



Red Hat Single Sign-On 7.4

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

Red Hat Single Sign-On 7.4 向け

Red Hat Single Sign-On 7.4 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

Red Hat Single Sign-On 7.4 向け

Enter your first name here. Enter your surname here.

Enter your organisation's name here. Enter your organisational division here.

Enter your email address here.

法律上の通知

Copyright © 2022 | You need to change the HOLDER entity in the en-US/Securing_Applications_and_Services_Guide.ent file |.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

概要

本ガイドは、Red Hat Single Sign-On 7.4 を使用してアプリケーションとサービスのセキュリティを保護する情報で構成されています。

目次

多様性を受け入れるオープンソースの強化	7
第1章 概要	8
1.1. クライアントアダプターとは	8
1.2. サポート対象のプラットフォーム	8
1.2.1. OpenID Connect	8
1.2.1.1. Java	8
1.2.1.2. JavaScript (クライアント側)	8
1.2.1.3. Node.js (サーバー側)	8
1.2.2. SAML	8
1.2.2.1. Java	8
1.2.2.2. Apache HTTP Server	8
1.3. サポートされるプロトコル	9
1.3.1. OpenID Connect	9
1.3.2. SAML 2.0	9
1.3.3. OpenID Connect 対SAML	9
第2章 OPENID CONNECT	11
2.1. JAVA アダプター	11
2.1.1. Java アダプターの設定	11
2.1.2. JBoss EAP Adapter	15
2.1.2.1. アダプターのインストール	15
2.1.2.2. JBoss SSO	17
2.1.2.3. 各 WAR に必要な設定	17
2.1.2.4. Adapter サブシステムによる WAR のセキュリティ保護	18
2.1.2.5. セキュリティドメイン	19
2.1.3. RPM からの JBoss EAP アダプターのインストール	19
2.1.4. JBoss Fuse 6 Adapter	21
2.1.4.1. Fuse 6 での Web アプリケーションのセキュリティ保護	21
2.1.4.2. Keycloak 機能のインストール	22
2.1.4.2.1. Maven リポジトリからのインストール	22
2.1.4.2.2. ZIP バンドルからのインストール	22
2.1.4.3. クラシック WAR アプリケーションのセキュリティ保護	23
2.1.4.3.1. 外部アダプターの設定	24
2.1.4.4. OSGI サービスとしてデプロイされたサーブレットのセキュア化	25
2.1.4.5. Apache Camel アプリケーションのセキュリティ保護	26
2.1.4.6. Camel RestDSL	28
2.1.4.7. 別の Jetty Engine での Apache CXF エンドポイントのセキュリティ保護	29
2.1.4.8. デフォルトの Jetty Engine での Apache CXF エンドポイントのセキュリティ保護	31
2.1.4.9. Fuse 管理サービスのセキュリティ保護	33
2.1.4.9.1. Fuse ターミナルへの SSH 認証の使用	33
2.1.4.9.2. JMX 認証の使用	34
2.1.4.10. Hawtio 管理コンソールのセキュリティ保護	34
2.1.4.10.1. JBoss EAP 6.4 での Hawtio のセキュリティ保護	35
2.1.5. JBoss Fuse 7 Adapter	37
2.1.5.1. Fuse 7 での Web アプリケーションのセキュリティ保護	37
2.1.5.2. Keycloak 機能のインストール	38
2.1.5.2.1. Maven リポジトリからのインストール	38
2.1.5.2.2. ZIP バンドルからのインストール	38
2.1.5.3. クラシック WAR アプリケーションのセキュリティ保護	39
2.1.5.3.1. 設定リゾルバー	40

2.1.5.4. OSGI サービスとしてデプロイされたサーブレットのセキュア化	41
2.1.5.5. Apache Camel アプリケーションのセキュリティ保護	42
2.1.5.6. Camel RestDSL	43
2.1.5.7. 別の Undertow エンジンでの Apache CXF エンドポイントのセキュリティ保護	44
2.1.5.8. デフォルトの Undertow エンジンでの Apache CXF エンドポイントのセキュリティ保護	45
2.1.5.9. Fuse 管理サービスのセキュリティ保護	47
2.1.5.9.1. Fuse ターミナルへの SSH 認証の使用	47
2.1.5.9.2. JMX 認証の使用	48
2.1.5.10. Hawtio 管理コンソールのセキュリティ保護	48
2.1.6. Spring Boot アダプター	50
2.1.6.1. アダプターのインストール	50
2.1.6.2. 必要な Spring Boot アダプター設定	51
2.1.7. Java Servlet Filter Adapter	51
2.1.8. セキュリティコンテキスト	53
2.1.9. エラー処理	53
2.1.10. ログアウト	54
2.1.11. パラメーター転送	54
2.1.12. クライアント認証	55
2.1.12.1. クライアント ID およびクライアントシークレット	55
2.1.12.2. 署名済み JWT によるクライアント認証	55
2.1.13. マルチテナンシー	56
2.1.14. アプリケーションクラスタリング	57
2.1.14.1. ステートレストークンストア	58
2.1.14.2. 相対 URI の最適化	58
2.1.14.3. 管理 URL の設定	59
2.1.14.4. アプリケーションノードの登録	59
2.1.14.5. 各リクエストでトークンを更新する	60
2.2. JAVASCRIPT アダプター	60
2.2.1. セッションステータスの iframe	63
2.2.2. 暗黙的フローおよびハイブリッドフロー	64
2.2.3. Cordova でのハイブリッドアプリケーション	64
2.2.4. 以前のブラウザー	66
2.2.5. JavaScript アダプターリファレンス	66
2.2.5.1. コンストラクター	66
2.2.5.2. プロパティー	66
2.2.5.3. メソッド	67
2.2.5.3.1. init(options)	67
2.2.5.3.2. login(options)	68
2.2.5.3.3. createLoginUrl(options)	69
2.2.5.3.4. logout(options)	69
2.2.5.3.5. createLogoutUrl(options)	69
2.2.5.3.6. register(options)	69
2.2.5.3.7. createRegisterUrl(options)	69
2.2.5.3.8. accountManagement()	70
2.2.5.3.9. createAccountUrl()	70
2.2.5.3.10. hasRealmRole(role)	70
2.2.5.3.11. hasResourceRole(role, resource)	70
2.2.5.3.12. loadUserProfile()	70
2.2.5.3.13. isTokenExpired(minValidity)	70
2.2.5.3.14. updateToken(minValidity)	70
2.2.5.3.15. clearToken()	71
2.2.5.4. コールバックイベント	71
2.3. NODE.JS アダプター	71

2.3.1. インストール	71
2.3.2. 使用法	72
2.3.3. ミドルウェアのインストール	73
2.3.4. 認証の確認	73
2.3.5. リソースの保護	73
2.3.6. 追加の URL	75
2.4. その他の OPENID CONNECT ライブラリー	75
2.4.1. エンドポイント	76
2.4.1.1. 承認エンドポイント	76
2.4.1.2. トークンエンドポイント	76
2.4.1.3. userInfo エンドポイント	76
2.4.1.4. ログアウトエンドポイント	76
2.4.1.5. 証明書エンドポイント	77
2.4.1.6. イン트로スペクションエンドポイント	77
2.4.1.7. 動的クライアント登録エンドポイント	77
2.4.1.8. トークン失効エンドポイント	77
2.4.2. アクセストークンの検証	77
2.4.3. フロー	78
2.4.3.1. 承認コード	78
2.4.3.2. 暗黙的	78
2.4.3.3. リソースオーナーパスワード認証情報	78
2.4.3.3.1. CURL の使用例	79
2.4.3.4. クライアント認証情報	79
2.4.4. リダイレクト URI	79
第3章 SAML	81
3.1. JAVA アダプター	81
3.1.1. 汎用アダプターの設定	81
3.1.1.1. SP 要素	82
3.1.1.2. サービスプロバイダーキーのキー要素	83
3.1.1.2.1. キーストア要素	83
3.1.1.2.2. キーの PEMS	84
3.1.1.3. SP PrincipalNameMapping 要素	84
3.1.1.4. RoleIdentifiers 要素	85
3.1.1.5. RoleMappingsProvider 要素	85
3.1.1.5.1. プロパティーベースのロールマッピングプロバイダー	86
3.1.1.5.2. 独自のロールマッピングプロバイダーの追加	87
3.1.1.6. IDP 要素	87
3.1.1.7. IDP AllowedClockSkew サブ要素	87
3.1.1.8. IDP SingleSignOnService サブ要素	88
3.1.1.9. IDP SingleLogoutService サブ要素	88
3.1.1.10. IDP キーサブ要素	89
3.1.1.11. IDP HttpClient サブ要素	90
3.1.2. JBoss EAP Adapter	91
3.1.2.1. アダプターのインストール	91
3.1.2.2. JBoss SSO	92
3.1.3. RPM からの JBoss EAP アダプターのインストール	92
3.1.3.1. WAR ごとの設定	94
3.1.3.2. Red Hat Single Sign-On SAML サブシステムを使用した WAR のセキュリティ保護	95
3.1.4. Java Servlet Filter Adapter	97
3.1.5. ID プロバイダーを使用した登録	98
3.1.6. ログアウト	98
3.1.6.1. クラスター化された環境でログアウト	99

3.1.6.2. クロス DC シナリオにおけるログアウト	99
3.1.7. assertion 属性の取得	100
3.1.8. エラー処理	102
3.1.9. トラブルシューティング	102
3.1.10. マルチテナンシー	102
3.2. APACHE HTTPD モジュール MOD_AUTH_MELLON	104
3.2.1. Red Hat Single Sign-On での mod_auth_mellon の設定	104
3.2.1.1. パッケージのインストール	104
3.2.1.2. Apache SAML の設定ディレクトリーの作成	105
3.2.1.3. Mellon サービスプロバイダーの設定	105
3.2.1.4. サービスプロバイダーメタデータの作成	106
3.2.1.5. Red Hat Single Sign-On ID プロバイダーへの Mellon サービスプロバイダーの追加	107
3.2.1.5.1. レルムのクライアントとしての Mellon Service Provider の追加	107
3.2.1.5.2. Mellon SP クライアントの追加	107
3.2.1.5.3. Mellon SP クライアントの編集	108
3.2.1.5.4. アイデンティティプロバイダーメタデータの取得	108
第4章 DOCKER レジストリーの設定	110
4.1. DOCKER レジストリー設定ファイルのインストール	110
4.2. DOCKER レジストリー環境変数オーバーライドインストール	110
4.3. DOCKER COMPOSE YAML ファイル	111
第5章 クライアント登録	112
5.1. 認証	112
5.1.1. ベアラートークン	112
5.1.2. 初期アクセストークン	112
5.1.3. 登録アクセストークン	113
5.2. RED HAT SINGLE SIGN-ON REPRESENTATIONS	113
5.3. RED HAT SINGLE SIGN-ON アダプターの設定	113
5.4. OPENID CONNECT 動的クライアント登録	114
5.5. SAML エンティティ記述子	114
5.6. CURL の使用例	114
5.7. JAVA クライアント登録 API の使用例	114
5.8. クライアント登録ポリシー	115
第6章 クライアント登録 CLI	117
6.1. クライアント登録 CLI で使用する新しい一般ユーザーの設定	117
6.2. クライアント登録 CLI で使用するクライアントの設定	117
6.3. クライアント登録 CLI のインストール	118
6.4. クライアント登録 CLI の使用	119
6.4.1. ログイン	119
6.4.2. 代替設定の使用	120
6.4.3. 初期アクセスおよび登録アクセストークン	120
6.4.4. クライアント設定の作成	121
6.4.5. クライアント設定の取得	122
6.4.6. クライアント設定の変更	122
6.4.7. クライアント設定の削除	123
6.4.8. 無効な登録アクセストークンの更新	123
6.5. トラブルシューティング	124
第7章 トークンの交換	125
7.1. 内部トークンから内部トークンへの交換	127
7.1.1. Exchange のパーミッションの付与	127
7.1.2. リクエストの作成	130

7.2. 内部トークンから外部トークンへの交換	130
7.2.1. Exchange のパーミッションの付与	131
7.2.2. リクエストの作成	133
7.3. 外部トークンから内部トークンへの交換	134
7.3.1. Exchange のパーミッションの付与	134
7.3.2. リクエストの作成	134
7.4. 権限借用	135
7.4.1. Exchange のパーミッションの付与	135
7.4.2. リクエストの作成	135
7.5. ダイレクトの NAKED IMPERSONATION	136
7.5.1. Exchange のパーミッションの付与	136
7.5.2. リクエストの作成	139
7.6. サービスアカウントでアクセス許可モデルを展開	139
7.7. 交換の脆弱性	139

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、[弊社 の CTO、Chris Wright のメッセージ](#)を参照してください。

第1章 概要

Red Hat Single Sign-On は、OpenID Connect (OAuth 2.0 への拡張) と SAML 2.0 の両方をサポートします。クライアントおよびサービスのセキュリティを保護する場合は、最初に使用するものを決定する必要があります。OpenID Connect や他の SAML でセキュリティ保護するオプションを選択することもできます。

クライアントやサービスのセキュリティを保護するには、選択したプロトコルにアダプターまたはライブラリーが必要になります。Red Hat Single Sign-On には、選択したプラットフォーム用の独自のアダプターが含まれていますが、汎用 OpenID Connect Relying Party ライブラリーおよび SAML Service Provider ライブラリーを使用することもできます。

1.1. クライアントアダプターとは

Red Hat Single Sign-On クライアントアダプターは、Red Hat Single Sign-On でアプリケーションおよびサービスのセキュリティを簡単に保護するライブラリーです。基礎となるプラットフォームやフレームワークへの密接な統合を提供するため、ライブラリーではなくアダプターを呼び出します。これにより、アダプターは使いやすく、ライブラリーで通常必要とされるものよりも必要なボーラプレートコードよりも少なくなります。

1.2. サポート対象のプラットフォーム

1.2.1. OpenID Connect

1.2.1.1. Java

- [JBoss EAP](#)
- [Fuse](#)
- [Servlet Filter](#)
- [Spring Boot](#)

1.2.1.2. JavaScript (クライアント側)

- [JavaScript](#)

1.2.1.3. Node.js (サーバー側)

- [Node.js](#)

1.2.2. SAML

1.2.2.1. Java

- [JBoss EAP](#)

1.2.2.2. Apache HTTP Server

- [mod_auth_mellon](#)

1.3. サポートされるプロトコル

1.3.1. OpenID Connect

[OpenID Connect](#) (OIDC) は、[OAuth 2.0](#) の拡張機能である認証プロトコルです。OAuth 2.0 は承認プロトコルを構築するためのフレームワークでしかなく、主に不完全ですが、OIDC は完全な認証および承認プロトコルです。OIDC は、JWT ([Json Web Token](#)) 標準セットも多用しています。これらの標準は、アイデンティティプロバイダーの JSON 形式を定義し、そのデータをコンパクトで Web フレンドリーで暗号化する方法を定義しています。

OIDC を使用する場合は、本当に 2 つのタイプのユースケースがあります。1 つ目のアプリケーションは、Red Hat Single Sign-On サーバーがユーザーを認証するよう依頼するアプリケーションです。ログインに成功すると、アプリケーションは ID トークンと アクセストークンを受け取ります。ID トークンには、ユーザー名、電子メール、その他のプロフィール情報など、ユーザーに関する情報が含まれています。アクセストークンはレلمムによってデジタル署名され、アプリケーションでユーザーがアクセスできるリソースを判別するためにアプリケーションが使用できるアクセス情報 (ユーザーロールマッピングなど) が含まれます。

2 つ目のタイプのユースケースは、リモートサービスへのアクセスを取得するクライアントのもので、この場合、クライアントはユーザーの代わりに他のリモートサービスで起動するのに使用できる アクセストークンを取得するよう Red Hat Single Sign-On に要求します。Red Hat Single Sign-On はユーザーを認証し、ユーザーに要求したクライアントにアクセスを付与するよう依頼します。その後、クライアントはアクセストークンを受け取ります。このアクセストークンはレلمムによってデジタル署名されます。クライアントはこのアクセストークンを使用してリモートサービスで REST 呼び出しを行うことができます。REST サービスは、アクセストークンを抽出し、トークンの署名を検証した後、トークン内のアクセス情報に基づいて、リクエストを処理するかどうかを決定します。

1.3.2. SAML 2.0

[SAML 2.0](#) は OIDC と同様の仕様ですが、より古く、より成熟したものです。SOAP と [plethora](#) の WS-* 仕様にはルートがあるため、OIDC よりも詳細度が少くなる傾向があります。SAML 2.0 は主に、認証サーバーとアプリケーション間の XML ドキュメント変更によって機能する認証プロトコルです。XML 署名と暗号化は、要求と応答の検証に使用されます。

Red Hat Single Sign-On SAML では、ブラウザーアプリケーションと REST 呼び出しという 2 種類のユースケースを利用できます。

SAML を使用する場合は、主に 2 つのタイプのユースケースがあります。1 つ目のアプリケーションは、Red Hat Single Sign-On サーバーがユーザーを認証するよう依頼するアプリケーションです。ログインが成功すると、アプリケーションには、ユーザーのさまざまな属性を指定する SAML アサーションが含まれる XML ドキュメントを受け取ります。XML ドキュメントはレلمムによってデジタル署名され、アプリケーションでユーザーがアクセスできるリソースを判別するためにアプリケーションが使用できるアクセス情報 (ユーザーロールマッピングなど) が含まれます。

2 つ目のタイプのユースケースは、リモートサービスへのアクセスを取得するクライアントのもので、この場合、クライアントはユーザーの代わりに他のリモートサービスで起動するのに使用できる SAML アサーションを取得するよう Red Hat Single Sign-On に要求します。

