model;
        this.properties = properties;
    }
```

このプロバイダークラスのコンストラクターは **KeycloakSession**、**ComponentModel**、およびプロパティーファイルへの参照を保存します。後ほどこれらすべてを使用します。また、読み込んだユーザーのマップがあることにも注意してください。ユーザーを特定すると、このマップに保存し、同じトランザクションで再度ユーザーが作成されないようにします。多数のプロバイダーがこれを行う必要があるので (JPA と統合しているプロバイダー)、この慣習は適切です。また、プロバイダークラスインスタンスはトランザクションごとに 1 度作成され、トランザクションの完了後に閉じられることにも注意してください。

6.5.1.1. UserLookupProvider 実装

```
    @Override
    public UserModel getUserByUsername(RealmModel realm, String username) {
        UserModel adapter = loadedUsers.get(username);
        if (adapter == null) {
            String password = properties.getProperty(username);
            if (password != null) {
                adapter = createAdapter(realm, username);
                loadedUsers.put(username, adapter);
            }
        }
        return adapter;
    }

    protected UserModel createAdapter(RealmModel realm, String username) {
        return new AbstractUserAdapter(session, realm, model) {
            @Override
            public String getUsername() {
                return username;
            }
        };
    }

    @Override
    public UserModel getUserById(RealmModel realm, String id) {
        StorageId storageId = new StorageId(id);
        String username = storageId.getExternalId();
        return getUserByUsername(realm, username);
    }
```

```
    @Override
    public UserModel getUserByEmail(RealmModel realm, String email) {
        return null;
    }
```

getUserByUsername() メソッドは、ユーザーがログインすると Red Hat build of Keycloak ログインページにより呼び出されます。実装では最初に **loadedUsers** マップを確認し、ユーザーがこのトランザクション内にすでに読み込まれているかどうかを確認します。読み込まれていない場合は、ユーザー名のプロパティーファイルを探します。**UserModel** の実装を作成する場合は、将来のの参照のために **loadedUsers** に保存し、このインスタンスを返します。

createAdapter() メソッドはヘルパークラス **org.keycloak.storage.adapter.AbstractUserAdapter** を使用します。これにより、**UserModel** のベース実装が提供されます。外部 ID としてユーザーのユーザー名を使用して、必要なストレージ ID 形式に基づいてユーザー ID を自動的に生成します。

```
"f:" + component id + ":" + username
```

AbstractUserAdapter のすべての get メソッドは、Null または空のコレクションを返します。ただし、ロールマッピングおよびグループマッピングを返すメソッドは、すべてのユーザーのレルムに設定されたデフォルトのロールおよびグループを返します。**AbstractUserAdapter** のすべてのセットメソッドは、**org.keycloak.storage.ReadOnlyException** を発生させます。そのため、管理コンソールでユーザーを変更しようとすると、エラーが発生します。

getUserById() メソッドは、**org.keycloak.storage.StorageId** ヘルパークラスを使用して **id** パラメーターを解析します。**StorageId.getExternalId()** メソッドが呼び出され、**id** パラメーターに埋め込まれたユーザー名を取得します。その後、メソッドは **getUserByUsername()** に委譲します。

メールは保存されないため、**getUserByEmail()** メソッドは null を返します。

6.5.1.2. CredentialInputValidator 実装

次に、**CredentialInputValidator** のメソッド実装を確認します。

```
    @Override
    public boolean isConfiguredFor(RealmModel realm, UserModel user, String credentialType) {
        String password = properties.getProperty(user.getUsername());
        return credentialType.equals(PasswordCredentialModel.TYPE) && password != null;
    }

    @Override
    public boolean supportsCredentialType(String credentialType) {
        return credentialType.equals(PasswordCredentialModel.TYPE);
    }

    @Override
    public boolean isValid(RealmModel realm, UserModel user, CredentialInput input) {
        if (!supportsCredentialType(input.getType())) return false;

        String password = properties.getProperty(user.getUsername());
        if (password == null) return false;
        return password.equals(input.getChallengeResponse());
    }
```

isConfiguredFor() メソッドはランタイムによって呼び出され、特定の認証情報タイプがユーザーに設定されているかどうかを判断します。この方法は、パスワードがユーザーに設定されていることを確認します。

supportsCredentialType() メソッドは、特定の認証情報タイプに対して検証がサポートされるかどうかを返します。認証情報のタイプが **password** であるかを確認します。

isValid() メソッドはパスワードを検証します。**CredentialInput** パラメーターは、すべての認証情報タイプの抽象インターフェイスにすぎません。認証情報タイプもサポートし、**UserCredentialModel** のインスタンスであることを確認します。ユーザーがログインページでログインすると、**UserCredentialModel** のインスタンスに入力されたパスワード入力のプレーンテキストが追加されます。**isValid()** メソッドは、この値をプロパティーファイルに保存されているプレーンテキストのパスワードと照合します。戻り値 **true** はパスワードが有効であることを意味します。

6.5.1.3. CredentialInputUpdater 実装

前述したように、**CredentialInputUpdater** インターフェイスは、ユーザーパスワードの変更を禁止するためだけに実装します。そうしなければ Red Hat build of Keycloak のローカルストレージでランタイムによりパスワードを上書きできるため、このような対応が必要です。本章の後半で詳しく説明します。

```
    @Override
    public boolean updateCredential(RealmModel realm, UserModel user, CredentialInput input) {
        if (input.getType().equals(PasswordCredentialModel.TYPE)) throw new
ReadOnlyException("user is read only for this update");

        return false;
    }

    @Override
    public void disableCredentialType(RealmModel realm, UserModel user, String credentialType) {

    }

    @Override
    public Stream<String> getDisableableCredentialTypesStream(RealmModel realm, UserModel
user) {
        return Stream.empty();
    }
```

updateCredential() メソッドは、認証情報タイプがパスワードであるかどうかを確認します。存在する場合は、**ReadOnlyException** が発生します。

6.5.2. プロバイダーファクトリーの実装

プロバイダークラスが完成したので、プロバイダーファクトリークラスを見ていきます。

```
public class PropertyFileUserStorageProviderFactory
            implements UserStorageProviderFactory<PropertyFileUserStorageProvider> {

    public static final String PROVIDER_NAME = "readonly-property-file";

    @Override
```

```
    public String getId() {
        return PROVIDER_NAME;
    }
```

最初に **UserStorageProviderFactory** クラスを実装する場合は、具体的なプロバイダークラス実装をテンプレートパラメーターとして渡す必要があります。ここで、以前に定義したプロバイダークラス (**PropertyFileUserStorageProvider**) を指定します。

警告

template パラメーターを指定しない場合には、プロバイダーは機能しません。ランタイムはクラスのイントロスペクションを実行し、プロバイダーが実装する **機能インターフェイス** を判別します。

getId() メソッドはランタイムのファクトリーを識別し、レルムのユーザーストレージプロバイダーを有効にする場合も管理コンソールに表示される文字列になります。

6.5.2.1. 初期化

```
    private static final Logger logger =
Logger.getLogger(PropertyFileUserStorageProviderFactory.class);
    protected Properties properties = new Properties();

    @Override
    public void init(Config.Scope config) {
        InputStream is = getClass().getClassLoader().getResourceAsStream("/users.properties");

        if (is == null) {
            logger.warn("Could not find users.properties in classpath");
        } else {
            try {
                properties.load(is);
            } catch (IOException ex) {
                logger.error("Failed to load users.properties file", ex);
            }
        }
    }