1.3.3. OpenID Connect 対 SAML

OpenID Connect と SAML の選択は、古い成熟したプロトコル (SAML) の代わりに新しいプロトコル (OIDC) を使用する場合でも問題はありません。

多くの場合、Red Hat Single Sign-On では、OIDC の使用を推奨します。

SAML は OIDC よりも詳細な状態である傾向があります。

交換されたデータの詳細レベルを超えた場合、SAML は Web で機能するように OIDC が Web で機能するように設計されている仕様を比較すると、Web 上で動作するように調整されました。たとえば、OIDC は、SAML よりもクライアント側に簡単に実装できるため、HTML5/JavaScript アプリケーションにも適しています。トークンは JSON 形式であるため、JavaScript による使いやすさが簡単です。また、Web アプリケーションでセキュリティを実装するためにより優れた機能もいくつかあります。たとえば、ユーザーがログインしたままかどうかを簡単に判断するために仕様で使用されている [iframe における効果的な方法](#) を確認してください。

SAML がその使い方もあります。OIDC 仕様が進化したように、SAML で数年以上の機能を実装することがわかります。多くの場合は、成熟度が高いため、またそれらにはセキュリティを確保した既存のアプリケーションがすでにあるため、SAML over OIDC を選択していることがよく見られます。

第2章 OPENID CONNECT

本セクションでは、Red Hat Single Sign-On アダプターまたは汎用 OpenID Connect Relying Party ライブラリーを使用して、OpenID Connect でアプリケーションとサービスのセキュリティを保護する方法を説明します。

2.1. JAVA アダプター

Red Hat Single Sign-On には、Java アプリケーションのさまざまなアダプターがあります。正しいアダプターの選択は、ターゲットプラットフォームによって異なります。

すべての Java アダプターは、「[Java アダプター設定](#)」の章で説明されている共通の設定オプションセットを共有します。

2.1.1. Java アダプターの設定

Red Hat Single Sign-On がサポートする各 Java アダプターは、単純な JSON ファイルで設定できます。これは次のようになります。

```
{
  "realm" : "demo",
  "resource" : "customer-portal",
  "realm-public-key" : "MIGfMA0GCSqGSIb3D...31LwIDAQAB",
  "auth-server-url" : "https://localhost:8443/auth",
  "ssl-required" : "external",
  "use-resource-role-mappings" : false,
  "enable-cors" : true,
  "cors-max-age" : 1000,
  "cors-allowed-methods" : "POST, PUT, DELETE, GET",
  "cors-exposed-headers" : "WWW-Authenticate, My-custom-exposed-Header",
  "bearer-only" : false,
  "enable-basic-auth" : false,
  "expose-token" : true,
  "verify-token-audience" : true,
  "credentials" : {
    "secret" : "234234-234234-234234"
  },

  "connection-pool-size" : 20,
  "disable-trust-manager" : false,
  "allow-any-hostname" : false,
  "truststore" : "path/to/truststore.jks",
  "truststore-password" : "geheim",
  "client-keystore" : "path/to/client-keystore.jks",
  "client-keystore-password" : "geheim",
  "client-key-password" : "geheim",
  "token-minimum-time-to-live" : 10,
  "min-time-between-jwks-requests" : 10,
  "public-key-cache-ttl" : 86400,
  "redirect-rewrite-rules" : {
    "^/wsmaster/api/(.*)$" : "/api/$1"
  }
}
```

システムプロパティの置き換えには `${...}` エンクロージャーを使用できます。たとえば、`${jboss.server.config.dir}` は `/path/to/Red Hat Single Sign-On` に置き換えられます。環境変数の代替も、`env` 接頭辞 (例: `${env.MY_ENVIRONMENT_VARIABLE}`) でサポートされます。

初期設定ファイルは管理コンソールから取得できます。これは、管理コンソールを開き、メニューから **Clients** を選択して、対応するクライアントをクリックします。クライアントのページが開いたら、インストール タブをクリックし、**Keycloak OIDC JSON** を選択します。

各設定オプションの説明は次のとおりです。

realm

レルムの名前これは 必須 です。

resource

アプリケーションの client-id 各アプリケーションには、アプリケーションを識別するのに使用される client-id があります。これは 必須 です。

realm-public-key

レルム公開鍵の PEM 形式。これは管理コンソールから取得できます。これは 任意 ですが、設定することは推奨されません。設定しないと、アダプターは Red Hat Single Sign-On からダウンロードし、必要に応じて常に再ダウンロードを行います (例: Red Hat Single Sign-On の鍵をローテーション)。ただし、realm-public-key が設定されている場合、アダプターは Red Hat Single Sign-On から新しい鍵をダウンロードしません。そのため、Red Hat Single Sign-On が鍵をローテーションするとアダプターが中断されます。

auth-server-url

Red Hat Single Sign-On サーバーのベース URL。他のすべての Red Hat シングルサインオンページおよび REST サービスエンドポイントはこれから派生します。通常、形式は `https://host:port/auth` です。これは 必須 です。

ssl-required

Red Hat Single Sign-On サーバーとの間のすべての通信が HTTPS を介して行われるようにします。実稼働環境では、これを **all** に設定する必要があります。これは 任意 です。デフォルト値は **external** です。つまり、外部要求に HTTPS がデフォルトで必要になります。有効な値は、「all」、「external」、および「none」です。

confidential-port

SSL/TLS を介してセキュアな接続のために、Red Hat Single Sign-On サーバーによって使用される機密ポート。これは 任意 です。デフォルト値は **8443** です。

use-resource-role-mappings

true に設定すると、アダプターはユーザーのアプリケーションレベルのロールマッピングのトークンを検索します。false の場合は、ユーザーロールマッピングのレルムレベルを確認します。これは 任意 です。デフォルト値は **false** です。

public-client

true に設定すると、アダプターはクライアントの認証情報を Red Hat Single Sign-On に送信しません。これは 任意 です。デフォルト値は **false** です。

enable-cors

これにより、CORS サポートが有効になります。CORS プリフライトリクエストを処理します。また、アクセストークンを調べて、有効な発信元を判別します。これは 任意 です。デフォルト値は **false** です。

cors-max-age

CORS が有効な場合は、**Access-Control-Max-Age** ヘッダーの値が設定されます。これは 任意 です。設定されていない場合、このヘッダーは CORS 応答で返されません。

cors-allowed-methods

CORS が有効な場合は、**Access-Control-Allow-Methods** ヘッダーの値が設定されます。これはコンマ区切りの文字列である必要があります。これは 任意 です。設定されていない場合、このヘッダーは CORS 応答で返されません。

cors-allowed-headers

CORS が有効な場合は、**Access-Control-Allow-Headers** ヘッダーの値を設定します。これはコンマ区切りの文字列である必要があります。これは 任意 です。設定されていない場合、このヘッダーは CORS 応答で返されません。

cors-exposed-headers

CORS が有効な場合は、**Access-Control-Expose-Headers** ヘッダーの値を設定します。これはコンマ区切りの文字列である必要があります。これは 任意 です。設定されていない場合、このヘッダーは CORS 応答で返されません。

bearer-only

これは、サービスに **true** に設定する必要があります。有効な場合、アダプターはユーザーの認証を試行しませんが、ベアラートークンのみを検証します。これは 任意 です。デフォルト値は **false** です。

autodetect-bearer-only

アプリケーションが Web アプリケーションと Web サービス (例: SOAP または REST) の両方に対応する場合は、**true** に設定する必要があります。Web アプリケーションの認証されていないユーザーを Keycloak ログインページにリダイレクトできますが、認証されていない SOAP または REST クライアントに HTTP **401** ステータスコードを送信することができます。ログインページへのリダイレクトは理解できません。Keycloak は、**X-Requested-With**、**SOAPAction**、**Accept** などの一般的なヘッダーに基づいて SOAP クライアントまたは REST クライアントを自動検出します。デフォルト値は **false** です。

enable-basic-auth

これは、Basic 認証もサポートするようにアダプターに指示します。このオプションが有効な場合には、シークレット も指定する必要があります。これは 任意 です。デフォルト値は **false** です。

expose-token

true の場合、(JavaScript HTTP 呼び出しを介して) 認証されたブラウザークライアントは、URL **root/k_query_bearer_token** を使用して署名されたアクセストークンを取得できます。これは 任意 です。デフォルト値は **false** です。

credentials

アプリケーションの認証情報を指定します。これは、鍵が認証情報タイプであり、値が認証情報タイプの値であるオブジェクト表記です。現在、パスワードおよび jwt がサポートされています。これは、アクセスタイプが「Confidential」を持つクライアントにのみ 必須 になります。

connection-pool-size

この設定オプションでは、Red Hat Single Sign-On サーバーへプールする接続の数を定義します。これは 任意 です。デフォルト値は **20** です。

disable-trust-manager

Red Hat Single Sign-On サーバーに HTTPS が必要で、この設定オプションが **true** に設定されている場合は、トラストストアを指定する必要はありません。この設定は開発時のみ使用してください。これは SSL 証明書の検証を無効にするため、実稼働環境では 使用しないでください。これは 任意 です。デフォルト値は **false** です。

allow-any-hostname

Red Hat Single Sign-On サーバーに HTTPS が必要で、この設定オプションが **true** に設定されている場合、Red Hat Single Sign-On サーバーの証明書はトラストストア経由で検証されますが、ホスト名の検証は行われません。この設定は開発時のみ使用してください。これは SSL 証明書の検証を無効にするため、実稼働環境では 使用しないでください。この設定は、テスト環境で役に立つ場合があります。これは 任意 です。デフォルト値は **false** です。

proxy-url

HTTP プロキシが使用されている場合はその URL。

truststore

値は、トラストストアファイルへのファイルパスです。**classpath:** でパスを接頭辞にすると、代わりにデプロイメントのクラスパスからトラストストアが取得されます。Red Hat Single Sign-On サーバーへの送信 HTTPS 通信に使用されます。HTTPS 要求を実行するクライアントでは、通信先のサーバーのホストを確認する方法が必要です。これは、トラストストアが行なうことです。キーストアには、1つ以上の信頼できるホスト証明書または認証局が含まれます。Red Hat Single Sign-On サーバーの SSL キーストアの公開証明書を抽出して、このトラストストアを作成できます。これは、**ssl-required** が **none**、または **disable-trust-manager** が **true** でない限り、必須 になります。

truststore-password

トラストストアのパスワード。これは、トラストストア が設定され、トラストストアにパスワードが必要な場合は 必須 になります。

client-keystore

これはキーストアファイルに対するファイルパスです。このキーストアには、アダプターが Red Hat Single Sign-On サーバーに対して HTTPS を要求する際に双方向 SSL のクライアント証明書が含まれます。これは 任意 です。

client-keystore-password

クライアントキーストアのパスワード。これは、**client-keystore** が設定されている場合は 必須 になります。

client-key-password

クライアントのキーのパスワードこれは、**client-keystore** が設定されている場合は 必須 になります。

always-refresh-token

true の場合、アダプターはすべてのリクエストでトークンを更新します。警告: これを有効にすると、アプリケーションへのリクエストごとに Red Hat Single Sign-On へのリクエストが行われます。

register-node-at-startup

true の場合、アダプターは登録リクエストを Red Hat Single Sign-On に送信します。デフォルトは **false** で、アプリケーションがクラスター化されている場合にのみ便利です。詳細は、[「アプリケーションクラスタリング」](#) を参照してください。

register-node-period

Red Hat Single Sign-On に再登録アダプターを使用する期間。アプリケーションがクラスター化されている場合に役立ちます。詳細は、[「アプリケーションクラスタリング」](#) を参照してください。

token-store

使用できる値は **session** および **cookie** です。デフォルトは **session** で、アダプターはアカウント情報を HTTP セッションに保存します。代替 **cookie** とは、cookie への情報の格納を意味します。詳細は、[「アプリケーションクラスタリング」](#) を参照してください。

token-cookie-path

cookie ストアを使用する場合、このオプションはアカウント情報を保存するために使用される cookie のパスを設定します。相対パスの場合は、アプリケーションがコンテキストルートで実行されていることを前提とし、そのコンテキストルートへの相対的として解釈されます。絶対パスの場合は、クッキーパスの設定に絶対パスが使用されます。デフォルトは、コンテキストルートへの相対パスを使用します。

principal-attribute

UserPrincipal 名を入力する OpenID Connect ID Token 属性。トークン属性が null の場合、デフォルトは **sub** に設定されます。使用できる値は

sub、**preferred_username**、**email**、**name**、**nickname**、**given_name**、**family_name** です。

turn-off-change-session-id-on-login

一部のプラットフォームでログインに成功すると、セキュリティ攻撃ベクトルをプラグインするために、セッション ID がデフォルトで変更されます。これをオフにする場合は、これを true に変更します。これは 任意 です。デフォルト値は false です。

token-minimum-time-to-live

Red Hat Single Sign-On サーバーでアクティブなアクセストークンを事前更新してから失効する時間 (秒単位)。これは、アクセストークンが別の REST クライアントに送信され、評価前に期限切れになる可能性がある場合に特に便利です。この値は、レルムのアクセストークンの有効期間を超えてはいけなはずです。これは 任意 です。デフォルト値は 0 秒であるため、アダプターは期限切れの場合にのみアクセストークンを更新します。

min-time-between-jwks-requests

新しい公開鍵を取得するための Red Hat Single Sign-On への2つの要求間の最小間隔を指定する時間 (秒単位)。デフォルトは 10 秒です。アダプターは、不明な **kid** でトークンを認識すると、常に新しい公開鍵をダウンロードしようとします。ただし、10 秒に1回以上試行することはありません (デフォルト)。これは、攻撃者が不正な **kid** 強制アダプターを使用して多数のトークンを送信し、Red Hat Single Sign-On に多数のリクエストを送信するときに DoS を回避するためです。

public-key-cache-ttl

新しい公開鍵を取得する Red Hat Single Sign-On への 2 つのリクエストの最大間隔を指定する時間 (秒単位)。デフォルトは 86400 秒 (1 日) です。アダプターは、不明な **kid** でトークンを認識すると、常に新しい公開鍵をダウンロードしようとします。既知の **kid** でトークンを認識すると、以前ダウンロードした公開鍵のみを使用します。ただし、この設定した間隔 (デフォルトでは1日) につき少なくとも1回は新しい公開鍵がダウンロードされます。トークンの **kid** の規模が分かっている場合でも、常に新しい公開鍵がダウンロードされます。

ignore-oauth-query-parameter

デフォルトは **false** です。**true** に設定すると、ベアラートークン処理の **access_token** クエリーパラメーターの処理がオフになります。**access_token** を渡すだけでは、ユーザーを認証できません

redirect-rewrite-rules

必要場合は、リダイレクト URI の書き換えルールを指定します。これは、キーが Redirect URI を照合する正規表現で、値は代替文字列になります。**\$** 文字は、代替 String の後方参照に使用できません。

verify-token-audience

true に設定すると、ベアラートークンを使用した認証中に、アダプターはトークンにこのクライアント名 (リソース) がオーディエンスとして含まれているかどうかを確認します。このオプションは特に、主にベアラートークンで認証された要求を提供するサービスに有用です。これは、デフォルトでは **false** に設定されますが、セキュリティを強化する場合は、これを有効にすることが推奨されます。オーディエンスサポートの詳細は、「[オーディエンスサポート](#)」を参照してください。

2.1.2. JBoss EAP Adapter

JBoss EAP にデプロイされた WAR アプリケーションのセキュリティを保護するには、Red Hat Single Sign-On アダプターサブシステムをインストールおよび設定する必要があります。次に、WAR のセキュリティを保護する 2 つのオプションがあります。

WAR でアダプター設定ファイルを指定し、auth-method を web.xml 内の KEYCLOAK に変更できます。

または、WAR を修正する必要がなく、**standalone.xml** などの設定ファイルの Red Hat Single Sign-On アダプターサブシステム設定を介してセキュリティ保護できます。本セクションでは、両方の方法を説明します。

2.1.2.1. アダプターのインストール

アダプターは、使用しているサーバーのバージョンに応じて、別のアーカイブとして利用できます。

JBoss EAP 7 にインストールします。

EAP 7 アダプターは、ZIP ファイルを展開するか、RPM を使用してインストールできます。

ZIP ファイルから EAP 7 アダプターをインストールします。

```
$ cd $EAP_HOME
$ unzip rh-sso-7.4.10.GA-eap7-adapter.zip
```

JBoss EAP 6 にインストールします。

EAP 6 アダプターは、ZIP ファイルを展開するか、RPM を使用してインストールできます。

ZIP ファイルから EAP 6 アダプターをインストールします。

```
$ cd $EAP_HOME
$ unzip rh-sso-7.4.10.GA-eap6-adapter.zip
```

この ZIP アーカイブには、Red Hat Single Sign-On アダプターに固有の JBoss モジュールが含まれています。また、アダプターサブシステムを設定するための JBoss CLI スクリプトも含まれています。

サーバーが実行していない場合は adapter サブシステムを設定します。



注記

または、**-Dserver.config=standalone-ha.xml** などの異なる設定を使用してアダプターをインストールするために、コマンドラインからアダプターをインストールする際に **server.config** プロパティーを指定することもできます。

JBoss EAP 7.1 以降

```
$ ./bin/jboss-cli.sh --file=bin/adapter-elytron-install-offline.cli
```



注記

JBoss EAP 6.4 ではオフラインスクリプトは利用できません。

または、サーバーが実行中の場合は、以下を実行します。

JBoss EAP 7.1 以降

```
$ ./bin/jboss-cli.sh -c --file=bin/adapter-elytron-install.cli
```



注記

JBoss EAP 7.1 以上で従来の非 Elytron アダプターを使用することもできます。つまり、**adapter-install-offline.cli**を使用することができます。

JBoss EAP 6.4

```
$ ./bin/jboss-cli.sh -c --file=bin/adaptor-install.cli
```

2.1.2.2. JBoss SSO

JBoss EAP は、同じ JBoss EAP インスタンスにデプロイされた web アプリケーションに対するシングルサインオンのサポートが組み込まれています。これは、Red Hat Single Sign-On を使用する場合に有効にしないでください。

2.1.2.3. 各 WAR に必要な設定

本セクションでは、WAR パッケージ内の設定ファイルを追加および編集して、WAR を直接保護する方法を説明します。

最初に、WAR の **WEB-INF** ディレクトリーに **keycloak.json** アダプター設定ファイルを作成する必要があります。

この設定ファイルの形式は、「[Java アダプター設定](#)」セクションで説明されています。

次に、**web.xml** で **auth-method** を **KEYCLOAK** に設定する必要があります。また、標準のサーブレットセキュリティーを使用して URL に role-base 制約を指定する必要があります。

以下に例を示します。

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_3_0.xsd"
  version="3.0">

  <module-name>application</module-name>

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Admins</web-resource-name>
      <url-pattern>/admin/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>admin</role-name>
    </auth-constraint>
    <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
  </security-constraint>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Customers</web-resource-name>
      <url-pattern>/customers/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>user</role-name>
    </auth-constraint>
    <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
  </security-constraint>
```

```

<login-config>
  <auth-method>KEYCLOAK</auth-method>
  <realm-name>this is ignored currently</realm-name>
</login-config>

<security-role>
  <role-name>admin</role-name>
</security-role>
<security-role>
  <role-name>user</role-name>
</security-role>
</web-app>

```

2.1.2.4. Adapter サブシステムによる WAR のセキュリティ保護

WAR を変更して Red Hat Single Sign-On で保護する必要はありません。代わりに、Red Hat Single Sign-On Adapter サブシステム経由で外部からセキュリティ保護することができます。Keycloak を **auth-method** として指定する必要はありませんが、**web.xml** で **security-constraints** を定義する必要があります。ただし、**WEB-INF/keycloak.json** ファイルを作成する必要はありません。このメタデータは、代わりに Red Hat Single Sign-On サブシステムのサーバー設定 (つまり **standalone.xml** など) で定義されます。

```

<extensions>
  <extension module="org.keycloak.keycloak-adapter-subsystem"/>
</extensions>

<profile>
  <subsystem xmlns="urn:jboss:domain:keycloak:1.1">
    <secure-deployment name="WAR MODULE NAME.war">
      <realm>demo</realm>
      <auth-server-url>http://localhost:8081/auth</auth-server-url>
      <ssl-required>external</ssl-required>
      <resource>customer-portal</resource>
      <credential name="secret">password</credential>
    </secure-deployment>
  </subsystem>
</profile>

```

secure-deployment name 属性は、セキュリティを保護する WAR を識別します。この値は **web.xml** に **.war** を追加した **module-name** です。残りの構成は、[Java アダプター構成](#) で定義された **keycloak.json** 構成オプションとほぼ 1 対 1 で対応しています。

例外は **credential** 要素です。

これを容易にするために、Red Hat Single Sign-On 管理コンソールに移動し、この WAR が連携しているアプリケーションのクライアント/インストールタブに移動します。ここでは、カットアンドペーストできる XML ファイルの例を提供します。

同じレルムでセキュア化された複数のデプロイメントがある場合は、レルム設定を個別の要素で共有できます。以下に例を示します。

```

<subsystem xmlns="urn:jboss:domain:keycloak:1.1">
  <realm name="demo">
    <auth-server-url>http://localhost:8080/auth</auth-server-url>

```

```

    <ssl-required>external</ssl-required>
  </realm>
  <secure-deployment name="customer-portal.war">
    <realm>demo</realm>
    <resource>customer-portal</resource>
    <credential name="secret">password</credential>
  </secure-deployment>
  <secure-deployment name="product-portal.war">
    <realm>demo</realm>
    <resource>product-portal</resource>
    <credential name="secret">password</credential>
  </secure-deployment>
  <secure-deployment name="database.war">
    <realm>demo</realm>
    <resource>database-service</resource>
    <bearer-only>true</bearer-only>
  </secure-deployment>
</subsystem>

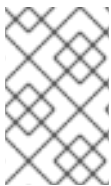
```

2.1.2.5. セキュリティードメイン

セキュリティーコンテキストは EJB 階層に自動的に伝播されます。

2.1.3. RPM からの JBoss EAP アダプターのインストール

RPM から EAP 7 アダプターをインストールします。



注記

Red Hat Enterprise Linux 7 では、チャンネル という用語はリポジトリ という用語に置き換えられました。これらの手順では、リポジトリ という用語のみが使用されています。

RPM から JBoss EAP 7 アダプターをインストールする前に、JBoss EAP 7.3 リポジトリにサブスクライブする必要があります。

前提条件

1. Red Hat Subscription Manager を使用して、Red Hat Enterprise Linux システムがお使いのアカウントに登録されている必要があります。詳細は、[Red Hat Subscription Management のドキュメント](#) を参照してください。
2. すでに別の JBoss EAP リポジトリにサブスクライブしている場合は、最初にそのリポジトリからサブスクライブを解除する必要があります。

Red Hat Enterprise Linux 6、7 の場合: Red Hat Subscription Manager を使用して、以下のコマンドで JBoss EAP 7.3 リポジトリにサブスクライブします。お使いの Red Hat Enterprise Linux のバージョンに応じて、<RHEL_VERSION> を 6 または 7 に置き換えてください。

```
$ sudo subscription-manager repos --enable=jb-eap-7-for-rhel-<RHEL_VERSION>-server-rpms
```

Red Hat Enterprise Linux 8 の場合: Red Hat Subscription Manager を使用して、以下のコマンドで JBoss EAP 7.3 リポジトリにサブスクライブします。


```
$ sudo subscription-manager repos --enable=jb-eap-7.3-for-rhel-8-x86_64-rpms --enable=rhel-8-for-x86_64-baseos-rpms --enable=rhel-8-for-x86_64-appstream-rpms
```

Red Hat Enterprise Linux 6、7 で以下のコマンドを使用して OIDC の JBoss EAP 7 アダプターをインストールします。

```
$ sudo yum install eap7-keycloak-adapter-sso7_4
```

または、Red Hat Enterprise Linux 8 に以下のいずれかを使用します。

```
$ sudo dnf install eap7-keycloak-adapter-sso7_4
```



注記

RPM インストールのデフォルトの EAP_HOME パスは /opt/rh/eap7/root/usr/share/wildfly です。

適切なモジュールインストールスクリプトを実行します。

OIDC モジュールの場合は、以下のコマンドを入力します。

```
$ $EAP_HOME/bin/jboss-cli.sh -c --file=$EAP_HOME/bin/adapter-install.cli
```

インストールが完了しました。

RPM から EAP 6 アダプターをインストールします。



注記

Red Hat Enterprise Linux 7 では、チャンネル という用語はリポジトリという用語に置き換えられました。これらの手順では、リポジトリという用語のみが使用されています。

RPM から EAP 6 アダプターをインストールする前に、JBoss EAP 6 リポジトリにサブスクライブする必要があります。

前提条件

1. Red Hat Subscription Manager を使用して、Red Hat Enterprise Linux システムがお使いのアカウントに登録されている必要があります。詳細は、[Red Hat Subscription Management のドキュメント](#) を参照してください。
- 2.すでに別の JBoss EAP リポジトリにサブスクライブしている場合は、最初にそのリポジトリからサブスクライブを解除する必要があります。

Red Hat Subscription Manager を使用して、以下のコマンドを使用して JBoss EAP 6 リポジトリにサブスクライブします。お使いの Red Hat Enterprise Linux のバージョンに応じて、<RHEL_VERSION> を 6 または 7 に置き換えてください。

```
$ sudo subscription-manager repos --enable=jb-eap-6-for-rhel-<RHEL_VERSION>-server-rpms
```

以下のコマンドを使用して、OIDC の EAP 6 アダプターをインストールします。


```
$ sudo yum install keycloak-adapter-sso7_4-eap6
```



注記

RPM インストールのデフォルトの EAP_HOME パスは /opt/rh/eap6/root/usr/share/wildfly です。

適切なモジュールインストールスクリプトを実行します。

OIDC モジュールの場合は、以下のコマンドを入力します。

```
$ $EAP_HOME/bin/jboss-cli.sh -c --file=$EAP_HOME/bin/adapter-install.cli
```

インストールが完了しました。

2.1.4. JBoss Fuse 6 Adapter

Red Hat Single Sign-On は、[JBoss Fuse 6](#) 内で実行される Web アプリケーションのセキュリティ保護をサポートします。



警告

Fuse 6 のサポートされるバージョンは最新リリースのみです。以前のバージョンの Fuse 6 を使用する場合、一部の関数が正常に機能しない可能性があります。特に、[Hawtio](#) 統合は Fuse 6 の以前のバージョンでは機能しません。

以下の項目のセキュリティは Fuse でサポートされます。

- 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 OSGI として Fuse にデプロイされたサーブレット
- [Camel Jetty](#) コンポーネントで実行している [Apache Camel](#) エンドポイント
- 独自の個別 [Jetty 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](#)

2.1.4.1. Fuse 6 での Web アプリケーションのセキュリティ保護

最初に Red Hat Single Sign-On Karaf 機能をインストールする必要があります。次に、セキュリティ保護するアプリケーションのタイプに応じて手順を実行する必要があります。参照されるすべての Web アプリケーションには、Red Hat Single Sign-On Jetty オーセンティケーターを基盤の Jetty サーバー

に挿入する必要があります。これを行う手順は、アプリケーションの種別によって異なります。詳細は、以下を参照してください。

2.1.4.2. Keycloak 機能のインストール

最初に JBoss Fuse 環境に **keycloak** 機能をインストールする必要があります。keycloak 機能には、Fuse アダプターとサードパーティーのすべての依存関係が含まれます。Maven リポジトリまたはアーカイブからインストールできます。

2.1.4.2.1. Maven リポジトリからのインストール

前提条件として、オンラインで Maven リポジトリにアクセスできる必要がある。

Red Hat Single Sign-On では、最初に適切な Maven リポジトリを設定して、アーティファクトをインストールできます。詳細は、「[JBoss Enterprise Maven リポジトリページ](#)」を参照してください。

Maven リポジトリが <https://maven.repository.redhat.com/ga/> であるとします。

\$FUSE_HOME/etc/org.ops4j.pax.url.mvn.cfg ファイルに以下を追加し、サポートされているリポジトリの一覧にリポジトリを追加します。以下に例を示します。

```
org.ops4j.pax.url.mvn.repositories= \
  https://maven.repository.redhat.com/ga/@id=redhat.product.repo
  http://repo1.maven.org/maven2@id=maven.central.repo, \
  ...
```

Maven リポジトリを使用して keycloak 機能をインストールするには、以下の手順を実行します。

1. JBoss Fuse 6.3.0 を起動し、Karaf 端末タイプで以下を行います。

```
features:addurl mvn:org.keycloak/keycloak-osgi-features/9.0.17.redhat-00001/xml/features
features:install keycloak
```

2. また、Jetty 9 機能をインストールする必要がある場合があります。

```
features:install keycloak-jetty9-adapter
```

3. 機能がインストールされていることを確認します。

```
features:list | grep keycloak
```

2.1.4.2.2. ZIP バンドルからのインストール

これは、オフラインの場合、または Maven を使用して JAR ファイルやその他のアーティファクトを取得したくない場合に便利です。

ZIP アーカイブから Fuse アダプターをインストールするには、以下の手順を行います。

1. Red Hat Single Sign-On Fuse アダプター ZIP アーカイブをダウンロードします。
2. これを JBoss Fuse のルートディレクトリーに展開します。次に、依存関係が **system** ディレクトリーにインストールされます。既存の jar ファイルをすべて上書きできます。
これは JBoss Fuse 6.3.0 に使用します。

```
cd /path-to-fuse/jboss-fuse-6.3.0.redhat-XXX
unzip -q /path-to-adapter-zip/rh-sso-7.4.10.GA-fuse-adapter.zip
```

3. Fuse を起動し、fuse/karaf ターミナルでこれらのコマンドを実行します。

```
features:addurl mvn:org.keycloak/keycloak-osgi-features/9.0.17.redhat-00001/xml/features
features:install keycloak
```

4. 対応する Jetty アダプターをインストールします。アーティファクトは JBoss Fuse **system** ディレクトリーで直接利用できるため、Maven リポジトリを使用する必要はありません。

2.1.4.3. クラシック WAR アプリケーションのセキュリティー保護

WAR アプリケーションをセキュアにするのに必要な手順は次のとおりです。

1. **/WEB-INF/web.xml** ファイルで、必要を宣言します。

- <security-constraint> 要素のセキュリティー制約
- <login-config> 要素のログイン設定
- <security-role> 要素のセキュリティーロール
以下に例を示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0">

  <module-name>customer-portal</module-name>

  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Customers</web-resource-name>
      <url-pattern>/customers/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>user</role-name>
    </auth-constraint>
  </security-constraint>

  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>does-not-matter</realm-name>
  </login-config>

  <security-role>
    <role-name>admin</role-name>
  </security-role>
```

```
<security-role>
  <role-name>user</role-name>
</security-role>
</web-app>
```

2. **/WEB-INF/jetty-web.xml** ファイルにオーセンティケーターのある **jetty-web.xml** ファイルを追加します。
以下に例を示します。

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN"
"http://www.eclipse.org/jetty/configure_9_0.dtd">
<Configure class="org.eclipse.jetty.webapp.WebAppContext">
  <Get name="securityHandler">
    <Set name="authenticator">
      <New class="org.keycloak.adapters.jetty.KeycloakJettyAuthenticator">
      </New>
    </Set>
  </Get>
</Configure>
```

3. WAR の **/WEB-INF/** ディレクトリー内で、新しいファイル **keycloak.json** を作成します。この設定ファイルの形式は、「[Java アダプター設定](#)」のセクションで説明されています。「[外部アダプターの設定](#)」で説明されているように、このファイルを外部から利用できるようにすることもできます。
4. WAR アプリケーションが **org.keycloak.adapters.jetty** をインポートし、さらに **Import-Package** ヘッダーで **META-INF/MANIFEST.MF** ファイルに含まれるパッケージをインポートするようにしてください。プロジェクトに **maven-bundle-plugin** を使用すると、OSGI ヘッダーがマニフェストに適切に生成されます。パッケージの「*」解決は、アプリケーションや Blueprint または Spring 記述子で使用されていないため、**org.keycloak.adapters.jetty** パッケージはインポートしませんが、**jetty-web.xml** ファイルで使用されます。
インポートするパッケージの一覧は、以下のようになります。

```
org.keycloak.adapters.jetty;version="9.0.17.redhat-00001",
org.keycloak.adapters;version="9.0.17.redhat-00001",
org.keycloak.constants;version="9.0.17.redhat-00001",
org.keycloak.util;version="9.0.17.redhat-00001",
org.keycloak.*;version="9.0.17.redhat-00001",
*;resolution:=optional
```

2.1.4.3.1. 外部アダプターの設定

keycloak.json アダプター設定ファイルを WAR アプリケーション内にバンドルせず、命名規則に基づいて外部に利用可能にし、読み込む場合は、この設定方法を使用します。

この機能を有効にするには、このセクションを **/WEB_INF/web.xml** ファイルに追加します。

```
<context-param>
  <param-name>keycloak.config.resolver</param-name>
  <param-value>org.keycloak.adapters.osgi.PathBasedKeycloakConfigResolver</param-value>
</context-param>
```

このコンテキストパラメータは、`org.keycloak.adapters.osgi.PathBasedKeycloakConfigResolver` クラスを使用して、`keycloak.json` ファイルを WAR アプリケーションのクラスパスから読み込むために使用されます。

そのコンポーネントは、Java ノミハティ **keycloak.config** または **karaf.etc** を使用してベースノオルダーを検索し、設定を見つけます。次に、フォルダーのいずれかで **<your_web_context>-keycloak.json** という名前のファイルを検索します。

たとえば、Web アプリケーションにコンテキスト **my-portal** がある場合、アダプター設定は **\$FUSE_HOME/etc/my-portal-keycloak.json** ファイルから読み込まれます。

2.1.4.4. OSGI サービスとしてデプロイされたサーブレットのセキュア化

従来の WAR アプリケーションとしてデプロイされていない OSGI バンドルプロジェクトにサーブレットクラスがある場合は、このメソッドを使用できます。Fuse は Pax Web Whiteboard Extender を使用して、このようなサーブレットを Web アプリケーションとしてデプロイします。

Red Hat Single Sign-On でサーブレットをセキュアにするには、以下の手順を実行します。

1. Red Hat Single Sign-On は、jetty-web.xml を挿入し、アプリケーションのセキュリティー制約の設定を可能にする PaxWebIntegrationService を提供します。このようなサービスをアプリケーション内の **OSGI-INF/blueprint/blueprint.xml** ファイルで宣言する必要があります。サーブレットはそれに依存する必要があることに注意してください。設定例:

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

  <!-- Using jetty bean just for the compatibility with other fuse services -->
  <bean id="servletConstraintMapping" class="org.eclipse.jetty.security.ConstraintMapping">
    <property name="constraint">
      <bean class="org.eclipse.jetty.util.security.Constraint">
        <property name="name" value="cst1"/>
        <property name="roles">
          <list>
            <value>user</value>
          </list>
        </property>
        <property name="authenticate" value="true"/>
        <property name="dataConstraint" value="0"/>
      </bean>
    </property>
    <property name="pathSpec" value="/product-portal/*"/>
  </bean>

  <bean id="keycloakPaxWebIntegration"
    class="org.keycloak.adapters.osgi.PaxWebIntegrationService"
    init-method="start" destroy-method="stop">
    <property name="jettyWebXmlLocation" value="/WEB-INF/jetty-web.xml" />
    <property name="bundleContext" ref="blueprintBundleContext" />
    <property name="constraintMappings">
      <list>
        <ref component-id="servletConstraintMapping" />
      </list>
    </property>
  </bean>

  <bean id="productServlet" class="org.keycloak.example.ProductPortalServlet" depends-
```

```

on="keycloakPaxWebIntegration">
  </bean>

  <service ref="productServlet" interface="javax.servlet.Servlet">
    <service-properties>
      <entry key="alias" value="/product-portal" />
      <entry key="servlet-name" value="ProductServlet" />
      <entry key="keycloak.config.file" value="/keycloak.json" />
    </service-properties>
  </service>

</blueprint>