    @Override
    public PropertyFileUserStorageProvider create(KeycloakSession session, ComponentModel
model) {
        return new PropertyFileUserStorageProvider(session, model, properties);
    }
```

UserStorageProviderFactory インターフェイスには、実装できる任意の **init()** メソッドがあります。Red Hat build of Keycloak の起動時に、各プロバイダーファクトリーのインスタンスが 1 つだけ作成されます。また、システムの起動時に、これらの各ファクトリーのインスタンスでも **init()** メソッドが呼び出されます。同様に実装できる **postInit()** メソッドも実装できます。各ファクトリーの **init()** メソッドが呼び出されると、**postInit()** メソッドが呼び出されます。

init() メソッド実装では、クラスパスからユーザー宣言が含まれるプロパティーファイルを見つけます。次に **properties** フィールドを、そこに保存されたユーザー名とパスワードの組み合わせでロードします。

Config.Scope パラメーターは、サーバー設定を介して設定されるファクトリー設定です。

たとえば、次の引数を使用してサーバーを実行します。

```
kc.[sh|bat] start --spi-storage-readonly-property-file-path=/other-users.properties
```

ハードコーディングするのではなく、ユーザープロパティーファイルのクラスパスを指定できます。次に、**PropertyFileUserStorageProviderFactory.init()** で設定を取得できます。

```
public void init(Config.Scope config) {
    String path = config.get("path");
    InputStream is = getClass().getClassLoader().getResourceAsStream(path);

    ...
}
```

6.5.2.2. 作成方法

プロバイダーファクトリー作成における最後の手順は **create()** メソッドです。

```
    @Override
    public PropertyFileUserStorageProvider create(KeycloakSession session, ComponentModel
model) {
        return new PropertyFileUserStorageProvider(session, model, properties);
    }
```

PropertyFileUserStorageProvider クラスを割り当てるだけです。この create メソッドはトランザクションごとに 1 回呼び出されます。

6.5.3. パッケージおよびデプロイメント

プロバイダー実装のクラスファイルは jar に配置する必要があります。また、**META-INF/services/org.keycloak.storage.UserStorageProviderFactory** ファイルでプロバイダーファクトリークラスを宣言する必要があります。

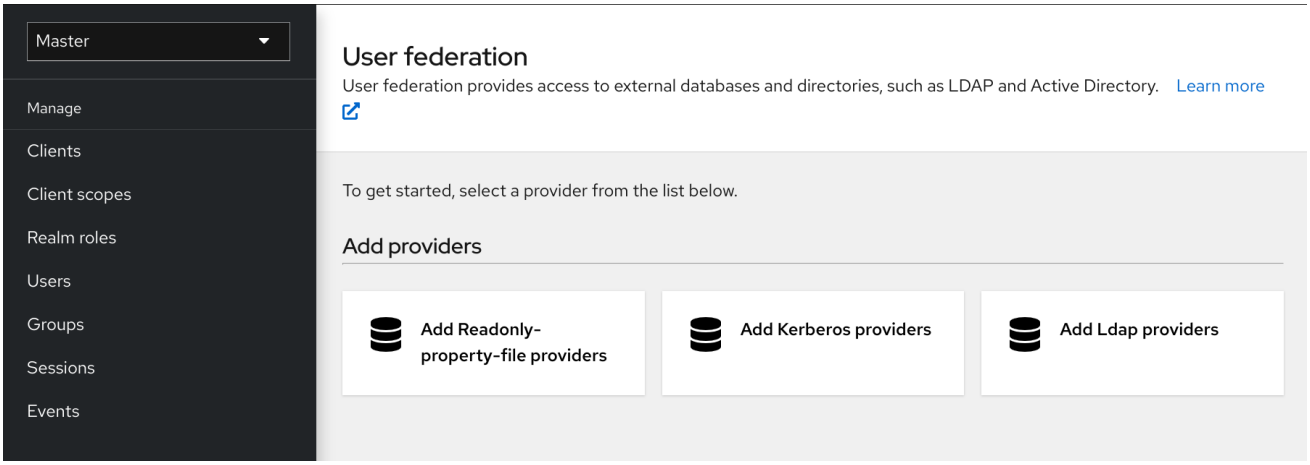
```
org.keycloak.examples.federation.properties.FilePropertiesStorageFactory
```

この jar をデプロイするには、**providers/** ディレクトリーにコピーしてから **bin/kc.[sh|bat] build** を実行します。

6.5.4. 管理コンソールでのプロバイダーの有効化

管理コンソールの **User Federation** ページで、レルムごとにユーザーストレージプロバイダーを有効にします。

ユーザーフェデレーション

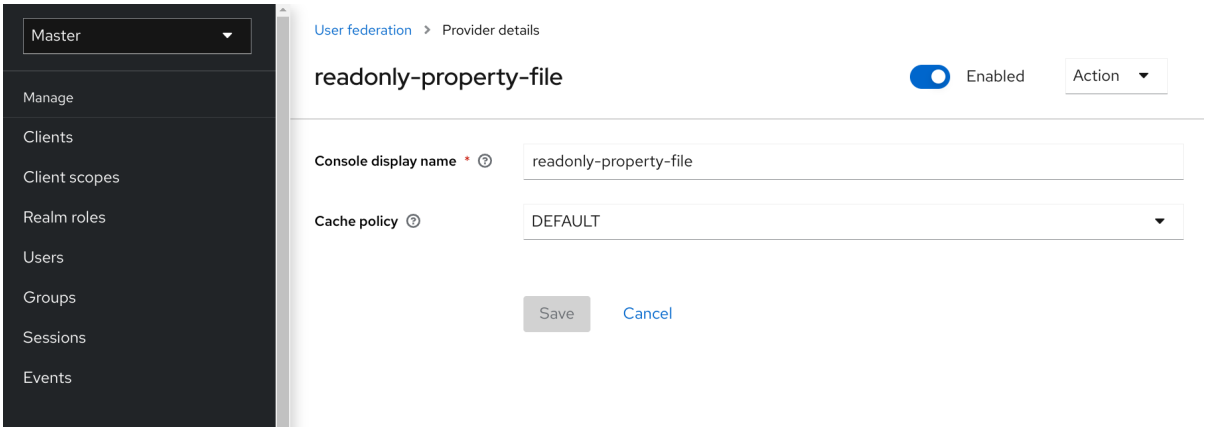


手順

1. **readonly-property-file** リストから作成したプロバイダーを選択します。
プロバイダーの設定ページが表示されます。

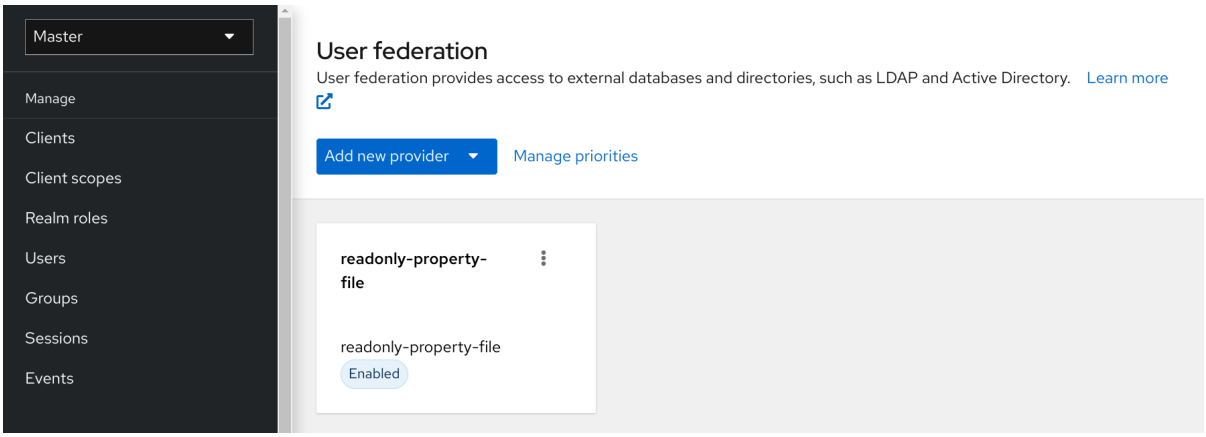
2. 設定するものがないため、**Save** をクリックします。

設定されたプロバイダー



3. **User Federation** のメインページに戻ります。
これで、プロバイダーがリスト表示されます。

ユーザーフェデレーション



これで、**users.properties** ファイルで宣言されたユーザーでログインできるようになります。このユーザーは、ログイン後にのみアカウントページを表示できます。

6.6. 設定方法

PropertyFileUserStorageProvider の例は少々工夫されています。これは、プロバイダーの jar に組み込まれているプロパティーファイルにハードコーディングされているので、あまり便利ではありません。プロバイダーのインスタンスごとにこのファイルの場所を設定する場合があります。言い換えると、このプロバイダーを複数の異なるレルムで複数回再利用したり、全く異なるユーザープロパティーファイルを参照する場合などです。また、管理コンソール UI 内でこの設定を実行する必要があります。

UserStorageProviderFactory には、プロバイダー設定を実装できる追加のメソッドがあります。プロバイダーごとに設定する変数を記述すると、管理コンソールは自動的に一般的な入力ページをレンダリングしてこの設定を収集します。実装されている場合、コールバックメソッドは、設定が保存される前、プロバイダーが初めて作成されるとき、および更新されるときにも検証を行います。**UserStorageProviderFactory** は、**org.keycloak.component.ComponentFactory** インターフェイスからこれらのメソッドを継承します。