```

- (プロジェクトが Web アプリケーションではない場合でも) プロジェクト内に **WEB-INF** ディレクトリーがあり、[Classic WAR application](#) セクションに、**/WEB-INF/jetty-web.xml** ファイルおよび **/WEB-INF/keycloak.json** ファイルを作成します。security-constraints が Blueprint 設定ファイルで宣言されているため、**web.xml** ファイルは必要ありません。
2. **META-INF/MANIFEST.MF** の **Import-Package** には、少なくとも以下のインポートが含まれている必要があります。

```

org.keycloak.adapters.jetty;version="9.0.17.redhat-00001",
org.keycloak.adapters;version="9.0.17.redhat-00001",
org.keycloak.constants;version="9.0.17.redhat-00001",
org.keycloak.util;version="9.0.17.redhat-00001",
org.keycloak.*;version="9.0.17.redhat-00001",
*;resolution:=optional

```

2.1.4.5. Apache Camel アプリケーションのセキュリティ保護

[camel-jetty](#) コンポーネントで実装された Apache Camel エンドポイントのセキュリティを保護するには、**KeycloakJettyAuthenticator** で securityHandler を追加し、適切なセキュリティ制約を挿入します。**OSGI-INF/blueprint/blueprint.xml** ファイルを以下のような設定を持つ Camel アプリケーションに追加できます。ロール、セキュリティ制約のマッピング、および Red Hat Single Sign-On アダプター設定は、お使いの環境とニーズに合わせて若干異なる可能性があります。

以下に例を示します。

```

<?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"
  xmlns:camel="http://camel.apache.org/schema/blueprint"
  xsi:schemaLocation="
    http://www.osgi.org/xmlns/blueprint/v1.0.0
    http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
    http://camel.apache.org/schema/blueprint http://camel.apache.org/schema/blueprint/camel-
    blueprint.xsd">

  <bean id="kcAdapterConfig" class="org.keycloak.representations.adapters.config.AdapterConfig">
    <property name="realm" value="demo"/>
    <property name="resource" value="admin-camel-endpoint"/>
    <property name="bearerOnly" value="true"/>
    <property name="authServerUrl" value="http://localhost:8080/auth" />
    <property name="sslRequired" value="EXTERNAL"/>
  </bean>

```

```

</bean>

<bean id="keycloakAuthenticator" class="org.keycloak.adapters.jetty.KeycloakJettyAuthenticator">
  <property name="adapterConfig" ref="kcAdapterConfig"/>
</bean>

<bean id="constraint" class="org.eclipse.jetty.util.security.Constraint">
  <property name="name" value="Customers"/>
  <property name="roles">
    <list>
      <value>admin</value>
    </list>
  </property>
  <property name="authenticate" value="true"/>
  <property name="dataConstraint" value="0"/>
</bean>

<bean id="constraintMapping" class="org.eclipse.jetty.security.ConstraintMapping">
  <property name="constraint" ref="constraint"/>
  <property name="pathSpec" value="/*"/>
</bean>

<bean id="securityHandler" class="org.eclipse.jetty.security.ConstraintSecurityHandler">
  <property name="authenticator" ref="keycloakAuthenticator" />
  <property name="constraintMappings">
    <list>
      <ref component-id="constraintMapping" />
    </list>
  </property>
  <property name="authMethod" value="BASIC"/>
  <property name="realmName" value="does-not-matter"/>
</bean>

<bean id="sessionHandler" class="org.keycloak.adapters.jetty.spi.WrappingSessionHandler">
  <property name="handler" ref="securityHandler" />
</bean>

<bean id="helloProcessor" class="org.keycloak.example.CamelHelloProcessor" />

<camelContext id="blueprintContext"
  trace="false"
  xmlns="http://camel.apache.org/schema/blueprint">
  <route id="httpBridge">
    <from uri="jetty:http://0.0.0.0:8383/admin-camel-endpoint?
handlers=sessionHandler&matchOnUriPrefix=true" />
    <process ref="helloProcessor" />
    <log message="The message from camel endpoint contains ${body}"/>
  </route>
</camelContext>
</blueprint>

```

- **META-INF/MANIFEST.MF** の **Import-Package** には以下のインポートが含まれる必要があります。

```
javax.servlet;version="[3,4)",
```



```

javax.servlet.http;version="[3,4)",
org.apache.camel.*,
org.apache.camel;version="[2.13,3)",
org.eclipse.jetty.security;version="[9,10)",
org.eclipse.jetty.server.nio;version="[9,10)",
org.eclipse.jetty.util.security;version="[9,10)",
org.keycloak.*;version="9.0.17.redhat-00001",
org.osgi.service.blueprint,
org.osgi.service.blueprint.container,
org.osgi.service.event,

```

2.1.4.6. Camel RestDSL

Camel RestDSL は、REST エンドポイントを定義するのに使用される Camel 機能です。しかし、引き続き特定の実装クラスを使用し、Red Hat Single Sign-On との統合方法を説明します。

インテグレーションメカニズムを設定する方法は、RestDSL 定義のルートを設定する Camel コンポーネントに依存します。

以下の例は、Jetty コンポーネントを使用して統合を設定する方法を示しています。これには、上述の Blueprint の例で定義された一部の Bean への参照が含まれます。

```

<bean id="securityHandlerRest" class="org.eclipse.jetty.security.ConstraintSecurityHandler">
  <property name="authenticator" ref="keycloakAuthenticator" />
  <property name="constraintMappings">
    <list>
      <ref component-id="constraintMapping" />
    </list>
  </property>
  <property name="authMethod" value="BASIC"/>
  <property name="realmName" value="does-not-matter"/>
</bean>

<bean id="sessionHandlerRest" class="org.keycloak.adapters.jetty.spi.WrappingSessionHandler">
  <property name="handler" ref="securityHandlerRest" />
</bean>

<camelContext id="blueprintContext"
  trace="false"
  xmlns="http://camel.apache.org/schema/blueprint">

  <restConfiguration component="jetty" contextPath="/restdsl"
    port="8484">
    <!--the link with Keycloak security handlers happens here-->
    <endpointProperty key="handlers" value="sessionHandlerRest"></endpointProperty>
    <endpointProperty key="matchOnUriPrefix" value="true"></endpointProperty>
  </restConfiguration>

  <rest path="/hello" >
    <description>Hello rest service</description>
    <get uri="{id}" outType="java.lang.String">
      <description>Just an hello</description>
      <to uri="direct:justDirect" />
    </get>
  </rest>

```



```

</rest>

<route id="justDirect">
  <from uri="direct:justDirect"/>
  <process ref="helloProcessor" />
  <log message="RestDSL correctly invoked ${body}"/>
  <setBody>
    <constant>(__ This second sentence is returned from a Camel RestDSL
endpoint__)</constant>
  </setBody>
</route>

</camelContext>

```

2.1.4.7. 別の Jetty Engine での Apache CXF エンドポイントのセキュリティー保護

別の Jetty エンジンで Red Hat Single Sign-On によってセキュア化された CXF エンドポイントを実行するには、以下の手順を実行します。

1. **META-INF/spring/beans.xml** をアプリケーションに追加し、そのアプリケーションで、**httpj:engine-factory** が注入された Jetty SecurityHandler で **KeycloakJettyAuthenticator** を宣言します。CFX JAX-WS アプリケーションの設定は以下のようになります。

```

<?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:jaxws="http://cxf.apache.org/jaxws"
  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://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    http://www.springframework.org/schema/osgi
    http://www.springframework.org/schema/osgi/spring-osgi.xsd
    http://cxf.apache.org/transports/http-jetty/configuration
    http://cxf.apache.org/schemas/configuration/http-jetty.xsd">

  <import resource="classpath:META-INF/cxf/cxf.xml" />

  <bean id="kcAdapterConfig"
    class="org.keycloak.representations.adapters.config.AdapterConfig">
    <property name="realm" value="demo"/>
    <property name="resource" value="custom-cxf-endpoint"/>
    <property name="bearerOnly" value="true"/>
    <property name="authServerUrl" value="http://localhost:8080/auth" />
    <property name="sslRequired" value="EXTERNAL"/>
  </bean>

  <bean id="keycloakAuthenticator"
    class="org.keycloak.adapters.jetty.KeycloakJettyAuthenticator">
    <property name="adapterConfig">
      <ref local="kcAdapterConfig" />
    </property>

```

```

</bean>

<bean id="constraint" class="org.eclipse.jetty.util.security.Constraint">
  <property name="name" value="Customers"/>
  <property name="roles">
    <list>
      <value>user</value>
    </list>
  </property>
  <property name="authenticate" value="true"/>
  <property name="dataConstraint" value="0"/>
</bean>

<bean id="constraintMapping" class="org.eclipse.jetty.security.ConstraintMapping">
  <property name="constraint" ref="constraint"/>
  <property name="pathSpec" value="/*"/>
</bean>

<bean id="securityHandler" class="org.eclipse.jetty.security.ConstraintSecurityHandler">
  <property name="authenticator" ref="keycloakAuthenticator" />
  <property name="constraintMappings">
    <list>
      <ref local="constraintMapping" />
    </list>
  </property>
  <property name="authMethod" value="BASIC"/>
  <property name="realmName" value="does-not-matter"/>
</bean>

<httpj:engine-factory bus="cxf" id="kc-cxf-endpoint">
  <httpj:engine port="8282">
    <httpj:handlers>
      <ref local="securityHandler" />
    </httpj:handlers>
    <httpj:sessionSupport>true</httpj:sessionSupport>
  </httpj:engine>
</httpj:engine-factory>

<jaxws:endpoint
  implementor="org.keycloak.example.ws.ProductImpl"
  address="http://localhost:8282/ProductServiceCF" depends-on="kc-cxf-endpoint"
/>

</beans>

```

CXF JAX-RS アプリケーションの場合、唯一の違いは、エンジンファクトリーに依存するエンドポイントの構成にある可能性があります。

```

<jaxrs:server serviceClass="org.keycloak.example.rs.CustomerService"
address="http://localhost:8282/rest"
depends-on="kc-cxf-endpoint">
  <jaxrs:providers>
    <bean class="com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider" />
  </jaxrs:providers>
</jaxrs:server>

```

2. **META-INF/MANIFEST.MF** の **Import-Package** には、これらのインポートが含まれる必要があります。

```
META-INF.cxf;version="[2.7,3.2)",
META-INF.cxf.osgi;version="[2.7,3.2)";resolution:=optional,
org.apache.cxf.bus;version="[2.7,3.2)",
org.apache.cxf.bus.spring;version="[2.7,3.2)",
org.apache.cxf.bus.resource;version="[2.7,3.2)",
org.apache.cxf.transport.http;version="[2.7,3.2)",
org.apache.cxf.*;version="[2.7,3.2)",
org.springframework.beans.factory.config,
org.eclipse.jetty.security;version="[9,10)",
org.eclipse.jetty.util.security;version="[9,10)",
org.keycloak.*;version="9.0.17.redhat-00001"
```

2.1.4.8. デフォルトの Jetty Engine での Apache CXF エンドポイントのセキュリティー保護

一部のサービスは、起動時にデプロイされたサーブレットが自動的に含まれる場合があります。このようなサービスの1つは、`http://localhost:8181/cxf` コンテキストで実行している CXF サーブレットです。このようなエンドポイントの保護は複雑になる可能性があります。現在 Red Hat Single Sign-On が使用している1つの方法は `ServletReregistrationService` で、起動時に組み込みサーブレットをアンデプロイし、Red Hat Single Sign-On がセキュア化されたコンテキストに再デプロイできるようにします。

アプリケーション内の構成ファイル **OSGI-INF/blueprint/blueprint.xml** は、以下のようになります。アプリケーションにエンドポイント固有の JAX-RS **customerservice** エンドポイントを追加しますが、**/cxf** コンテキスト全体を保護することに注意してください。

```
<?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"
  xmlns:jaxrs="http://cxf.apache.org/blueprint/jaxrs"
  xsi:schemaLocation="
http://www.osgi.org/xmlns/blueprint/v1.0.0 http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
http://cxf.apache.org/blueprint/jaxrs http://cxf.apache.org/schemas/blueprint/jaxrs.xsd">

  <!-- JAXRS Application -->

  <bean id="customerBean" class="org.keycloak.example.rs.CxfCustomerService" />

  <jaxrs:server id="cxfJaxrsServer" address="/customerservice">
    <jaxrs:providers>
      <bean class="com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider" />
    </jaxrs:providers>
    <jaxrs:serviceBeans>
      <ref component-id="customerBean" />
    </jaxrs:serviceBeans>
  </jaxrs:server>

  <!-- Securing of whole /cxf context by unregister default cxf servlet from paxweb and re-register
with applied security constraints -->

  <bean id="cxfConstraintMapping" class="org.eclipse.jetty.security.ConstraintMapping">
    <property name="constraint">
      <bean class="org.eclipse.jetty.util.security.Constraint">
```

```

        <property name="name" value="cst1"/>
        <property name="roles">
            <list>
                <value>user</value>
            </list>
        </property>
        <property name="authenticate" value="true"/>
        <property name="dataConstraint" value="0"/>
    </bean>
</property>
<property name="pathSpec" value="/cxf/*"/>
</bean>

<bean id="cxfKeycloakPaxWebIntegration"
class="org.keycloak.adapters.osgi.PaxWebIntegrationService"
    init-method="start" destroy-method="stop">
    <property name="bundleContext" ref="blueprintBundleContext" />
    <property name="jettyWebXmlLocation" value="/WEB-INF/jetty-web.xml" />
    <property name="constraintMappings">
        <list>
            <ref component-id="cxfConstraintMapping" />
        </list>
    </property>
</bean>

<bean id="defaultCxfReregistration"
class="org.keycloak.adapters.osgi.ServletReregistrationService" depends-
on="cxfKeycloakPaxWebIntegration"
    init-method="start" destroy-method="stop">
    <property name="bundleContext" ref="blueprintBundleContext" />
    <property name="managedServiceReference">
        <reference interface="org.osgi.service.cm.ManagedService" filter="
(service.pid=org.apache.cxf.osgi)" timeout="5000" />
    </property>
</bean>

</blueprint>

```

そのため、デフォルトの CXF HTTP 宛先で実行している他のすべての CXF サービスもセキュア化されます。同様に、アプリケーションがアンデプロイされると、**/cxf** コンテキスト全体のセキュリティも保護されなくなります。このため、[「別の Jetty Engine で CXF アプリケーションのセキュリティを保護」](#)で説明されているように、各アプリケーションに独自の Jetty エンジンを使用すると、各アプリケーションのセキュリティをより詳細に制御できます。

- **WEB-INF** ディレクトリーは、(プロジェクトが web アプリケーションではない場合でも) プロジェクト内に置く必要がある場合があります。また、[Classic WAR アプリケーション](#)と同様の方法で **/WEB-INF/keycloak.json** ファイルおよび **/WEB-INF/keycloak.json** ファイルを編集する必要もあります。セキュリティ制約がブループリント設定ファイルで宣言されているため、**web.xml** ファイルは必要ありません。
- **META-INF/MANIFEST.MF** の **Import-Package** には以下のインポートが含まれる必要があります。

```

META-INF.cxf;version="[2.7,3.2)",
META-INF.cxf.osgi;version="[2.7,3.2)";resolution:=optional,
org.apache.cxf.transport.http;version="[2.7,3.2)",

```

```
org.apache.cxf.*;version="[2.7,3.2)",
com.fasterxml.jackson.jaxrs.json;version="[2.5,3)",
org.eclipse.jetty.security;version="[9,10)",
org.eclipse.jetty.util.security;version="[9,10)",
org.keycloak.*;version="9.0.17.redhat-00001",
org.keycloak.adapters.jetty;version="9.0.17.redhat-00001",
*;resolution:=optional
```

2.1.4.9. Fuse 管理サービスのセキュリティー保護

2.1.4.9.1. Fuse ターミナルへの SSH 認証の使用

Red Hat Single Sign-On は、主に Web アプリケーションの認証のユースケースに対応しています。ただし、他の Web サービスおよびアプリケーションが Red Hat Single Sign-On で保護されている場合は、SSH などの非 Web 管理サービスを Red Hat Single Sign-On の認証情報で保護するのが最善の方法です。これは、Red Hat Single Sign-On へのリモート接続を可能にする JAAS ログインモジュールを使用して実行できます。また、「[リソース所有者のパスワード認証情報](#)」に基づいて認証情報を検証できます。

SSH 認証を有効にするには、以下の手順を実行します。

1. Red Hat Single Sign-On では、SSH 認証に使用されるクライアント (**ssh-jmx-admin-client** など) を作成します。このクライアントでは、**Direct Access Grants Enabled** を **On** に選択する必要があります。
2. **\$FUSE_HOME/etc/org.apache.karaf.shell.cfg** ファイルで、以下のプロパティを更新または指定します。

```
sshRealm=keycloak
```

3. 以下のような内容で **\$FUSE_HOME/etc/keycloak-direct-access.json** ファイルを追加します (環境と Red Hat Single Sign-On クライアント設定に基づきます)。

```
{
  "realm": "demo",
  "resource": "ssh-jmx-admin-client",
  "ssl-required" : "external",
  "auth-server-url" : "http://localhost:8080/auth",
  "credentials": {
    "secret": "password"
  }
}
```

このファイルは、SSH 認証の JAAS レルム **keycloak** から JAAS **DirectAccessGrantsLoginModule** によって使用されるクライアントアプリケーション設定を指定します。

4. Fuse を起動し、**keycloak** JAAS レルムをインストールします。最も簡単な方法は、JAAS レルムが事前定義されている **keycloak-jaas** 機能をインストールすることです。より高いランクの独自の JAAS レルム **keycloak** を使用すると、機能の定義済みレルムをオーバーライドできます。詳細は、[JBoss Fuse ドキュメント](#) を参照してください。
Fuse 端末で以下のコマンドを使用します。

```
features:addurl mvn:org.keycloak/keycloak-osgi-features/9.0.17.redhat-00001/xml/features
features:install keycloak-jaas
```

5. 端末に以下を入力して、**admin** ユーザーとして SSH を使用してログインします。

```
ssh -o PubkeyAuthentication=no -p 8101 admin@localhost
```

6. パスワード **password** でログインします。



注記

一部のオペレーティングシステムでは、SSH コマンドの `-o` オプション `-o` オプション `-o HostKeyAlgorithms=+ssh-dss` を使用する必要もあります。これは、後の SSH クライアントはデフォルトで **ssh-dss** アルゴリズムを使用できないためです。しかし、デフォルトでは JBoss Fuse 6.3.0 は現在使用されています。

ユーザーには、すべての操作を実行するためのレلمロール **admin**、または操作のサブセットを実行するための別のロール (たとえば、読み取り専用の Karaf コマンドのみを実行するようにユーザーを制限する **viewer** ロール) が必要であることに注意してください。利用可能なロールは **\$FUSE_HOME/etc/org.apache.karaf.shell.cfg** または **\$FUSE_HOME/etc/system.properties** で設定されます。

2.1.4.9.2. JMX 認証の使用

jconsole または他の外部ツールを使用して RMI 経由で JMX にリモートで接続する場合は、JMX 認証が必要な場合があります。jolokia エージェントはデフォルトで hawtio にインストールされているため、hawtio/jolokia を使用の方が良いでしょう。詳細は「[Hawtio 管理コンソール](#)」を参照してください。

JMX 認証を使用するには、以下の手順を実行します。

1. **\$FUSE_HOME/etc/org.apache.karaf.management.cfg** ファイルで、jmxRealm プロパティを以下のように変更します。

```
jmxRealm=keycloak
```

2. 上記の SSH セクションで説明されているように、**keycloak-jaas** 機能をインストールして **\$FUSE_HOME/etc/keycloak-direct-access.json** ファイルを設定します。

3. jconsole では、以下のような URL を使用できます。

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

および認証情報: admin/password (環境に応じて管理者権限を持つユーザーに基づきます)。

2.1.4.10. Hawtio 管理コンソールのセキュリティ保護

Red Hat Single Sign-On で Hawtio 管理コンソールをセキュアにするには、以下の手順を実行します。

1. これらのプロパティを **\$FUSE_HOME/etc/system.properties** ファイルに追加します。

```
hawtio.keycloakEnabled=true
hawtio.realm=keycloak
hawtio.keycloakClientConfig=file://${karaf.base}/etc/keycloak-hawtio-client.json
```



```
hawtio.rolePrincipalClasses=org.keycloak.adapters.jaas.RolePrincipal,org.apache.karaf.jaas.boot.principal.RolePrincipal
```

2. レルムの Red Hat Single Sign-On 管理コンソールでクライアントを作成します。たとえば、Red Hat Single Sign-On の **demo** レルムでは、クライアント **hawtio-client** を作成し、アクセスタイプとして **public** を指定し、Hawtio を参照するリダイレクト URI (http://localhost:8181/hawtio/*) を指定します。対応する Web Origin も設定する必要があります (この場合は <http://localhost:8181>)。
3. 以下の例のような内容を使用して、**\$FUSE_HOME/etc** ディレクトリーに **keycloak-hawtio-client.json** ファイルを作成します。Red Hat Single Sign-On 環境に応じて、**realm**、**resource**、および **auth-server-url** プロパティーを変更します。**resource** プロパティーは、前のステップで作成したクライアントを参照する必要があります。このファイルは、クライアント (Hawtio JavaScript アプリケーション) で使用されます。

```
{
  "realm": "demo",
  "resource": "hawtio-client",
  "auth-server-url": "http://localhost:8080/auth",
  "ssl-required": "external",
  "public-client": true
}
```

4. 以下の例のような内容を使用して、**\$FUSE_HOME/etc** ディレクトリーに **keycloak-hawtio.json** ファイルを作成します。Red Hat Single Sign-On 環境に応じて、**realm** プロパティーおよび **auth-server-url** プロパティーを変更します。このファイルは、サーバーのアダプター (JAAS ログインモジュール) 側で使用します。

```
{
  "realm": "demo",
  "resource": "jaas",
  "bearer-only": true,
  "auth-server-url": "http://localhost:8080/auth",
  "ssl-required": "external",
  "use-resource-role-mappings": false,
  "principal-attribute": "preferred_username"
}
```

5. JBoss Fuse 6.3.0 を起動し、まだ実行していない場合は keycloak 機能をインストールします。Karaf ターミナルのコマンドの例を以下に示します。

```
features:addurl mvn:org.keycloak/keycloak-osgi-features/9.0.17.redhat-00001/xml/features
features:install keycloak
```

6. <http://localhost:8181/hawtio> にアクセスし、Red Hat Single Sign-On レルムからユーザーとしてログインします。
Hawtio に対して正常に認証するには、適切なレルムロールが必要です。利用可能なロールは、**hawtio.roles** の **\$FUSE_HOME/etc/system.properties** ファイルで設定されます。

2.1.4.10.1. JBoss EAP 6.4 での Hawtio のセキュリティ保護

JBoss EAP 6.4 サーバーで Hawtio を実行するには、以下の手順を実行します。

1. 管理コンソールにログインし、レルムを **demo** に変更します。

1. 前述したように Red Hat Single Sign-On を設定します (Hawtio 管理コンソールのセキュア化)。以下が前提となります。
 - Red Hat Single Sign-On レルムの **demo** およびクライアント **hawtio-client** がある。
 - Red Hat Single Sign-On が **localhost:8080** で実行されている。
 - デプロイされた Hawtio を持つ JBoss EAP 6.4 サーバーは **localhost:8181** で実行されます。このサーバーのディレクトリーは、次のステップで **\$EAP_HOME** と呼ばれます。
2. JBoss Fuse 6.3.0 バージョンに一致する Hawtio Web アーカイブを **\$EAP_HOME/standalone/configuration** ディレクトリーにコピーします。Hawtio のデプロイメントに関する詳細は、[Fuse Hawtio のドキュメント](#) を参照してください。
3. 上記の内容の **keycloak-hawtio.json** および **keycloak-hawtio-client.json** ファイルを **\$EAP_HOME/standalone/configuration** ディレクトリーにコピーします。
4. [JBoss アダプターのドキュメント](#) で説明されているように、Red Hat Single Sign-On アダプターサブシステムを JBoss EAP 6.4 サーバーにインストールします。
5. **\$EAP_HOME/standalone/configuration/standalone.xml** ファイルで、以下の例のようにシステムプロパティーを設定します。

```
<extensions>
...
</extensions>

<system-properties>
  <property name="hawtio.authenticationEnabled" value="true" />
  <property name="hawtio.realm" value="hawtio" />
  <property name="hawtio.roles" value="admin,viewer" />
  <property name="hawtio.rolePrincipalClasses"
value="org.keycloak.adapters.jaas.RolePrincipal" />
  <property name="hawtio.keycloakEnabled" value="true" />
  <property name="hawtio.keycloakClientConfig" value="${jboss.server.config.dir}/keycloak-
hawtio-client.json" />
  <property name="hawtio.keycloakServerConfig" value="${jboss.server.config.dir}/keycloak-
hawtio.json" />
</system-properties>
```

6. Hawtio レルムを **security-domains** セクションの同じファイルに追加します。

```
<security-domain name="hawtio" cache-type="default">
  <authentication>
    <login-module code="org.keycloak.adapters.jaas.BearerTokenLoginModule"
flag="required">
      <module-option name="keycloak-config-file"
value="${hawtio.keycloakServerConfig}"/>
    </login-module>
  </authentication>
</security-domain>
```

7. **Secure-deployment** セクション **hawtio** をアダプターサブシステムに追加します。これにより、Hawtio WAR が JAAS ログインモジュールクラスを検索できるようになります。


```
<subsystem xmlns="urn:jboss:domain:keycloak:1.1">
  <secure-deployment name="your-hawtio-webarchive.war" />
</subsystem>
```

8. Hawtio を使用して JBoss EAP 6.4 サーバーを再起動します。

```
cd $EAP_HOME/bin
./standalone.sh -Djboss.socket.binding.port-offset=101
```

9. Hawtio (<http://localhost:8181/hawtio>) にアクセスします。Red Hat Single Sign-On でセキュリティが保護されています。

2.1.5. JBoss Fuse 7 Adapter

Red Hat Single Sign-On は、JBoss Fuse 7 内で実行している Web アプリケーションのセキュリティ保護をサポートします。

JBoss Fuse 7.x は Undertow [HTTP エンジン](#) にバンドルされ、Undertow はさまざまな種類の Web アプリケーションの実行に使用されるため、JBoss Fuse 7 は基本的に JBoss [EAP 7 アダプター](#) と同じ Undertow アダプターを使用します。



警告

唯一サポートされるのは Fuse 7 の最新バージョンです。以前のバージョンの Fuse 7 を使用する場合は、一部の関数が正常に機能しない可能性があります。特に、統合は 7.0.1 未満の Fuse 7 バージョンですべてのバージョンで機能しません。

以下の項目のセキュリティは Fuse でサポートされます。

- 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](#)

2.1.5.1. Fuse 7 での Web アプリケーションのセキュリティ保護

最初に Red Hat Single Sign-On Karaf 機能をインストールする必要があります。次に、セキュリティ保護するアプリケーションのタイプに応じて手順を実行する必要があります。参照されるすべての Web

アプリケーションには、Red Hat Single Sign-On の Undertow 認証メカニズムを基盤の Web サーバーに挿入する必要があります。これを行う手順は、アプリケーションの種別によって異なります。詳細は、以下を参照してください。

2.1.5.2. Keycloak 機能のインストール

最初に、JBoss Fuse 環境に **keycloak-pax-http-undertow** 機能および **keycloak-jaas** 機能をインストールする必要があります。**keycloak-pax-http-undertow** 機能には、Fuse アダプターとサードパーティーのすべての依存関係が含まれます。**keycloak-jaas** には、SSH および JMX 認証にレلمで使用する JAAS モジュールが含まれます。Maven リポジトリまたはアーカイブからインストールできます。

2.1.5.2.1. Maven リポジトリからのインストール

前提条件として、オンラインで Maven リポジトリにアクセスできる必要がある。

Red Hat Single Sign-On では、最初に適切な Maven リポジトリを設定して、アーティファクトをインストールできます。詳細は、「[JBoss Enterprise Maven リポジトリページ](#)」を参照してください。

Maven リポジトリが <https://maven.repository.redhat.com/ga/> であるとします。

\$FUSE_HOME/etc/org.ops4j.pax.url.mvn.cfg ファイルに以下を追加し、サポートされているリポジトリの一覧にリポジトリを追加します。以下に例を示します。

```
config:edit org.ops4j.pax.url.mvn
config:property-append org.ops4j.pax.url.mvn.repositories
,https://maven.repository.redhat.com/ga/@id=redhat.product.repo
config:update

feature:repo-refresh
```

Maven リポジトリを使用して keycloak 機能をインストールするには、以下の手順を実行します。

1. JBoss Fuse 7.x を起動し、Karaf 端末タイプで以下を行います。

```
feature:repo-add mvn:org.keycloak/keycloak-osgi-features/9.0.17.redhat-00001/xml/features
feature:install keycloak-pax-http-undertow keycloak-jaas
```

2. Undertow 機能のインストールが必要になる場合があります。

```
feature:install pax-http-undertow
```

3. 機能がインストールされていることを確認します。

```
feature:list | grep keycloak
```

2.1.5.2.2. ZIP バンドルからのインストール

これは、オフラインの場合、または Maven を使用して JAR ファイルやその他のアーティファクトを取得したくない場合に便利です。

ZIP アーカイブから Fuse アダプターをインストールするには、以下の手順を行います。

1. Red Hat Single Sign-On Fuse アダプター ZIP アーカイブをダウンロードします。

2. これを JBoss Fuse のルートディレクトリーに展開します。次に、依存関係が **system** ディレクトリーにインストールされます。既存の jar ファイルをすべて上書きできます。
これは JBoss Fuse 7.x に使用します。

```
cd /path-to-fuse/fuse-karaf-7.x
unzip -q /path-to-adapter-zip/rh-ssso-7.4.10.GA-fuse-adapter.zip
```

3. Fuse を起動し、fuse/karaf ターミナルでこれらのコマンドを実行します。

```
feature:repo-add mvn:org.keycloak/keycloak-osgi-features/9.0.17.redhat-00001/xml/features
feature:install keycloak-pax-http-undertow keycloak-jaas
```

4. 対応する Undertow アダプターをインストールします。アーティファクトは JBoss Fuse **system** ディレクトリーで直接利用できるため、Maven リポジトリを使用する必要はありません。

2.1.5.3. クラシック WAR アプリケーションのセキュリティー保護

WAR アプリケーションをセキュアにするのに必要な手順は次のとおりです。

1. **/WEB-INF/web.xml** ファイルで、必要を宣言します。

- <security-constraint> 要素のセキュリティー制約
- <login-config> 要素のログイン設定。<auth-method> が **KEYCLOAK** であることを確認します。
- <security-role> 要素のセキュリティーロール
以下に例を示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0">

  <module-name>customer-portal</module-name>

  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Customers</web-resource-name>
      <url-pattern>/customers/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>user</role-name>
    </auth-constraint>
  </security-constraint>

  <login-config>
    <auth-method>KEYCLOAK</auth-method>
```

```

    <realm-name>does-not-matter</realm-name>
  </login-config>

  <security-role>
    <role-name>admin</role-name>
  </security-role>
  <security-role>
    <role-name>user</role-name>
  </security-role>
</web-app>

```

2. WAR の **/WEB-INF/** ディレクトリー内で、新しいファイル `keycloak.json` を作成します。この設定ファイルの形式は、「[Java アダプター設定](#)」のセクションで説明されています。「[外部アダプターの設定](#)」で説明されているように、このファイルを外部から利用できるようにすることもできます。

以下に例を示します。

```

{
  "realm": "demo",
  "resource": "customer-portal",
  "auth-server-url": "http://localhost:8080/auth",
  "ssl-required": "external",
  "credentials": {
    "secret": "password"
  }
}

```

3. Fuse 6 アダプターとは異なり、MANIFEST.MF には特別な OSGi インポートは必要ありません。

2.1.5.3.1. 設定リゾルバー

keycloak.json アダプター設定ファイルは、バンドル内に保存する (デフォルトの動作) が、ファイルシステムのディレクトリーに保存することができます。設定ファイルの実際のソースを指定するには、**keycloak.config.resolver** デプロイメントパラメーターを希望の設定リゾルバークラスに設定します。たとえば、従来の WAR アプリケーションで、以下のように **web.xml** ファイルに **keycloak.config.resolver** コンテキストパラメーターを設定します。

```

<context-param>
  <param-name>keycloak.config.resolver</param-name>
  <param-value>org.keycloak.adapters.osgi.PathBasedKeycloakConfigResolver</param-value>
</context-param>

```

以下のリゾルバーは **keycloak.config.resolver** で利用できます。

org.keycloak.adapters.osgi.BundleBasedKeycloakConfigResolver

これはデフォルトのリゾルバーです。設定ファイルは、セキュリティが保護されている OSGi バンドル内に想定されるものです。デフォルトでは、**WEB-INF/keycloak.json** という名前のファイルを読み込みますが、このファイル名は **configLocation** プロパティーで設定できます。

org.keycloak.adapters.osgi.PathBasedKeycloakConfigResolver

このリゾルバーは、**keycloak.config** システムプロパティーで指定されるディレクトリー内の **<your_web_context>-keycloak.json** という名前のファイルを検索します。**keycloak.config** が設定されていない場合は、代わりに **karaf.etc** システムプロパティーが使用されます。

たとえば、Web アプリケーションがコンテキスト **my-portal** にデプロイされている場合、アダプター設定は、**\${keycloak.config}/my-portal-keycloak.json** ファイルまたは **\${karaf.etc}/my-portal-keycloak.json** ファイルから読み込まれます。

org.keycloak.adapters.osgi.HierarchicalPathBasedKeycloakConfigResolver

このリゾルバーは、上記の **PathBasedKeycloakConfigResolver** と似ています。指定されたURIパスについて、構成場所が最も具体的なものから順にチェックされます。

たとえば、**/my/web-app/context** URI の場合は、最初の値が見つかるまで、以下の設定場所が検索されます。

- **\${karaf.etc}/my-web-app-context-keycloak.json**
- **\${karaf.etc}/my-web-app-keycloak.json**
- **\${karaf.etc}/my-keycloak.json**
- **\${karaf.etc}/keycloak.json**

2.1.5.4. OSGI サービスとしてデプロイされたサーブレットのセキュア化

従来の WAR アプリケーションとしてデプロイされていない OSGI バンドルプロジェクトにサーブレットクラスがある場合は、このメソッドを使用できます。Fuse は Pax Web Whiteboard Extender を使用して、このようなサーブレットを Web アプリケーションとしてデプロイします。

Red Hat Single Sign-On でサーブレットをセキュアにするには、以下の手順を実行します。

1. Red Hat Single Sign-On

は、**org.keycloak.adapters.osgi.undertow.PaxWebIntegrationService** を提供します。これにより、アプリケーションの認証方法とセキュリティ制約を設定できます。このようなサービスをアプリケーション内の **OSGI-INF/blueprint/blueprint.xml** ファイルで宣言する必要があります。サーブレットはそれに依存する必要があることに注意してください。設定例:

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

  <bean id="servletConstraintMapping"
class="org.keycloak.adapters.osgi.undertow.PaxWebSecurityConstraintMapping">
    <property name="roles">
      <list>
        <value>user</value>
      </list>
    </property>
    <property name="authentication" value="true"/>
    <property name="url" value="/product-portal/*"/>
  </bean>

  <!-- This handles the integration and setting the login-config and security-constraints
parameters -->
  <bean id="keycloakPaxWebIntegration"
class="org.keycloak.adapters.osgi.undertow.PaxWebIntegrationService"
    init-method="start" destroy-method="stop">
    <property name="bundleContext" ref="blueprintBundleContext" />
```

```

    <property name="constraintMappings">
      <list>
        <ref component-id="servletConstraintMapping" />
      </list>
    </property>
  </bean>

  <bean id="productServlet" class="org.keycloak.example.ProductPortalServlet" depends-
on="keycloakPaxWebIntegration" />

  <service ref="productServlet" interface="javax.servlet.Servlet">
    <service-properties>
      <entry key="alias" value="/product-portal" />
      <entry key="servlet-name" value="ProductServlet" />
      <entry key="keycloak.config.file" value="/keycloak.json" />
    </service-properties>
  </service>
</blueprint>