```
    List<ProviderConfigProperty> getConfigProperties();

    default
    void validateConfiguration(KeycloakSession session, RealmModel realm, ComponentModel
model)
            throws ComponentValidationException
    {

    }

    default
    void onCreate(KeycloakSession session, RealmModel realm, ComponentModel model) {

    }

    default
    void onUpdate(KeycloakSession session, RealmModel realm, ComponentModel model) {

    }
```

ComponentFactory.getConfigProperties() メソッドは、**org.keycloak.provider.ProviderConfigProperty** インスタンスのリストを返します。これらのインスタンスは、プロバイダーの各設定変数のレンダリングおよび保存に必要なメタデータを宣言します。

6.6.1. 設定例

PropertyFileUserStorageProviderFactory の例を拡張して、プロバイダーインスタンスをディスク上の特定のファイルを指定できるようにします。

PropertyFileUserStorageProviderFactory

```
public class PropertyFileUserStorageProviderFactory
                implements UserStorageProviderFactory<PropertyFileUserStorageProvider> {

    protected static final List<ProviderConfigProperty> configMetadata;

    static {
        configMetadata = ProviderConfigurationBuilder.create()
                .property().name("path")
```

```
                .type(ProviderConfigProperty.STRING_TYPE)
                .label("Path")
                .defaultValue("${jboss.server.config.dir}/example-users.properties")
                .helpText("File path to properties file")
                .add().build();
    }

    @Override
    public List<ProviderConfigProperty> getConfigProperties() {
        return configMetadata;
    }
```

ProviderConfigurationBuilder クラスは、設定プロパティーのリストを作成するためのヘルパークラスです。ここでは、String タイプである **path** という名前の変数を指定します。このプロバイダーの管理コンソール設定ページでは、この設定変数は **Path** としてラベル付けされ、デフォルト値は **${jboss.server.config.dir}/example-users.properties** です。この設定オプションのツールチップにカーソルを合わせると、ヘルプテキスト (**File path to properties file (プロパティーファイルへのファイルパス)**) が表示されます。

次に、このファイルがディスクに存在することを確認します。有効なユーザープロパティーファイルを参照しない限り、レルムでこのプロバイダーのインスタンスを有効するべきではありません。これを実行するには、**validateConfiguration()** メソッドを実装します。

```
    @Override
    public void validateConfiguration(KeycloakSession session, RealmModel realm, ComponentModel
config)
                throws ComponentValidationException {
        String fp = config.getConfig().getFirst("path");
        if (fp == null) throw new ComponentValidationException("user property file does not exist");
        fp = EnvUtil.replace(fp);
        File file = new File(fp);
        if (!file.exists()) {
            throw new ComponentValidationException("user property file does not exist");
        }
    }
```

validateConfiguration() メソッドは、**ComponentModel** から設定変数を提供して、そのファイルがディスク上に存在するかどうかを確認します。**org.keycloak.common.util.EnvUtil.replace()** メソッドを使用していることに注目してください。このメソッドを使用すると、**${}** を含む文字列で、その値がシステムプロパティー値に置き換えられます。**${jboss.server.config.dir}** 文字列はサーバーの **conf/** ディレクトリーに対応するものであり、この例では非常に役立ちます。

次に、古い **init()** メソッドを削除します。ユーザープロパティーファイルはプロバイダーインスタンスごとに一意であるため、これを行います。このロジックは **create()** メソッドに移動します。

```
    @Override
    public PropertyFileUserStorageProvider create(KeycloakSession session, ComponentModel
model) {
        String path = model.getConfig().getFirst("path");

        Properties props = new Properties();
        try {
            InputStream is = new FileInputStream(path);
            props.load(is);
            is.close();
```

```
        } catch (IOException e) {
            throw new RuntimeException(e);
        }

        return new PropertyFileUserStorageProvider(session, model, props);
    }
```

全トランザクションでユーザープロパティーファイル全体をディスクから読み取るので、当然このロジックは効率的ではありませんが、今回の例で、設定変数をフックする方法をシンプルに説明できたかと思います。

6.6.2. 管理コンソールでのプロバイダーの設定

設定が有効になったため、管理コンソールでプロバイダーを設定する際に **path** 変数を設定できます。

6.7. ユーザーおよびクエリー機能インターフェイスの追加/削除

今回の例で対応していない内容の1つとして、ユーザーやパスワードの追加および削除を可能にする操作です。また、この例で定義されたユーザーは、管理コンソールでクエリーや表示ができません。このような拡張機能を追加するには、example プロバイダーが **UserQueryMethodsProvider** インターフェイス (または **UserQueryProvider**) および **UserRegistrationProvider**インターフェイスを実装する必要があります。

6.7.1. UserRegistrationProvider の実装

この手順を使用して、特定のストアからユーザーの追加および削除を実装します。その場合、最初にプロパティーファイルをディスクに保存する必要があります。

PropertyFileUserStorageProvider

```
    public void save() {
        String path = model.getConfig().getFirst("path");
        path = EnvUtil.replace(path);
        try {
            FileOutputStream fos = new FileOutputStream(path);
            properties.store(fos, "");
            fos.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
```

次に、**addUser()** メソッドおよび **removeUser()** メソッドの実装が簡単になります。

PropertyFileUserStorageProvider