```

- (プロジェクトが Web アプリケーションではない場合でも) プロジェクト内に **WEB-INF** ディレクトリーがあり、[Classic WAR application](#) で説明されているように、**/WEB-INF/keycloak.json** ファイルを作成します。security-constraints が Blueprint 設定ファイルで宣言されているため、**web.xml** ファイルは必要ありません。

2. Fuse 6 アダプターとは異なり、MANIFEST.MF には特別な OSGi インポートは必要ありません。

2.1.5.5. Apache Camel アプリケーションのセキュリティ保護

Blueprint で適切なセキュリティ制約を注入し、使用されているコンポーネントを **undertow-keycloak** に更新することで、[camel-undertow](#) コンポーネントで実装された Apache Camel エンドポイントのセキュリティを保護できます。**OSGI-INF/blueprint/blueprint.xml** ファイルを、以下のような設定を持つ Camel アプリケーションに追加する必要があります。ルール、セキュリティ制約のマッピング、およびアダプター設定は、お使いの環境とニーズに合わせて若干異なる可能性があります。

undertow-keycloak コンポーネントは、標準の **undertow** コンポーネントと比較すると、2つの新しいプロパティーを追加します。

- **configResolver** は、Red Hat Single Sign-On アダプター設定を提供するリゾルバー Bean です。利用可能なリゾルバーは [Configuration Resolvers](#) セクションに記載されています。
- **allowedRoles** はロールのコンマ区切りの一覧です。サービスにアクセスするユーザーは、アクセスを許可するために少なくとも1つのロールが必要です。

以下に例を示します。

```

<?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"
  xmlns:camel="http://camel.apache.org/schema/blueprint"
  xsi:schemaLocation="
    http://www.osgi.org/xmlns/blueprint/v1.0.0
    http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
    http://camel.apache.org/schema/blueprint http://camel.apache.org/schema/blueprint/camel-
    blueprint-2.17.1.xsd">

```



```

<bean id="keycloakConfigResolver"
class="org.keycloak.adapters.osgi.BundleBasedKeycloakConfigResolver" >
  <property name="bundleContext" ref="blueprintBundleContext" />
</bean>

<bean id="helloProcessor" class="org.keycloak.example.CamelHelloProcessor" />

<camelContext id="blueprintContext"
  trace="false"
  xmlns="http://camel.apache.org/schema/blueprint">

  <route id="httpBridge">
    <from uri="undertow-keycloak:http://0.0.0.0:8383/admin-camel-endpoint?
matchOnUriPrefix=true&configResolver=#keycloakConfigResolver&allowedRoles=admin"
/>
    <process ref="helloProcessor" />
    <log message="The message from camel endpoint contains ${body}" />
  </route>

</camelContext>

</blueprint>

```

- **META-INF/MANIFEST.MF** の **Import-Package** には以下のインポートが含まれる必要があります。

```

javax.servlet;version="[3,4)",
javax.servlet.http;version="[3,4)",
javax.net.ssl,
org.apache.camel.*,
org.apache.camel;version="[2.13,3)",
io.undertow.*,
org.keycloak.*;version="9.0.17.redhat-00001",
org.osgi.service.blueprint,
org.osgi.service.blueprint.container

```

2.1.5.6. Camel RestDSL

Camel RestDSL は、REST エンドポイントを定義するのに使用される Camel 機能です。しかし、引き続き特定の実装クラスを使用し、Red Hat Single Sign-On との統合方法を説明します。

インテグレーションメカニズムを設定する方法は、RestDSL 定義のルートを設定する Camel コンポーネントに依存します。

以下の例は、**undertow-keycloak** コンポーネントを使用して統合を設定する方法を示しています。これには、上述の Blueprint の例で定義された一部の Bean への参照が含まれます。

```

<camelContext id="blueprintContext"
  trace="false"
  xmlns="http://camel.apache.org/schema/blueprint">

  <!-- the link with Keycloak security handlers happens by using undertow-keycloak component -->
  <restConfiguration apiComponent="undertow-keycloak" contextPath="/restdsl" port="8484">
    <endpointProperty key="configResolver" value="#keycloakConfigResolver" />
    <endpointProperty key="allowedRoles" value="admin,superadmin" />
  </restConfiguration>
</camelContext>

```

```

</restConfiguration>

<rest path="/hello" >
  <description>Hello rest service</description>
  <get uri="{id}" outType="java.lang.String">
    <description>Just a hello</description>
    <to uri="direct:justDirect" />
  </get>
</rest>

<route id="justDirect">
  <from uri="direct:justDirect"/>
  <process ref="helloProcessor" />
  <log message="RestDSL correctly invoked ${body}"/>
  <setBody>
    <constant>(__This second sentence is returned from a Camel RestDSL
endpoint__)</constant>
  </setBody>
</route>

</camelContext>

```

2.1.5.7. 別の Undertow エンジンでの Apache CXF エンドポイントのセキュリティー保護

別の Undertow エンジンで Red Hat Single Sign-On によってセキュア化された CXF エンドポイントを実行するには、以下の手順を実行します。

1. **OSGI-INF/blueprint/blueprint.xml** をアプリケーションに追加し、[Camel 設定](#) と同様に適切な設定リゾルバー Bean を追加します。**httpu:engine-factory** では、camel 設定を使用した **org.keycloak.adapters.osgi.undertow.CxfKeycloakAuthHandler** ハンドラーを宣言します。CFX JAX-WS アプリケーションの設定は以下のようになります。

```

<?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"
  xmlns:jaxws="http://cxf.apache.org/blueprint/jaxws"
  xmlns:cxf="http://cxf.apache.org/blueprint/core"
  xmlns:httpu="http://cxf.apache.org/transport/http-undertow/configuration".
  xsi:schemaLocation="
    http://cxf.apache.org/transport/http-undertow/configuration
    http://cxf.apache.org/schemas/configuration/http-undertow.xsd
    http://cxf.apache.org/blueprint/core http://cxf.apache.org/schemas/blueprint/core.xsd
    http://cxf.apache.org/blueprint/jaxws http://cxf.apache.org/schemas/blueprint/jaxws.xsd">

  <bean id="keycloakConfigResolver"
class="org.keycloak.adapters.osgi.BundleBasedKeycloakConfigResolver" >
    <property name="bundleContext" ref="blueprintBundleContext" />
  </bean>

  <httpu:engine-factory bus="cxf" id="kc-cxf-endpoint">
    <httpu:engine port="8282">
      <httpu:handlers>
        <bean class="org.keycloak.adapters.osgi.undertow.CxfKeycloakAuthHandler">
          <property name="configResolver" ref="keycloakConfigResolver" />

```



```

        </bean>
    </http:handlers>
</http:engine>
</http:engine-factory>

    <jaxws:endpoint implementor="org.keycloak.example.ws.ProductImpl"
        address="http://localhost:8282/ProductServiceCF" depends-on="kc-cxf-
endpoint"/>

</blueprint>

```

CXF JAX-RS アプリケーションの場合、唯一の違いは、エンジンファクトリーに依存するエンドポイントの構成にある可能性があります。

```

<jaxrs:server serviceClass="org.keycloak.example.rs.CustomerService"
address="http://localhost:8282/rest"
depends-on="kc-cxf-endpoint">
    <jaxrs:providers>
        <bean class="com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider" />
    </jaxrs:providers>
</jaxrs:server>

```

2. **META-INF/MANIFEST.MF** の **Import-Package** には、これらのインポートが含まれる必要があります。

```

META-INF.cxf;version="[2.7,3.3)",
META-INF.cxf.osgi;version="[2.7,3.3)";resolution:=optional,
org.apache.cxf.bus;version="[2.7,3.3)",
org.apache.cxf.bus.spring;version="[2.7,3.3)",
org.apache.cxf.bus.resource;version="[2.7,3.3)",
org.apache.cxf.transport.http;version="[2.7,3.3)",
org.apache.cxf.*;version="[2.7,3.3)",
org.springframework.beans.factory.config,
org.keycloak.*;version="9.0.17.redhat-00001"

```

2.1.5.8. デフォルトの Undertow エンジンでの Apache CXF エンドポイントのセキュリティー保護

一部のサービスは、起動時にデプロイされたサーブレットが自動的に含まれる場合があります。このようなサービスの1つは、`http://localhost:8181/cxf` コンテキストで実行している CXF サーブレットです。Fuse の Pax Web は、設定 `admin` を使用した既存のコンテキストの変更をサポートします。これは、Red Hat Single Sign-On によるエンドポイントを保護するのに使用できます。

アプリケーション内の構成ファイル **OSGI-INF/blueprint/blueprint.xml** は、以下のようになります。アプリケーションへのエンドポイント固有の JAX-RS **customerservice** エンドポイントを追加することに注意してください。

```

<?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"
    xmlns:jaxrs="http://cxf.apache.org/blueprint/jaxrs"
    xsi:schemaLocation="
http://www.osgi.org/xmlns/blueprint/v1.0.0 http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
http://cxf.apache.org/blueprint/jaxrs http://cxf.apache.org/schemas/blueprint/jaxrs.xsd">

```

```

<!-- JAXRS Application -->
<bean id="customerBean" class="org.keycloak.example.rs.CxfCustomerService" />

<jaxrs:server id="cxfJaxrsServer" address="/customerservice">
  <jaxrs:providers>
    <bean class="com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider" />
  </jaxrs:providers>
  <jaxrs:serviceBeans>
    <ref component-id="customerBean" />
  </jaxrs:serviceBeans>
</jaxrs:server>
</blueprint>

```

`/${karaf.etc}/org.ops4j.pax.web.context-anyName.cfg` ファイルを作成する必要があります。これは、**pax-web-runtime** バンドルによって追跡されるファクトリー PID 設定として処理されます。このような設定には、標準の **web.xml** のプロパティの一部に対応する以下のプロパティが含まれる場合があります。

```

bundle.symbolicName = org.apache.cxf.cxf-rt-transport-http
context.id = default

context.param.keycloak.config.resolver =
org.keycloak.adapters.osgi.HierarchicalPathBasedKeycloakConfigResolver

login.config.authMethod = KEYCLOAK

security.cxf.url = /cxf/customerservice/*
security.cxf.roles = admin, user

```

設定管理ファイルで利用可能なプロパティの完全な説明は、Fuse のドキュメントを参照してください。上記のプロパティは以下の意味になります。

bundle.symbolicName および **context.id**

org.ops4j.pax.web.service.WebContainer 内のバンドルおよびそのデプロイメントコンテキストを特定します。

context.param.keycloak.config.resolver

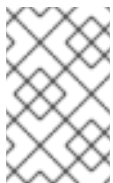
keycloak.config.resolver コンテキストパラメーターの値を、従来の WAR の **web.xml** と同じバンドルに提供します。利用可能なリゾルバーは、「[設定リゾルバー](#)」を参照してください。

login.config.authMethod

認証方法。**KEYCLOAK** でなければなりません。

security.anyName.url および **security.anyName.roles**

個別のセキュリティ制約のプロパティの値は、**web.xml** の **security-constraint/web-resource-collection/url-pattern** および **security-constraint/auth-constraint/role-name** に設定されます。ロールは、コンマとその周囲の空白で区切られます。**anyName** 識別子は任意ですが、同じセキュリティ制約の個別のプロパティに一致する必要があります。



注記

一部の Fuse バージョンには、ロールを "," (コンマと単一スペース) で区切る必要があるバグが含まれています。ロールを分離するには、必ずこの表記を使用してください。

META-INF/MANIFEST.MF の **Import-Package** には、少なくとも以下のインポートが含まれている必要があります。

```
javax.ws.rs;version="[2,3)",
META-INF.cxf;version="[2.7,3.3)",
META-INF.cxf.osgi;version="[2.7,3.3)";resolution:=optional,
org.apache.cxf.transport.http;version="[2.7,3.3)",
org.apache.cxf.*;version="[2.7,3.3)",
com.fasterxml.jackson.jaxrs.json;version="${jackson.version}"
```

2.1.5.9. Fuse 管理サービスのセキュリティー保護

2.1.5.9.1. Fuse ターミナルへの SSH 認証の使用

Red Hat Single Sign-On は、主に Web アプリケーションの認証のユースケースに対応しています。ただし、他の Web サービスおよびアプリケーションが Red Hat Single Sign-On で保護されている場合は、SSH などの非 Web 管理サービスを Red Hat Single Sign-On の認証情報で保護するのが最善の方法です。これは、Red Hat Single Sign-On へのリモート接続を可能にする JAAS ログインモジュールを使用して実行できます。また、「[リソース所有者のパスワード認証情報](#)」に基づいて認証情報を検証できます。

SSH 認証を有効にするには、以下の手順を実行します。

1. Red Hat Single Sign-On では、SSH 認証に使用されるクライアント (**ssh-jmx-admin-client** など) を作成します。このクライアントでは、**Direct Access Grants Enabled** を **On** に選択する必要があります。
2. **\$FUSE_HOME/etc/org.apache.karaf.shell.cfg** ファイルで、以下のプロパティを更新または指定します。

```
sshRealm=keycloak
```

3. 以下のような内容で **\$FUSE_HOME/etc/keycloak-direct-access.json** ファイルを追加します (環境と Red Hat Single Sign-On クライアント設定に基づきます)。

```
{
  "realm": "demo",
  "resource": "ssh-jmx-admin-client",
  "ssl-required": "external",
  "auth-server-url": "http://localhost:8080/auth",
  "credentials": {
    "secret": "password"
  }
}
```

このファイルは、SSH 認証の JAAS レルム **keycloak** から JAAS **DirectAccessGrantsLoginModule** によって使用されるクライアントアプリケーション設定を指定します。

4. Fuse を起動し、**keycloak** JAAS レルムをインストールします。最も簡単な方法は、JAAS レルムが事前定義されている **keycloak-jaas** 機能をインストールすることです。より高いランクの独自の JAAS レルム **keycloak** を使用すると、機能の定義済みレルムをオーバーライドできます。詳細は、[JBoss Fuse ドキュメント](#) を参照してください。
Fuse 端末で以下のコマンドを使用します。

```
features:addurl mvn:org.keycloak/keycloak-osgi-features/9.0.17.redhat-00001/xml/features
features:install keycloak-jaas
```

5. 端末に以下を入力して、**admin** ユーザーとして SSH を使用してログインします。

```
ssh -o PubkeyAuthentication=no -p 8101 admin@localhost
```

6. パスワード **password** でログインします。



注記

一部のオペレーティングシステムでは、SSH コマンドの `-o` オプション `-o` オプション `-o HostKeyAlgorithms=+ssh-dss` を使用する必要もあります。これは、後の SSH クライアントはデフォルトで **ssh-dss** アルゴリズムを使用できないためです。しかし、デフォルトでは JBoss Fuse 7.x で使用されています。

ユーザーには、すべての操作を実行するためのレلمロール **admin**、または操作のサブセットを実行するための別のロール (たとえば、読み取り専用の Karaf コマンドのみを実行するようにユーザーを制限する **viewer** ロール) が必要であることに注意してください。利用可能なロールは **\$FUSE_HOME/etc/org.apache.karaf.shell.cfg** または **\$FUSE_HOME/etc/system.properties** で設定されます。

2.1.5.9.2. JMX 認証の使用

jconsole または他の外部ツールを使用して RMI 経由で JMX にリモートで接続する場合は、JMX 認証が必要な場合があります。jolokia エージェントはデフォルトで hawtio にインストールされているため、hawtio/jolokia を使用の方が良いでしょう。詳細は「[Hawtio 管理コンソール](#)」を参照してください。

JMX 認証を使用するには、以下の手順を実行します。

1. **\$FUSE_HOME/etc/org.apache.karaf.management.cfg** ファイルで、jmxRealm プロパティを以下のように変更します。

```
jmxRealm=keycloak
```

2. 上記の SSH セクションで説明されているように、**keycloak-jaas** 機能をインストールして **\$FUSE_HOME/etc/keycloak-direct-access.json** ファイルを設定します。

3. jconsole では、以下のような URL を使用できます。

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

および認証情報: admin/password (環境に応じて管理者権限を持つユーザーに基づきます)。

2.1.5.10. Hawtio 管理コンソールのセキュリティ保護

Red Hat Single Sign-On で Hawtio 管理コンソールをセキュアにするには、以下の手順を実行します。

1. レلمの Red Hat Single Sign-On 管理コンソールでクライアントを作成します。たとえば、Red Hat Single Sign-On の **demo** レلمでは、クライアント **hawtio-client** を作成し、アクセスタイプとして **public** を指定し、Hawtio を参照するリダイレクト URI (http://localhost:8181/hawtio/*) を指定します。対応する Web Origin (この場合は

http://localhost:8181) を設定します。**hawtio-client** クライアント詳細の **Scope** タブにある **account** クライアントの **view-profile** クライアントロールを含むように、クライアントスコープマッピングを設定します。

- 以下の例のような内容を使用して、**\$FUSE_HOME/etc** ディレクトリーに **keycloak-hawtio-client.json** ファイルを作成します。Red Hat Single Sign-On 環境に応じて、**realm**、**resource**、および **auth-server-url** プロパティーを変更します。**resource** プロパティーは、前のステップで作成したクライアントを参照する必要があります。このファイルは、クライアント (Hawtio JavaScript アプリケーション) で使用されます。

```
{
  "realm": "demo",
  "clientId": "hawtio-client",
  "url": "http://localhost:8080/auth",
  "ssl-required": "external",
  "public-client": true
}
```

- 以下の例のような内容を使用して、**\$FUSE_HOME/etc** ディレクトリーに **keycloak-direct-access.json** ファイルを作成します。Red Hat Single Sign-On 環境に応じて、**realm** プロパティーおよび **url** プロパティーを変更します。このファイルは JavaScript クライアントによって使用されます。

```
{
  "realm": "demo",
  "resource": "ssh-jmx-admin-client",
  "auth-server-url": "http://localhost:8080/auth",
  "ssl-required": "external",
  "credentials": {
    "secret": "password"
  }
}
```

- 以下の例のような内容を使用して、**\$FUSE_HOME/etc** ディレクトリーに **keycloak-hawtio.json** ファイルを作成します。Red Hat Single Sign-On 環境に応じて、**realm** プロパティーおよび **auth-server-url** プロパティーを変更します。このファイルは、サーバーのアダプター (JAAS ログインモジュール) 側で使用します。

```
{
  "realm": "demo",
  "resource": "jaas",
  "bearer-only": true,
  "auth-server-url": "http://localhost:8080/auth",
  "ssl-required": "external",
  "use-resource-role-mappings": false,
  "principal-attribute": "preferred_username"
}
```

- JBoss Fuse 7.x を起動し、[Keycloak 機能をインストール](#) します。次に、Karaf ターミナルで以下を入力します。

```
system:property -p hawtio.keycloakEnabled true
system:property -p hawtio.realm keycloak
system:property -p hawtio.keycloakClientConfig file://\${karaf.base}/etc/keycloak-hawtio-client.json
```

```
system:property -p hawtio.rolePrincipalClasses
org.keycloak.adapters.jaas.RolePrincipal,org.apache.karaf.jaas.boot.principal.RolePrincipal
restart io.hawt.hawtio-war
```

6. <http://localhost:8181/hawtio> にアクセスし、Red Hat Single Sign-On レルムからユーザーとしてログインします。

Hawtio に対して正常に認証するには、適切なレルムロールが必要です。利用可能なロールは、**hawtio.roles** の **\$FUSE_HOME/etc/system.properties** ファイルで設定されます。

2.1.6. Spring Boot アダプター

Spring Boot アプリケーションのセキュリティを保護するには、Keycloak Spring Boot アダプター JAR をアプリケーションに追加する必要があります。その後、通常の Spring Boot 設定 (**application.properties**) を使用して追加の設定を提供する必要があります。以下の手順を実施します。



注記

Spring Boot アダプターは非推奨となり、RH-SSO の 8.0 以降のバージョンには含まれません。このアダプターは、RH-SSO 7.x のライフサイクル期間、メンテナンスされます。ユーザーは Spring Security に移行して、Spring Boot アプリケーションを RH-SSO と統合する必要があります。

2.1.6.1. アダプターのインストール

Keycloak Spring Boot アダプターは Spring Boot の自動設定を活用するため、Keycloak Spring Boot スターターをプロジェクトに追加する必要があります。

Maven を使用して追加するには、以下を依存関係に追加します。

```
<dependency>
  <groupId>org.keycloak</groupId>
  <artifactId>keycloak-spring-boot-starter</artifactId>
</dependency>
```

アダプター BOM 依存関係を追加します。

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.keycloak.bom</groupId>
      <artifactId>keycloak-adapter-bom</artifactId>
      <version>9.0.17.redhat-00001</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

現在、以下の埋め込みコンテナはサポートされており、Starter の使用時には追加の依存関係は必要ありません。

- Tomcat

- Undertow
- Jetty

2.1.6.2. 必要な Spring Boot アダプター設定

本セクションでは、Keycloak を使用するように Spring Boot アプリケーションを設定する方法を説明します。

keycloak.json ファイルの代わりに、通常の Spring Boot 設定を介して Spring Boot アダプターのレルムを設定します。以下に例を示します。

```
keycloak.realm = demorealm
keycloak.auth-server-url = http://127.0.0.1:8080/auth
keycloak.ssl-required = external
keycloak.resource = demoapp
keycloak.credentials.secret = 11111111-1111-1111-1111-111111111111
keycloak.use-resource-role-mappings = true
```

keycloak.enabled = false を設定して、Keycloak Spring Boot Adapter (テストなど) を無効できます。

Policy Enforcer を設定するには、**keycloak.json** とは異なり、**policy-enforcer** ではなく **policy-enforcer-config** を使用する必要があります。

また、通常 **web.xml** にある Java EE セキュリティー設定を指定する必要があります。Spring Boot Adapter は **login-method** を **KEYCLOAK** に設定し、起動時に **security-constraints** を設定します。以下は、設定例です。

```
keycloak.securityConstraints[0].authRoles[0] = admin
keycloak.securityConstraints[0].authRoles[1] = user
keycloak.securityConstraints[0].securityCollections[0].name = insecure stuff
keycloak.securityConstraints[0].securityCollections[0].patterns[0] = /insecure

keycloak.securityConstraints[1].authRoles[0] = admin
keycloak.securityConstraints[1].securityCollections[0].name = admin stuff
keycloak.securityConstraints[1].securityCollections[0].patterns[0] = /admin
```



警告

Spring アプリケーションを WAR としてデプロイする予定の場合は、Spring Boot アダプターを使用せず、使用しているアプリケーションサーバーまたはサーブレットコンテナの専用アダプターを使用してください。Spring Boot には **web.xml** ファイルも含まれている必要があります。

2.1.7. Java Servlet Filter Adapter

Java Servlet アプリケーションをプラットフォームにデプロイする場合は、Red Hat Single Sign-On アダプターがないため、サーブレットフィルターアダプターを使用できます。このアダプターは、他のアダプターとは異なります。web.xml ではセキュリティ制約を定義しません。代わりに、Red Hat

Single Sign-On サブレットフィルターアダプターを使用してフィルターマッピングを定義し、セキュリティを保護する url パターンを保護します。



警告

Backchannel ログアウトの動作は、標準アダプターとは少々異なります。HTTP セッションを無効にする代わりに、セッション ID をログアウトとしてマークします。セッション ID に基づいて HTTP セッションを無効にする標準的な方法はありません。

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_3_0.xsd"
  version="3.0">

  <module-name>application</module-name>

  <filter>
    <filter-name>Keycloak Filter</filter-name>
    <filter-class>org.keycloak.adapters.servlet.KeycloakOIDCFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>Keycloak Filter</filter-name>
    <url-pattern>/keycloak/*</url-pattern>
    <url-pattern>/protected/*</url-pattern>
  </filter-mapping>
</web-app>
```

上記のスニペットには、url-patterns が 2 つあります。/protected/* は保護する必要があるファイルですが、/keycloak/* url-pattern は Red Hat Single Sign-On サーバーからコールバックを処理します。

構成された **url-patterns** の下の一部のパスを除外する必要がある場合は、フィルターの init パラメーター **keycloak.config.skipPattern** を使用して、keycloak フィルターがフィルターチェーンにすぐに委譲するパスパターンを記述する正規表現を構成できます。デフォルトでは skipPattern が設定されていません。

パターンは、**context-path** なしで **requestURI** に対して一致します。コンテキストパス /myapp を指定すると、/myapp/index.html のリクエストはスキップパターンに対して /index.html と照合されます。

```
<init-param>
  <param-name>keycloak.config.skipPattern</param-name>
  <param-value>^(path1|path2|path3).*</param-value>
</init-param>
```

フィルターの url-pattern で対応しているセキュアなセクションを参照する管理 URL を使用して、Red Hat Single Sign-On 管理コンソールでクライアントを設定する必要があります。

管理 URL は、バックチャネルログアウトなどを行うための管理 URL へのコールバックを作成します。この例の管理 URL は **http[s]://hostname/{context-root}/keycloak** となります。

Red Hat Single Sign-On フィルターには、他のアダプターと同じ設定パラメーターがあります。ただし、コンテキストパラメーターではなく、フィルター init パラメーターとして定義する必要があります。

このフィルターを使用するには、この maven アーティファクトを WAR pom に組み込みます。

```
<dependency>
  <groupId>org.keycloak</groupId>
  <artifactId>keycloak-servlet-filter-adapter</artifactId>
  <version>9.0.17.redhat-00001</version>
</dependency>
```

2.1.8. セキュリティコンテキスト

KeycloakSecurityContext インターフェースは、トークンに直接アクセスする必要がある場合に利用できます。これは、トークン (ユーザープロフィール情報など) から追加情報を取得する場合や、Red Hat Single Sign-On が保護する RESTful サービスを呼び出す場合に便利です。

サーブレット環境では、`HttpServletRequest` の属性としてセキュアな呼び出しを利用できます。

```
HttpServletRequest
  .getAttribute(KeycloakSecurityContext.class.getName());
```

または、`HttpSession` のセキュアでない要求で利用できます。

```
HttpServletRequest.getSession()
  .getAttribute(KeycloakSecurityContext.class.getName());
```

2.1.9. エラー処理

Red Hat Single Sign-On には、サーブレットベースのクライアントアダプターに対するエラー処理機能があります。認証でエラーが発生した場合、Red Hat Single Sign-On は

HttpServletResponse.sendError() を呼び出します。**web.xml** ファイル内にエラーページを設定して、必要なエラーを処理できます。Red Hat Single Sign-On は 400、401、403、および 500 のエラーをスローできます。

```
<error-page>
  <error-code>403</error-code>
  <location>/ErrorHandler</location>
</error-page>
```

Red Hat Single Sign-On は、取得可能な **HttpServletRequest** 属性も設定します。属性名は **org.keycloak.adapters.spi.AuthenticationError** で、**org.keycloak.adapters.OIDCAuthenticationError** にキャストする必要があります。

以下に例を示します。

```
import org.keycloak.adapters.OIDCAuthenticationError;
import org.keycloak.adapters.OIDCAuthenticationError.Reason;
...
```

```
OIDCAuthenticationError error = (OIDCAuthenticationError) httpServletRequest
  .getAttribute('org.keycloak.adapters.spi.AuthenticationError');
```

```
Reason reason = error.getReason();
System.out.println(reason.name());
```

2.1.10. ログアウト

Web アプリケーションからログアウトするには、複数の方法でログアウトできます。Java EE サーブレットコンテナでは、**HttpServletRequest.logout()** を呼び出すことができます。他のブラウザーアプリケーションの場合は、ブラウザーを **http://auth-server/auth/realms/{realm-name}/protocol/openid-connect/logout?redirect_uri=encodedRedirectUri** にリダイレクトできます。これは、ブラウザーに SSO セッションがある場合にログアウトします。

HttpServletRequest.logout() オプションを使用する場合、アダプターはリフレッシュトークンを渡す Red Hat Single Sign-On サーバーに対してバックチャンネル POST 呼び出しを実行します。保護されていないページ (有効なトークンを確認しないページ) からメソッドが実行されると、更新トークンは利用できません。その場合、アダプターは呼び出しをスキップします。このため、現在のトークンが常に考慮され、必要に応じて Red Hat Single Sign-On server サーバーとの対話が実行されるように、保護されたページを使用して **HttpServletRequest.logout()** を実行することが推奨されます。

ログアウト処理の一部として外部 ID プロバイダからのログアウトを回避したい場合は、パラメータ **initiating_idp** を指定して、値を問題の ID プロバイダーの ID (エイリアス) にすることができます。これは、ログアウトエンドポイントが外部アイデンティティプロバイダーによって開始される単一ログアウトの一部として呼び出される場合に便利です。

2.1.11. パラメーター転送

Red Hat Single Sign-On の初期承認エンドポイントリクエストでは、さまざまなパラメーターがサポートされます。ほとんどのパラメーターは、[OIDC 仕様](#) に記載されています。一部のパラメーターは、アダプター設定に基づいてアダプターによって自動的に追加されます。ただし、呼び出しごとに追加できるパラメーターはいくつかあります。セキュアなアプリケーション URI を開くと、特定のパラメーターは Red Hat Single Sign-On 承認エンドポイントに転送されます。

例えば、オフライントークンを要求する場合は、次のように **scope** パラメーターを指定してセキュアなアプリケーションの URI を開くことができます。

```
http://myappserver/mysecuredapp?scope=offline_access
```

およびパラメーター **scope=offline_access** を指定すると、Red Hat Single Sign-On 認証エンドポイントに自動的に転送されます。

サポートされるパラメーターは以下のとおりです。

- **scope** - スコープのスペースで区切られたリストを使用します。スペースで区切られたリストは、通常、特定のクライアントで定義された [クライアントスコープ](#) を参照します。スコープの **openid** は、アダプターによって常にスコープのリストに追加されることに注意してください。例えば、スコープオプションの **address phone** を入力すると、Red Hat Single Sign-On へのリクエストにはスコープパラメーター **scope=openid address phone** が含まれます。
- **prompt** - Red Hat Single Sign-On では、以下の設定がサポートされます。
 - **login** - SSO は無視され、ユーザーがすでに認証済みであっても Red Hat Single Sign-On ログインページが常に表示されます。
 - **consent** - 同意が必要なクライアントにのみ適用されます。これを使用すると、ユーザーが以前にこのクライアントに同意した場合でも、Consent ページが常に表示されます。

- **none** - ログインページは表示されません。その代わりに、ユーザはアプリケーションにリダイレクトされ、ユーザが認証されていない場合はエラーが表示されます。この設定により、アプリケーション側でフィルター/インターセプターを作成し、ユーザにカスタムのエラーページを表示できます。詳細は、仕様を参照してください。
- **max_age** - ユーザーがすでに認証されている場合にのみ使用します。認証が永続する最大時間を指定します。ユーザーの認証時から測定されます。ユーザーが **maxAge** より長く認証されると、SSO は無視され、再認証が必要となります。
- **login_hint** - ログインフォームの username/email フィールドを事前に入力するのに使用されます。
- **kc_idp_hint** - Red Hat Single Sign-On に対して、ログインページの表示をスキップし、指定のアイデンティティプロバイダーに自動的にリダイレクトするために使用されます。詳細は、[Identity Providerのドキュメント](#)を参照してください。

ほとんどのパラメーターは、[OIDC 仕様](#)に記載されています。唯一の例外は **kc_idp_hint** パラメーターで、これは Red Hat Single Sign-On に固有のもので、自動的に使用する ID プロバイダーの名前が含まれています。詳細は、『[サーバー管理ガイド](#)』の「**Identity Brokering**」のセクションを参照してください。



警告

割り当てられたパラメーターを使用して URL を開くと、アダプターはアプリケーションで認証されている場合に Red Hat Single Sign-On にリダイレクトしません。たとえば、`http://myappserver/mysecuredapp?prompt=login` を開いても、アプリケーション **mysecuredapp** に対して認証済みの場合は、Red Hat Single Sign-On のログインページに自動的にリダイレクトされません。この動作は今後変更される可能性があります。

2.1.12. クライアント認証

機密 OIDC クライアントがバックチャネル要求を送信する必要がある場合 (トークンのコードを交換したり、トークンを更新する場合など)、Red Hat Single Sign-On サーバーに対して認証する必要があります。デフォルトでは、クライアント ID およびクライアントシークレット、署名済み JWT によるクライアント認証、またはクライアントシークレットを使用した署名済み JWT によるクライアント認証の 3 つの方法があります。

2.1.12.1. クライアント ID およびクライアントシークレット

これは、OAuth2 仕様で説明されている従来の方法です。クライアントには、アダプター (アプリケーション) サーバーと Red Hat Single Sign-On サーバーの両方を認識する必要があるシークレットがあります。Red Hat Single Sign-On 管理コンソールで特定のクライアントのシークレットを生成し、このシークレットをアプリケーション側の **keycloak.json** ファイルに貼り付けることができます。