```
    public static final String UNSET_PASSWORD="#$!-UNSET-PASSWORD";

    @Override
    public UserModel addUser(RealmModel realm, String username) {
        synchronized (properties) {
            properties.setProperty(username, UNSET_PASSWORD);
            save();
        }
```

```
        return createAdapter(realm, username);
    }

    @Override
    public boolean removeUser(RealmModel realm, UserModel user) {
        synchronized (properties) {
            if (properties.remove(user.getUsername()) == null) return false;
            save();
            return true;
        }
    }
```

ユーザーを追加する場合は、プロパティーマップのパスワード値を **UNSET_PASSWORD** に設定することに注意してください。これを実行するのは、プロパティー値に null 値を指定できないためです。また、これを反映するように **CredentialInputValidator** メソッドを変更する必要もあります。

プロバイダーが **UserRegistrationProvider** インターフェイスを実装している場合は、**addUser()** メソッドが呼び出されます。プロバイダーにユーザーの追加をオフにする設定スイッチがある場合は、このメソッドから **null** を返すとこのプロバイダーが飛ばされ、次のプロバイダーを呼び出します。

PropertyFileUserStorageProvider

```
    @Override
    public boolean isValid(RealmModel realm, UserModel user, CredentialInput input) {
        if (!supportsCredentialType(input.getType()) || !(input instanceof UserCredentialModel)) return
false;

        UserCredentialModel cred = (UserCredentialModel)input;
        String password = properties.getProperty(user.getUsername());
        if (password == null || UNSET_PASSWORD.equals(password)) return false;
        return password.equals(cred.getValue());
    }
```

プロパティーファイルを保存できるようになったため、パスワードの更新を許可しても問題ありません。

PropertyFileUserStorageProvider

```
    @Override
    public boolean updateCredential(RealmModel realm, UserModel user, CredentialInput input) {
        if (!(input instanceof UserCredentialModel)) return false;
        if (!input.getType().equals(PasswordCredentialModel.TYPE)) return false;
        UserCredentialModel cred = (UserCredentialModel)input;
        synchronized (properties) {
            properties.setProperty(user.getUsername(), cred.getValue());
            save();
        }
        return true;
    }
```

パスワードの無効化も実装できるようになりました。

PropertyFileUserStorageProvider

```
    @Override
```

```
    public void disableCredentialType(RealmModel realm, UserModel user, String credentialType) {
        if (!credentialType.equals(PasswordCredentialModel.TYPE)) return;
        synchronized (properties) {
            properties.setProperty(user.getUsername(), UNSET_PASSWORD);
            save();
        }

    }

    private static final Set<String> disableableTypes = new HashSet<>();

    static {
        disableableTypes.add(PasswordCredentialModel.TYPE);
    }

    @Override
    public Stream<String> getDisableableCredentialTypes(RealmModel realm, UserModel user) {

        return disableableTypes.stream();
    }
```

これらのメソッドを実装すると、管理コンソールでユーザーのパスワードを変更および無効化できるようになりました。

6.7.2. UserQueryProvider の実装

UserQueryProvider は、**UserQueryMethodsProvider** と **UserCountMethodsProvider** を組み合わせたものです。**UserQueryMethodsProvider** を実装しないと、管理コンソールは example プロバイダーによって読み込まれたユーザーを表示および管理できません。このインターフェイスの実装を見てみましょう。

PropertyFileUserStorageProvider

```
    @Override
    public int getUsersCount(RealmModel realm) {
        return properties.size();
    }

    @Override
    public Stream<UserModel> searchForUserStream(RealmModel realm, String search, Integer
firstResult, Integer maxResults) {
        Predicate<String> predicate = "*".equals(search) ? username -> true : username ->
username.contains(search);
        return properties.keySet().stream()
                .map(String.class::cast)
                .filter(predicate)
                .skip(firstResult)
                .map(username -> getUserByUsername(realm, username))
                .limit(maxResults);
    }
```

searchForUserStream() の最初の宣言は、**String** パラメーターを受け取ります。この例では、パラメーターは検索に使用するユーザー名を表します。この文字列は部分文字列にすることができます。そのため、検索を行う際には **String.contains()** メソッドを選択します。すべてのユーザーのリストを要求することを示すために * を使用していることに注目してください。このメソッドはプロパティーファ

イルのキーセットを繰り返し処理し、**getUserByUsername()** にユーザーを読み込みます。**firstResult** パラメーターおよび **maxResults** パラメーターに基づいてこの呼び出しにインデックス化していることに留意してください。外部ストアがページネーションをサポートしない場合には、同様のロジックを実行する必要があります。

PropertyFileUserStorageProvider

```
    @Override
    public Stream<UserModel> searchForUserStream(RealmModel realm, Map<String, String>
params, Integer firstResult, Integer maxResults) {
        // only support searching by username
        String usernameSearchString = params.get("username");
        if (usernameSearchString != null)
            return searchForUserStream(realm, usernameSearchString, firstResult, maxResults);

        // if we are not searching by username, return all users
        return searchForUserStream(realm, "*", firstResult, maxResults);
    }
```

Map パラメーターを取る **searchForUserStream()** メソッドは、姓名、ユーザー名、およびメールに基づいてユーザーを検索できます。ユーザー名のみが保存されるため、**Map** パラメーターに **username** 属性が含まれていない場合を除き、検索はユーザー名のみに基づいて行われます。この場合、すべてのユーザーが返されます。このような場合、**searchForUserStream(realm, search, firstResult, maxResults)** が使用されます。

PropertyFileUserStorageProvider

```
    @Override
    public Stream<UserModel> getGroupMembersStream(RealmModel realm, GroupModel group,
Integer firstResult, Integer maxResults) {
        return Stream.empty();
    }

    @Override
    public Stream<UserModel> searchForUserByUserAttributeStream(RealmModel realm, String
attrName, String attrValue) {
        return Stream.empty();
    }
```

グループまたは属性は保存されないため、他のメソッドは空のストリームを返します。

6.8. 外部ストレージの拡張

PropertyFileUserStorageProvider 例は実際に制限されています。プロパティーファイルに保存されているユーザーでログインできますが、それ以外にできることはほぼありません。このプロバイダーがロードしたユーザーが、特定のアプリケーションに完全にアクセスできるように、特別なロールまたはグループマッピングが必要な場合は、これらのユーザーに別のロールマッピングを追加する方法はありません。また、電子メール、名、姓などの重要な属性を変更したり、追加したりすることはできません。

このような状況では、Red Hat build of Keycloak のデータベースに追加情報を保存して外部ストアを拡張できます。これは フェデレーション (連合型) ユーザーストレージと呼ばれ、**org.keycloak.storage.federated.UserFederatedStorageProvider** 内でカプセル化されます。

UserFederatedStorageProvider

```
package org.keycloak.storage.federated;

public interface UserFederatedStorageProvider extends Provider,
        UserAttributeFederatedStorage,
        UserBrokerLinkFederatedStorage,
        UserConsentFederatedStorage,
        UserNotBeforeFederatedStorage,
        UserGroupMembershipFederatedStorage,
        UserRequiredActionsFederatedStorage,
        UserRoleMappingsFederatedStorage,
        UserFederatedUserCredentialStore {
    ...

}
```

UserFederatedStorageProvider インスタンスは、**UserStorageUtil.userFederatedStorage(KeycloakSession)** メソッドで利用できます。これには、あらゆる種類のメソッドがあり、属性、グループおよびロールマッピング、さまざまな認証情報タイプ、および必要なアクションを保存できます。外部ストアのデータモデルが、Red Hat build of Keycloak 機能セットに完全対応していない場合、このサービスを使用することで対応できます。

Red Hat build of Keycloak にはヘルパークラス **org.keycloak.storage.adapter.AbstractUserAdapterFederatedStorage** が同梱されています。このクラスは、ユーザー名の get/set を除くすべての **UserModel** メソッドを、ユーザーフェデレーションストレージに委譲します。外部ストレージ表現に委譲するのに必要なメソッドを上書きします。上書きする保護メソッドが少ないため、このクラスの javadoc を参照することを強く推奨します。具体的には、グループメンバーシップおよびロールマッピング関連です。

6.8.1. 拡張例

PropertyFileUserStorageProvider の例では、**AbstractUserAdapterFederatedStorage** を使用するためにプロバイダーに簡単な変更が必要になります。

PropertyFileUserStorageProvider

```
    protected UserModel createAdapter(RealmModel realm, String username) {
        return new AbstractUserAdapterFederatedStorage(session, realm, model) {
            @Override
            public String getUsername() {
                return username;
            }

            @Override
            public void setUsername(String username) {
                String pw = (String)properties.remove(username);
                if (pw != null) {
                    properties.put(username, pw);
                    save();
                }
            }
        };
    }
```

代わりに、**AbstractUserAdapterFederatedStorage** の匿名クラス実装を定義します。**setUsername()** メソッドはプロパティーファイルを変更し、これを保存します。

6.9. 実装ストラテジーのインポート

ユーザーストレージプロバイダーを実装する場合は、別のストラテジーを使用できます。ユーザーフェデレーションストレージを使用する代わりに、Red Hat build of Keycloak のビルトインユーザーデータベースでユーザーをローカルで作成し、外部ストアからこのローカルコピーに属性をコピーします。この方法には多くの利点があります。

- 基本的に、Red Hat build of Keycloak は外部ストアの永続ユーザーキャッシュになります。ユーザーがインポートされると、外部ストアに到達できなくなるため、負荷はなくなります。

- 公式ユーザーストアとして Red Hat build of Keycloak に移行し、古い外部ストアを非推奨にする場合は、アプリケーションを徐々に移行して Red Hat build of Keycloak を使用できます。すべてのアプリケーションが移行されたら、インポートされたユーザーのリンクを解除し、古いレガシー外部ストアを破棄します。

インポートストラテジーの使用には、いくつかの明確な欠点があります。

- 初めてユーザーを検索するには、Red Hat build of Keycloak データベースを複数回更新する必要があります。これを実行すると、負荷がかかってパフォーマンスが大幅に低下し、Red Hat build of Keycloak データベースに大きな負担がかかる可能性があります。ユーザーフェデレーションされたストレージアプローチは、必要に応じて追加のデータのみを保存し、外部ストアの機能によっては使用されない可能性があります。

- このインポート方法では、ローカルの Red Hat build of Keycloak ストレージと外部ストレージを同期する必要があります。User Storage SPI には同期をサポートするために実装できる機能インターフェイスがありますが、この操作はすぐに面倒で複雑になる可能性があります。

インポートストラテジーを実装するには、ユーザーがローカルにインポートされているかどうかを最初に確認します。ローカルにインポートされている場合には、ローカルユーザーを返します。インポートされていない場合には、ローカルにユーザーを作成して、外部ストアからデータをインポートします。また、ほとんどの変更が自動同期されるように、ローカルユーザーをプロキシー化することも可能です。

これは少し複雑になりますが、**PropertyFileUserStorageProvider** を拡張してこのアプローチを取ることができます。最初に **createAdapter()** メソッドを修正します。

PropertyFileUserStorageProvider

```
    protected UserModel createAdapter(RealmModel realm, String username) {
        UserModel local =
UserStoragePrivateUtil.userLocalStorage(session).getUserByUsername(realm, username);
        if (local == null) {
            local = UserStoragePrivateUtil.userLocalStorage(session).addUser(realm, username);
            local.setFederationLink(model.getId());
        }
        return new UserModelDelegate(local) {
            @Override
            public void setUsername(String username) {
                String pw = (String)properties.remove(username);
                if (pw != null) {
                    properties.put(username, pw);
                    save();
                }
```

```
            super.setUsername(username);
        }
    };
}
```

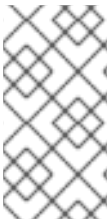
この方法では、**UserStoragePrivateUtil.userLocalStorage(session)** メソッドを呼び出して、ローカルの Red Hat build of Keycloak ユーザーストレージへの参照を取得します。ユーザーがローカルに保存されているかどうかを確認し、ない場合はローカルに追加します。ローカルユーザーの **id** を設定しないでください。**id** は、Red Hat build of Keycloak が自動生成します。また、**UserModel.setFederationLink()** を呼び出して、プロバイダーの **ComponentModel** に ID を渡すことにも注意してください。これにより、プロバイダーとインポートされたユーザーの間にリンクが設定されます。

注記

ユーザーストレージプロバイダーが削除されると、そのストレージプロバイダーによってインポートされたユーザーも削除されます。これは、**UserModel.setFederationLink()** を呼び出す目的の 1 つです。

また、ローカルユーザーがリンクしている場合は、**CredentialInputValidator** インターフェイスおよび **CredentialInputUpdater** インターフェイスから実装するメソッドのために、ストレージプロバイダーが依然として委譲される点に留意してください。検証または更新で **false** が返されると、Red Hat build of Keycloak はローカルストレージを使用して検証または更新できるか確認します。

また、**org.keycloak.models.utils.UserModelDelegate** クラスを使用してローカルユーザーをプロキシー処理している点に注意してください。このクラスは **UserModel** の実装です。すべてのメソッドは、それがインスタンス化された **UserModel** に委譲するだけです。プロパティーファイルと自動的に同期するため、この委譲クラスの **setUsername()** メソッドが上書きされます。プロバイダーの場合、これを使用して、ローカル **UserModel** 上の他のメソッドを **傍受** して、外部ストアと同期できます。たとえば、get メソッドでは、ローカルストアが同期していることを確認することができます。set メソッドでは、外部ストアがローカルストアと同期し続けます。**getId()** メソッドは、ユーザーをローカルで作成したときに自動生成された id を常に返す必要がある点に留意してください。インポート以外の他の例で示されているように、フェデレーション ID は返さないはずです。



注記

プロバイダーが **UserRegistrationProvider** インターフェイスを実装している場合、**removeUser()** メソッドはローカルストレージからユーザーを削除する必要はありません。ランタイムはこの操作を自動的に実行します。また、ローカルストレージから削除される前に **removeUser()** が呼び出されることに注意してください。

6.9.1. ImportedUserValidation インターフェイス

本章の最初の部分で、ユーザーへの問い合わせがどのように機能するかを説明しました。最初にローカルストレージを問い合わせし、ユーザーが見つかった場合は、クエリーが終了します。これは、上記の実装で問題になります。ローカル **UserModel** をプロキシーして、ユーザー名の同期を維持します。User Storage SPI には、リンクされたローカルユーザーがローカルデータベースから読み込まれるたびに実行されるコールバックがあります。

```
package org.keycloak.storage.user;
public interface ImportedUserValidation {
    /**
     * If this method returns null, then the user in local storage will be removed
     *
```

```
     * @param realm
     * @param user
     * @return null if user no longer valid
     */
    UserModel validate(RealmModel realm, UserModel user);
}
```

リンクされたローカルユーザーが読み込まれるたびに、ユーザーストレージプロバイダークラスがこのインターフェイスを実装している場合は、**validate()** メソッドが呼び出されます。ここでは、ローカルユーザーをパラメーターとしてプロキシー化して返すことができます。その新しい **UserModel** が使用されます。オプションで、ユーザーが外部ストアにまだ存在するかどうかを確認することもできます。**validate()** が **Null** を返すと、ローカルユーザーがこのデータベースから削除されます。

6.9.2. ImportSynchronization インターフェイス

インポートストラテジーにより、ローカルユーザーコピーが外部ストレージと同期できなくなっていることがわかります。おそらくユーザーは外部ストアから削除されてしまっているようです。ユーザーストレージ SPI には、これに対応するために実装できる追加のインターフェイス (**org.keycloak.storage.user.ImportSynchronization**) があります。

```
package org.keycloak.storage.user;

public interface ImportSynchronization {
    SynchronizationResult sync(KeycloakSessionFactory sessionFactory, String realmId,
UserStorageProviderModel model);
    SynchronizationResult syncSince(Date lastSync, KeycloakSessionFactory sessionFactory, String
realmId, UserStorageProviderModel model);
}
```

このインターフェイスはプロバイダーファクトリーによって実装されます。このインターフェイスがプロバイダーファクトリーによって実装されると、プロバイダーの管理コンソール管理ページには追加オプションが表示されます。ボタンをクリックして、手動で強制的に同期させることができます。これにより、**ImportSynchronization.sync()** メソッドが呼び出されます。また、同期を自動的にスケジュールできる追加の設定オプションが表示されます。自動同期は **syncSince()** メソッドを呼び出します。

6.10. ユーザーキャッシュ

ユーザーオブジェクトが ID、ユーザー名、またはメールクエリーによって読み込まれると、キャッシュされます。ユーザーオブジェクトがキャッシュされている場合、これは **UserModel** インターフェイス全体を繰り返し処理し、この情報をローカルのインメモリーのみのキャッシュにプルします。クラスターでは、このキャッシュは引き続きローカルに存在しますが、インバリデーションキャッシュになります。ユーザーオブジェクトが変更されると、これはエビクトされます。このエビクションイベントはクラスター全体に伝播され、他のノードのユーザーキャッシュも無効になります。

6.10.1. ユーザーキャッシュの管理

KeycloakSession.getProvider(UserCache.class) を呼び出してユーザーキャッシュにアクセスできます。

```
/**
 * All these methods effect an entire cluster of Keycloak instances.
 *
 * @author <a href="mailto:bill@burkecentral.com">Bill Burke</a>
 * @version $Revision: 1 $
```

```
 */
public interface UserCache extends UserProvider {
    /**
     * Evict user from cache.
     *
     * @param user
     */
    void evict(RealmModel realm, UserModel user);

    /**
     * Evict users of a specific realm
     *
     * @param realm
     */
    void evict(RealmModel realm);

    /**
     * Clear cache entirely.
     *
     */
    void clear();
}
```

特定のユーザー、特定のレルムに含まれるユーザー、またはキャッシュ全体をエビクトする方法があります。

6.10.2. OnUserCache Callback インターフェイス

プロバイダー実装に固有の追加情報をキャッシュすることもできます。ユーザーストレージ SPI には、ユーザーがキャッシュされるたびにのコールバック (**org.keycloak.models.cache.OnUserCache**) が設定されます。

```
public interface OnUserCache {
    void onCache(RealmModel realm, CachedUserModel user, UserModel delegate);
}
```

このコールバックが必要な場合は、プロバイダークラスはこのインターフェイスを実装する必要があります。**UserModel** パラメーターは、プロバイダーによって返される **UserModel** インスタンスです。**CachedUserModel** は、拡張された **UserModel** インターフェイスです。これは、ローカルストレージでローカルにキャッシュされるインスタンスです。

```
public interface CachedUserModel extends UserModel {

    /**
     * Invalidates the cache for this user and returns a delegate that represents the actual data provider
     *
     * @return
     */
    UserModel getDelegateForUpdate();

    boolean isMarkedForEviction();

    /**
     * Invalidate the cache for this model
     *
```

```
     */
    void invalidate();

    /**
     * When was the model was loaded from database.
     *
     * @return
     */
    long getCacheTimestamp();

    /**
     * Returns a map that contains custom things that are cached along with this model.  You can write
 to this map.
     *
     * @return
     */
    ConcurrentHashMap getCachedWith();
}
```

CachedUserModel インターフェイスを使用すると、ユーザーをキャッシュから強制的に削除 (エビクト) し、プロバイダーの **UserModel** インスタンスを取得できます。**getCachedWith()** メソッドは、ユーザーに関連する追加情報をキャッシュできるマップを返します。たとえば、認証情報は **UserModel** インターフェイスの一部ではありません。認証情報をメモリーにキャッシュする必要がある場合は **OnUserCache** を実装し、**getCachedWith()** メソッドを使用してユーザーの認証情報をキャッシュします。

6.10.3. キャッシュポリシー

ユーザーストレージプロバイダーの管理者向けのコンソール管理ページで、固有のキャッシュポリシーを指定できます。

6.11. JAKARTA EE の活用

バージョン 20 以降、Keycloak は Quarkus のみに依存します。WildFly とは異なり、Quarkus はアプリケーションサーバーではありません。詳細は、https://www.keycloak.org/migration/migrated-to-quarkus#_quarkus_is_not_an_application_server を参照してください。

したがって、以前のバージョンで Keycloak が WildFly 上で実行されていた場合のように、ユーザーストレージプロバイダーを Jakarta EE コンポーネント内にパッケージ化したり、EJB にしたりすることはできません。

プロバイダー実装は、前のセクションで説明したように、適切なユーザーストレージ SPI インターフェイスを実装するプレーンな Java オブジェクトである必要があります。また、この移行ガイドの説明に従ってパッケージ化してデプロイする必要があります。

- https://www.keycloak.org/migration/migrating-to-quarkus#_migrating_custom_providers

次の例に示すように、JPA Entity Manager によって外部データベースを統合できるカスタム **UserStorageProvider** クラスを引き続き実装できます。

- https://github.com/redhat-developer/rhbk-quickstarts/tree/24.x/extension/user-storage-jpa

CDI はサポートされていません。

6.12. REST 管理 API

管理者 REST API を使用して、ユーザーストレージプロバイダーのデプロイメントを作成、削除、および更新できます。User Storage SPI は汎用コンポーネントインターフェイス上に構築されるため、その汎用 API を使用してプロバイダーを管理します。

REST Component API は、レルム管理リソース下に存在します。

```
/admin/realms/{realm-name}/components
```

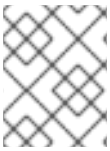
ここでは、この REST API と Java クライアントとのやり取りのみを紹介します。この API から **curl** からこの実行方法を抽出できます。

```
public interface ComponentsResource {
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<ComponentRepresentation> query();

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<ComponentRepresentation> query(@QueryParam("parent") String parent);

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<ComponentRepresentation> query(@QueryParam("parent") String parent,
@QueryParam("type") String type);

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<ComponentRepresentation> query(@QueryParam("parent") String parent,
                                               @QueryParam("type") String type,
                                               @QueryParam("name") String name);

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    Response add(ComponentRepresentation rep);

    @Path("{id}")
    ComponentResource component(@PathParam("id") String id);
}

public interface ComponentResource {
    @GET
    public ComponentRepresentation toRepresentation();

    @PUT
    @Consumes(MediaType.APPLICATION_JSON)
    public void update(ComponentRepresentation rep);

    @DELETE
    public void remove();
}
```

ユーザーストレージプロバイダーを作成するには、プロバイダー ID、文字列 **org.keycloak.storage.UserStorageProvider** のプロバイダータイプ、および設定を指定する必要があります。

```
import org.keycloak.admin.client.Keycloak;
import org.keycloak.representations.idm.RealmRepresentation;
...

Keycloak keycloak = Keycloak.getInstance(
    "http://localhost:8080",
    "master",
    "admin",
    "password",
    "admin-cli");
RealmResource realmResource = keycloak.realm("master");
RealmRepresentation realm = realmResource.toRepresentation();

ComponentRepresentation component = new ComponentRepresentation();
component.setName("home");
component.setProviderId("readonly-property-file");
component.setProviderType("org.keycloak.storage.UserStorageProvider");
component.setParentId(realm.getId());
component.setConfig(new MultivaluedHashMap());
component.getConfig().putSingle("path", "~/users.properties");

realmResource.components().add(component);

// retrieve a component

List<ComponentRepresentation> components = realmResource.components().query(realm.getId(),
                                                    "org.keycloak.storage.UserStorageProvider",
                                                    "home");
component = components.get(0);

// Update a component

component.getConfig().putSingle("path", "~/my-users.properties");
realmResource.components().component(component.getId()).update(component);

// Remove a component

realmREsource.components().component(component.getId()).remove();
```

6.13. 以前のユーザーフェデレーション SPI からの移行



注記

本章では、以前の (すでに廃止された)User Federation SPI を使用してプロバイダーを実装している場合のみを対象としています。

Keycloak バージョン 2.4.0 以前のバージョンでは、User Federation SPI がありました。Red Hat Single Sign-On バージョン 7.0 はサポート対象外ですが、この以前の SPI も使用できます。この以前の User Federation SPI は、Keycloak バージョン 2.5.0 および Red Hat Single Sign-On バージョン 7.1 から削除されました。ただし、本章では、この SPI でプロバイダーを作成している場合の移植に使用できるストラテジーについて説明します。

6.13.1. インポートと非インポート

以前の User Federation SPI は、Red Hat build of Keycloak のデータベースでユーザーのローカルコピーを作成し、情報を外部ストアからローカルコピーにインポートする必要がありました。ただし、要件ではなくなりました。以前のプロバイダーもそのままの状態で移植できますが、インポートなしのストラテジーの方が適しているかどうかを検討する必要があります。

インポートストラテジーの利点:

- 基本的に、Red Hat build of Keycloak は外部ストアの永続ユーザーキャッシュになります。ユーザーがインポートされると、外部ストアに到達できなくなるため、負荷がなくなります。

- 公式ユーザーストアとして Red Hat build of Keycloak に移行し、以前の外部ストアを非推奨にする場合は、アプリケーションを徐々に移行して Red Hat build of Keycloak を使用できます。すべてのアプリケーションが移行されたら、インポートされたユーザーのリンクを解除し、古いレガシー外部ストアを破棄します。

インポートストラテジーの使用には、いくつかの明確な欠点があります。

- 初めてユーザーを検索するには、Red Hat build of Keycloak データベースを複数回更新する必要があります。これを実行すると、負荷がかかってパフォーマンスが大幅に低下し、Red Hat build of Keycloak データベースに大きな負担がかかる可能性があります。ユーザーフェデレーションされたストレージアプローチは、必要に応じて追加のデータのみを保存し、外部ストアの機能によっては使用されない可能性があります。

- このインポート方法では、ローカルの Red Hat build of Keycloak ストレージと外部ストレージを同期する必要があります。User Storage SPI には同期をサポートするために実装できる機能インターフェイスがありますが、この操作はすぐに面倒で複雑になる可能性があります。

6.13.2. UserFederationProvider と UserStorageProvider

最初に、**UserFederationProvider** が完全なインターフェイスであったことに留意してください。このインターフェイスにすべてのメソッドを実装しました。ただし、**UserStorageProvider** は、このインターフェイスを、必要に応じて実装する複数の機能インターフェイスに分割します。

UserFederationProvider.getUserByUsername() および **getUserByEmail()** には、新しい SPI に完全に同等のものが含まれます。この 2 つの違いは、インポートの方法です。インポートストラテジーを続行する場合は、**KeycloakSession.userStorage().addUser()** を呼び出してユーザーをローカルに作成しなくなりました。代わりに **KeycloakSession.userLocalStorage().addUser()** を呼び出します。**userStorage()** メソッドがなくなりました。

UserFederationProvider.validateAndProxy() メソッドは任意の機能インターフェイス **ImportedUserValidation** に移動しました。以前のプロバイダーをそのまま移植する場合は、このインターフェイスを実装する必要があります。また、以前の SPI では、ローカルユーザーがキャッシュにある場合でも、ユーザーがアクセスされたたびにこのメソッドが呼び出されていました。このメソッドは、後の SPI では、ローカルユーザーがローカルストレージからロードされる場合にのみ呼び出されます。ローカルユーザーがキャッシュされていると、**ImportedUserValidation.validate()** メソッドは呼び出されません。

UserFederationProvider.isValid() メソッドは、後続の SPI に存在しなくなりました。

UserFederationProvider のメソッド **synchronizeRegistrations()**、**registerUser()**、および **removeUser()** が **UserRegistrationProvider** 機能インターフェイスに移動しました。プロバイダーがユーザーの作成や削除をサポートしていない場合は、この新しいインターフェイスを実装する必要はありません。以前のプロバイダーが新規ユーザー登録のサポートを切り替える場合は、これが新しい SPI でサポートされ、プロバイダーがユーザーの追加をサポートしていない場合は **UserRegistrationProvider.addUser()** から **Null** を返します。

認証情報を中心とする以前の **UserFederationProvider** メソッドは、**CredentialInputValidator** イン

ターフェイスおよび **CredentialInputUpdater** インターフェイスにカプセル化されるようになりました。また、認証情報の検証または更新をサポートしているかどうかに応じて実装することもできます。認証情報管理は、以前は **UserModel** メソッドに存在していました。これらは **CredentialInputValidator** インターフェイスおよび **CredentialInputUpdater** インターフェイスに移動されています。**CredentialInputUpdater** インターフェイスを実装していない場合は、Red Hat build of Keycloak ストレージでプロバイダーが提供する認証情報をローカルで上書きできます。そのため、認証情報を読み取り専用にするには、**CredentialInputUpdater.updateCredential()** メソッドを実装し、**ReadOnlyException** を返します。

searchByAttributes()、**getGroupMembers()** などの **UserFederationProvider** クエリーメソッドは、任意のインターフェイス **UserQueryProvider** でカプセル化されるようになりました。このインターフェイスを実装しないと、ユーザーは管理コンソールでは表示されません。ただし、引き続きログインはできます。

6.13.3. UserFederationProviderFactory と UserStorageProviderFactory

以前の SPI の同期メソッドは、任意の **ImportSynchronization** インターフェイス内でカプセル化されるようになりました。同期ロジックを実装している場合、新しい **UserStorageProviderFactory** が **ImportSynchronization** インターフェイスを実装します。

6.13.4. 新規モデルへのアップグレード

User Storage SPI インスタンスは異なるリレーショナルテーブルのセットに保存されます。Red Hat build of Keycloak は、移行スクリプトを自動的に実行します。以前のユーザーフェデレーションプロバイダーがレルムにデプロイされると、データの **id** を含む、後のストレージモデルにそのまま変換されます。この移行は、以前のユーザーフェデレーションプロバイダーのプロバイダー ID ("ldap", "kerberos") と同じユーザーストレージプロバイダーが存在する場合のみ行われます。

そのため、この知識をもとに、さまざまなアプローチを取ることができます。

1. 以前の Red Hat build of Keycloak デプロイメントでは、以前のプロバイダーを削除できます。これにより、インポートした全ユーザーのローカルリンクコピーが削除されます。その後、Red Hat build of Keycloak をアップグレードする際に、新しいプロバイダーをレルムにデプロイして設定します。

2. 2 つ目のオプションは、**UserStorageProviderFactory.getId()** という同じプロバイダー ID を持つように新規プロバイダーを作成します。このプロバイダーがサーバーにデプロイされていることを確認します。サーバーを起動し、組み込み移行スクリプトが以前のデータモデルから後のデータモデルに変換されるようにします。この場合、以前にリンクしたインポート済みのユーザーはすべて機能し、同じ状態になります。

インポートストラテジーを廃止して、ユーザーストレージプロバイダーを書き換えた場合は、Red Hat build of Keycloak をアップグレードする前に以前のプロバイダーを削除することを推奨します。これにより、インポートしたユーザーにリンクされたローカルのインポートコピーが削除されます。

6.14. ストリームベースのインターフェイス

Red Hat build of Keycloak のユーザーストレージインターフェイスの多くは、潜在的に大きなオブジェクトのセットを返すことができるクエリーメソッドを含んでおり、メモリー消費や処理時間の面で大きな影響を与える可能性があります。これは、クエリーメソッドのロジックでオブジェクトの内部状態の小さなサブセットのみが使用される場合に特に当てはまります。

これらのクエリーメソッドで大規模なデータセットを処理するためのより効率的な代替手段を開発者に提供するために、ユーザーストレージインターフェイスに **Streams** サブインターフェイスが追加されました。これらの **Streams** サブインターフェイスは、スーパーインターフェイスのオリジナルのコレ

クションベースのメソッドをストリームベースのバリアントに置き換え、コレクションベースのメソッドをデフォルトにしています。コレクションベースのクエリーメソッドのデフォルトの実装では、**Stream** 対応のメソッドが呼び出され、結果が適切なコレクションタイプに収集されます。

Streams サブインターフェイスは、データセットを処理するためのストリームベースのアプローチに焦点を当てた実装を可能にし、そのアプローチによる潜在的なメモリーとパフォーマンスの最適化の恩恵を受けることができます。実装される **Streams** サブインターフェイスを提供するインターフェイスには、いくつかの **機能インターフェイス**、**org.keycloak.storage.federated** パッケージ内のすべてのインターフェイス、およびカスタムストレージ実装の範囲に応じて実装される可能性があるいくつかのインターフェイスが含まれます。

開発者向けに **Streams** サブインターフェイスを提供しているインターフェイスのリストを参照してください。

パッケージ	クラス
org.keycloak.credential	**CredentialInputUpdater**(*)
org.keycloak.models	**GroupModel**、**RoleMapperModel**、**UserModel**
org.keycloak.storage.federated	すべてのインターフェイス
org.keycloak.storage.user	**UserQueryProvider**(*)

(*) はインターフェイスが **機能インターフェイス** であることを示しています。

ストリームアプローチの恩恵を受けたいユーザーストレージのカスタム実装は、オリジナルのインターフェイスの代わりに **Streams** サブインターフェイスを実装する必要があります。たとえば、次のコードでは、**UserQueryProvider** インターフェイスの **Streams** バリアントを使用しています。

```
public class CustomQueryProvider extends UserQueryProvider.Streams {
...
    @Override
    Stream<UserModel> getUsersStream(RealmModel realm, Integer firstResult, Integer maxResults)
{
        // custom logic here
    }

    @Override
    Stream<UserModel> searchForUserStream(String search, RealmModel realm) {
        // custom logic here
    }
...
}
```

第7章 VAULT SPI

7.1. VAULT プロバイダー

org.keycloak.vault パッケージの Vault SPI を使用して、Red Hat build of Keycloak のカスタム拡張を作成し、任意の Vault 実装に接続できます。

組み込みの **files-plaintext** プロバイダーは、この SPI の実装例です。一般的に以下のルールが適用されます。

- レルム間でシークレットのリークを防ぐために、レルムで取得できるシークレットを分離したり、制限したりすることができます。この場合、プロバイダーはシークレットの検索時に、エントリーの前にレルム名を付けるなど、レルム名を考慮する必要があります。たとえば、式 **${vault.key}** は、レルム **A** またはレルム **B** で使用されるかに応じて、一般的に異なるエントリー名に評価されます。レルムを区別するには、レルムを、**KeycloakSession** パラメーターから利用できる **VaultProviderFactory.create()** メソッドから、作成された **VaultProvider** インスタンスに渡す必要があります。

- Vault プロバイダーは、指定したシークレット名に **VaultRawSecret** 返す単一のメソッド **obtainSecret** を実装します。そのクラスは、シークレットの表現を **byte[]** または **ByteBuffer** のいずれかに保持し、必要に応じて 2 つの間で変換することが期待されます。このバッファーは、以下で説明されているように使用後に破棄されることに注意してください。

カスタムプロバイダーをパッケージ化してデプロイする方法は、サービスプロバイダーインターフェイス の章を参照してください。

7.2. VAULT からの値の使用

vault には機密データが含まれ、Red Hat build of Keycloak がシークレットを適宜処理します。シークレットにアクセスする場合には、シークレットは vault から取得され、必要な期間のみ JVM メモリーに保持されます。その後、JVM メモリーからコンテンツの破棄を試みることができます。これは、以下のように **try**-with-resources ステートメント内でのみ Vault シークレットを使用して実行されます。

```
char[] c;
try (VaultCharSecret cSecret = session.vault().getCharSecret(SECRET_NAME)) {
    // ... use cSecret
    c = cSecret.getAsArray().orElse(null);
    // if c != null, it now contains password
}

// if c != null, it now contains garbage
```

この例では、シークレットにアクセスするためのエントリーポイントとして **KeycloakSession.vault()** を使用します。**VaultProvider.obtainSecret** メソッドを直接使用することも可能になります。ただし、**vault()** メソッドは、オリジナルの解釈されていない値 (**vault().getRawSecret()** メソッド経由) を取得するのに加え、(通常はバイト配列である) 未編集のシークレットを文字アレイ (**vault().getCharSecret()** 経由) または **String** (via **vault().getStringSecret()** 経由) として解釈できるという利点があります。

String オブジェクトは変更できないため、ランダムなガベージでコンテンツを破棄できないことに注意してください。デフォルトの **VaultStringSecret** の実装では、**String** の内面化を防ぐための対策が取られていますが、**String** オブジェクトに格納されているシークレットは、少なくとも次の GC ラウンドまで存続します。そのため、プレーンなバイト配列、文字配列、およびバッファーを使用することが推奨されます。