```
"credentials": {
  "secret": "19666a4f-32dd-4049-b082-684c74115f28"
}
```

2.1.12.2. 署名済み JWT によるクライアント認証

これは [RFC7523](#) の仕様に基づいています。これは、以下のように動作します。

- クライアントには、秘密鍵と証明書が必要です。Red Hat シングルサインオンでは、クライアントアプリケーションのクラスパスかファイルシステムのどこかにある従来の **keystore** ファイルから利用できます。
- クライアントアプリケーションを起動すると、がクライアントアプリケーションのベース URL であると仮定して、`http://myhost.com/myapp/k_jwks` のような URL を使用して **JWKS** 形式で公開鍵をダウンロードすることができます。この URL は Red Hat Single Sign-On で使用できます (下記参照)。
- 認証中、クライアントは JWT トークンを生成し、その秘密鍵で署名し、**client_assertion** パラメーターの特定のバックチャンネル要求 (たとえば、code-to-token 要求) で Red Hat Single Sign-On に送信します。
- Red Hat Single Sign-On には、JWT で署名を検証できるように、クライアントの公開鍵または証明書が必要です。Red Hat Single Sign-On では、クライアントの認証情報を設定する必要があります。まず、管理コンソールの **Credentials** タブで、クライアントの認証方法として「**Signed JWT**」を選択する必要があります。次に、以下のいずれかを選択できます。
 - Red Hat Single Sign-On がクライアントの公開鍵をダウンロードできる JWKS URL を設定します。これは、`http://myhost.com/myapp/k_jwks` などの URL を指定できます (詳細は上記を参照してください)。クライアントはキーをいつでもローテーションできるため、このオプションは最も柔軟性の高いものであり、Red Hat Single Sign-On は設定を変更しなくても、必要に応じていつでも新しいキーをダウンロードします。より正確には、Red Hat Single Sign-On は、未知の **kid** (キー ID) 署名したトークンを見つけると、新しいキーをダウンロードします。
 - クライアントの公開鍵または証明書を、PEM 形式、JWK 形式、またはキーストアからアップロードします。このオプションを使用すると、公開鍵はハードコーディングされ、クライアントが新しいキーペアを生成する際に変更する必要があります。自分自身が利用可能でない場合は、Red Hat Single Sign-On の管理コンソールから独自のキーストアを生成することもできます。Red Hat Single Sign-On 管理コンソールの設定方法の詳細は、「[サーバー管理ガイド](#)」を参照してください。

アダプター側で設定するには、**keycloak.json** ファイルに以下のようなものを記述する必要があります。

```
"credentials": {
  "jwt": {
    "client-keystore-file": "classpath:keystore-client.jks",
    "client-keystore-type": "JKS",
    "client-keystore-password": "storepass",
    "client-key-password": "keypass",
    "client-key-alias": "clientkey",
    "token-expiration": 10
  }
}
```

この設定では、WAR のクラスパスに **keystore-client.jks** というキーストアファイルがある必要があります。接頭辞 **classpath:** を使用しない場合は、クライアントアプリケーションが実行しているファイルシステム上の任意のファイルを指定できます。

2.1.13. マルチテナンシー

この場合、複数の Red Hat Single Sign-On レルムを使用して、1つのターゲットアプリケーション (WAR) をセキュアにできることを意味します。レルムは、同じ Red Hat Single Sign-On インスタンスまたは別のインスタンスに配置できます。

実際には、これはアプリケーションが複数の **keycloak.json** アダプター設定ファイルを持つ必要があることを意味します。

異なるアダプター構成ファイルが異なるコンテキストパスにデプロイされた WAR の複数のインスタンスを持つことができます。しかし、これは不便である可能性があります。また、context-path 以外の項目に基づいてレルムを選択することもできます。

Red Hat Single Sign-On では、カスタム設定リゾルバーを設定して、リクエストごとに使用するアダプター設定を選択できるようになります。

これを実現するには、まず **org.keycloak.adapters.KeycloakConfigResolver** の実装を作成する必要があります。以下に例を示します。

```
package example;

import org.keycloak.adapters.KeycloakConfigResolver;
import org.keycloak.adapters.KeycloakDeployment;
import org.keycloak.adapters.KeycloakDeploymentBuilder;

public class PathBasedKeycloakConfigResolver implements KeycloakConfigResolver {

    @Override
    public KeycloakDeployment resolve(OIDCHttpFacade.Request request) {
        if (path.startsWith("alternative")) {
            KeycloakDeployment deployment = cache.get(realm);
            if (null == deployment) {
                InputStream is = getClass().getResourceAsStream("/tenant1-keycloak.json");
                return KeycloakDeploymentBuilder.build(is);
            }
        } else {
            InputStream is = getClass().getResourceAsStream("/default-keycloak.json");
            return KeycloakDeploymentBuilder.build(is);
        }
    }
}
```

また、どの **KeycloakConfigResolver** 実装を **web.xml** のコンテキストパラメーター **keycloak.config.resolver** で使用するかを設定する必要があります。

```
<web-app>
...
<context-param>
    <param-name>keycloak.config.resolver</param-name>
    <param-value>example.PathBasedKeycloakConfigResolver</param-value>
</context-param>
</web-app>
```

2.1.14. アプリケーションクラスタリング

本章では、JBoss EAP にデプロイされたクラスター化されたアプリケーションのサポートに関連します。

アプリケーションが次のいずれであるかに応じて、いくつかのオプションを利用できます。

- ステートレスまたはステートフル
- 分散可能 (複製可能 http セッション) または非分散可能
- ロードバランサーが提供するスティッキーセッションへの依存
- Red Hat Single Sign-On と同じドメインでホストされる

クラスタリングの処理は、通常のアプリケーションほど単純ではありません。主に、ブラウザーとサーバー側のアプリケーションがリクエストを Red Hat Single Sign-On に送信するため、ロードバランサーでスティッキーセッションを有効にするのは単純ではありません。

2.1.14.1. ステートレストークンストア

デフォルトでは、Red Hat Single Sign-On によってセキュア化された web アプリケーションは HTTP セッションを使用してセキュリティコンテキストを保存します。つまり、スティッキーセッションを有効にするか、HTTP セッションを複製する必要があります。

HTTP セッションにセキュリティコンテキストを保存する代わりに、アダプターを設定して Cookie にこれを保存するように設定できます。これは、アプリケーションをステートレスにしたい場合、またはセキュリティコンテキストを HTTP セッションに保存したくない場合に役立ちます。

セキュリティコンテキストの保存に cookie ストアを使用するには、アプリケーションの **WEB-INF/keycloak.json** を編集して、以下を追加してください。

```
"token-store": "cookie"
```



注記

token-store のデフォルト値は **session** で、HTTP セッションにセキュリティコンテキストを保存します。

クッキーストアを使用する 1 つの制限は、すべての HTTP リクエストに対してセキュリティコンテキスト全体がクッキーに渡されることです。これにより、パフォーマンスに影響する可能性があります。

別の小さな制限については、Single-Sign Out のサポートは限定されています。アダプターにより KEYCLOAK_ADAPTER_STATE cookie が削除されるため、アプリケーション自体からの init servlet logout (HttpServletRequest.logout) がアプリケーション自体からも問題なく機能します。ただし、別のアプリケーションから初期化されたバックチャネルログアウトは、cookie ストアを使用して Red Hat Single Sign-On によって伝播されません。そのため、アクセストークンのタイムアウトに short 値を使用することが推奨されます (例: 1 分)。



注記

一部のロードバランサーでは、Amazon ALB などのスティッキーセッション cookie 名またはコンテンツの設定は許可されません。これらの場合は、**shouldAttachRoute** オプションを **false** に設定することが推奨されます。

2.1.14.2. 相対 URI の最適化

Red Hat Single Sign-On およびアプリケーションが同じドメイン (リバースプロキシまたはロードバランサー経由) でホストされるデプロイメントシナリオでは、クライアント設定で相対 URI オプションを使用すると便利です。

相対 URI を使用すると、URI は Red Hat Single Sign-On へのアクセスに使用する URL と相対的に解決されます。

例えば、アプリケーションの URL が **https://acme.org/myapp** で、Red Hat Single Sign-On の URL が **https://acme.org/auth** の場合は、**https://acme.org/myapp** 代わりに **redirect-uri /myapp** を使用できます。

2.1.14.3. 管理 URL の設定

特定のクライアントの管理者 URL は、Red Hat Single Sign-On の管理コンソールで設定できます。これは、Red Hat Single Sign-On サーバーが、ユーザーのログアウトや失効ポリシーのプッシュなどのさまざまなタスクのためにアプリケーションにバックエンドリクエストを送信するのに使用されます。

たとえば、バックチャネルログアウトの仕組みは以下のようになります。

1. ユーザーが1つのアプリケーションからログアウトリクエストを送信します。
2. アプリケーションはログアウトリクエストを Red Hat Single Sign-On に送信します。
3. Red Hat Single Sign-On サーバーは、ユーザーセッションを無効にします。
4. 次に、Red Hat Single Sign-On サーバーは、セッションに関連付けられた管理 URL を持つアプリケーションにバックチャネルリクエストを送信します。
5. アプリケーションがログアウトリクエストを受け取ると、対応する HTTP セッションが無効になります。

管理の URL に **\${application.session.host}** が含まれている場合は、HTTP セッションに関連するノードの URL に置き換えられます。

2.1.14.4. アプリケーションノードの登録

前のセクションでは、Red Hat Single Sign-On が特定の HTTP セッションに関連付けられたノードにログアウトリクエストを送信する方法を説明します。ただし、場合によっては、管理者は、管理タスクを1つだけでなく、登録されているすべてのクラスターノードに伝達したいことがあります。たとえば、新しい not before ポリシーをアプリケーションにプッシュしたり、アプリケーションからすべてのユーザーをログアウトしたりするには、以下を実行します。

この場合、Red Hat Single Sign-On はすべてのアプリケーションクラスターノードを認識して、イベントをすべてのノードに送信できるようにします。これを行うには、自動検出メカニズムをサポートします。

1. 新規アプリケーションノードがクラスターに参加すると、登録要求を Red Hat Single Sign-On サーバーに送信します。
2. 要求は、定期的な間隔で Red Hat Single Sign-On に再送される場合があります。
3. Red Hat Single Sign-On サーバーが、指定のタイムアウト内で再登録要求を受信しない場合は、自動的に特定のノードの登録を解除します。
4. このノードは、登録されていないリクエストを送信する際に、Red Hat Single Sign-On でも登録が解除されます。これは通常、ノードのシャットダウンまたはアプリケーションのアンデプロイメント中に行われます。これは、アンデプロイメントリスナーが呼び出されていない場合

に強制シャットダウンが適切に動作しないため、自動登録解除が必要になります。

一部のクラスターアプリケーションでのみ必要であるため、起動登録および定期的な再登録はデフォルトでは無効になっています。

この機能を有効にするには、アプリケーションの **WEB-INF/keycloak.json** ファイルを編集し、以下を追加します。

```
"register-node-at-startup": true,  
"register-node-period": 600,
```

これは、アダプターが起動時に登録要求を送信し、10 分ごとに再登録します。

Red Hat Single Sign-On 管理コンソールでは、ノード再登録タイムアウトの最大数を指定できます (アダプター設定の **register-node-period** よりも大きくする必要があります)。管理コンソールを使用してクラスターノードを手動で追加および削除することもできます。これは、自動登録機能に依存したくない場合、または自動登録解除機能を使用せずに古いアプリケーションノードを削除する場合に便利です。

2.1.14.5. 各リクエストでトークンを更新する

デフォルトでは、アプリケーションアダプターは、有効期限が切れている場合にのみアクセストークンを更新します。ただし、すべてのリクエストでトークンを更新するようにアダプターを設定することもできます。アプリケーションが Red Hat Single Sign-On サーバーにより多くのリクエストを送信するため、パフォーマンスに影響する可能性があります。

この機能を有効にするには、アプリケーションの **WEB-INF/keycloak.json** ファイルを編集し、以下を追加します。

```
"always-refresh-token": true
```



注記

これは、パフォーマンスに大きく影響することがあります。ポリシーの前にログアウトを伝播するのにバックチャネルメッセージに依存しない場合にのみ、この機能を有効にします。考慮すべきもう1つの点は、デフォルトではアクセストークンの有効期限が短いため、ログアウトが伝播されなくても、トークンはログアウトから数分以内に有効期限が切れることです。

2.2. JAVASCRIPT アダプター

Red Hat Single Sign-On には、クライアント側 JavaScript ライブラリーが同梱されており、これを使用して HTML5/JavaScript アプリケーションをセキュアにすることができます。JavaScript アダプターは、Cordova アプリケーションに対して組み込みサポートがあります。

ライブラリーは **/auth/js/keycloak.js** にある Red Hat Single Sign-On サーバーから直接取得でき、ZIP アーカイブとして配布されます。

ベストプラクティスは、サーバーのアップグレード時に自動的に更新されるため、Red Hat Single Sign-On Server から直接 JavaScript アダプターを読み込むことです。代わりにアダプターを Web アプリケーションにコピーする場合は、サーバーをアップグレードした後のみアダプターをアップグレードするようにしてください。

クライアント側のアプリケーションの使用に関する重要なことは、クライアントの認証情報をクライア

ント側のアプリケーションに保存する安全な方法がないため、クライアントがパブリッククライアントである必要があることです。これにより、クライアント用に構成したリダイレクト URI が正しく、可能な限り具体的であることを確認することが非常に重要になります。

JavaScript アダプターを使用するには、まず Red Hat Single Sign-On 管理コンソールでアプリケーションのクライアントを作成する必要があります。**Access Type** で **public** が選択されていることを確認してください。

また、**Valid Redirect URI** と **Web Origins** を設定する必要があります。そうしないと、セキュリティの脆弱性が生じる可能性があるため、できるだけ具体的にしてください。

クライアントを作成したら、インストール タブをクリックして、**Format Option** の **Keycloak OIDC JSON** を選択し、**Download** をクリックします。ダウンロードした **keycloak.json** ファイルは、HTML ページと同じ場所にある Web サーバーでホストされている必要があります。

または、設定ファイルを省略し、アダプターを手動で設定することもできます。

以下の例は、JavaScript アダプターを初期化する方法を示しています。

```
<head>
  <script src="keycloak.js"></script>
  <script>
    var keycloak = new Keycloak();
    keycloak.init().then(function(authenticated) {
      alert(authenticated ? 'authenticated' : 'not authenticated');
    }).catch(function() {
      alert('failed to initialize');
    });
  </script>
</head>
```

keycloak.json ファイルが別の場所にある場合は、それを指定することができます。

```
var keycloak = new Keycloak('http://localhost:8080/myapp/keycloak.json');
```

代わりに、必要な設定で JavaScript オブジェクトを渡すことができます。

```
var keycloak = new Keycloak({
  url: 'http://keycloak-server/auth',
  realm: 'myrealm',
  clientId: 'myapp'
});
```

デフォルトでは認証を行うには **login** 関数を呼び出す必要があります。ただし、アダプターが自動的に認証されるようにするオプションは2つあります。**login-required** または **check-sso** を **init** 関数に渡すことができます。ユーザーが Red Hat Single Sign-On にログインしている場合は、**login-required** がクライアントを認証します。そうでないと、ログインページが表示されます。**check-sso** は、ユーザーがログインしていない場合のみクライアントを認証します。ユーザーがログインしていない場合、ブラウザーはアプリケーションにリダイレクトされ、認証されていないままになります。

silent check-sso オプションを設定できます。この機能を有効にすると、ブラウザーは Red Hat Single Sign-On サーバーへのフルリダイレクトを行わずにアプリケーションに戻りますが、このアクションは非表示の **iframe** で実行されるため、アプリケーションリソースを読み込むだけで済みます。アプリ

が初期化されたときにブラウザーによって1回解析され、Red Hat Single Sign-On からアプリにリダイレクトされた後に再度解析されることはありません。これは、SPA (Single Page Applications) の場合において特に便利です。

silent check-sso を有効にするには、init メソッドで **silentCheckSsoRedirectUri** 属性を指定する必要があります。この URI は、アプリケーションで有効なエンドポイントである必要があります (そして、Red Hat Single Sign-On 管理コンソールでクライアントの有効なリダイレクトとして設定する必要があります)。

```
keycloak.init({
  onLoad: 'check-sso',
  silentCheckSsoRedirectUri: window.location.origin + '/silent-check-sso.html'
})
```

認証の状態が正常にチェックされ、Red Hat Single Sign-On サーバーからトークンを取得すると、サイレント check-sso リダイレクト URI のページが iframe に読み込まれます。受信したトークンをメインアプリケーションに送信する以外のタスクはなく、次のようになります。

```
<html>
<body>
  <script>
    parent.postMessage(location.href, location.origin)
  </script>
</body>
</html>
```

指定された場所にあるこのページは、アプリケーション自身が提供する必要があり、JavaScript アダプターの一部ではないことに注意してください。



警告

Chrome バージョン 80 (2020 年 2 月リリース) からは、Red Hat Single Sign-On 側で SSL/TLS 接続が設定されている場合のみ、サイレントの **check-sso** 機能が動作するようになります。

login-required を有効にするには、**onLoad** を **login-required** に設定し、init メソッドに渡します。

```
keycloak.init({
  onLoad: 'login-required'
})
```

ユーザーが認証された後、アプリケーションは認証ヘッダーにベアラートークンを含めることで、Red Hat Single Sign-On で保護された RESTful サービスへのリクエストを行うことができます。以下に例を示します。

```
var loadData = function () {
  document.getElementById('username').innerText = keycloak.subject;

  var url = 'http://localhost:8080/restful-service';
```

```

var req = new XMLHttpRequest();
req.open('GET', url, true);
req.setRequestHeader('Accept', 'application/json');
req.setRequestHeader('Authorization', 'Bearer ' + keycloak.token);

req.onreadystatechange = function () {
  if (req.readyState === 4) {
    if (req.status === 200) {
      alert('Success');
    } else if (req.status === 403) {
      alert('Forbidden');
    }
  }
}

req.send();
};

```

注意すべきことの1つは、アクセストークンの有効期限はデフォルトで短いため、リクエストを送信する前にアクセストークンの更新が必要になることが場合があることです。これは **updateToken** メソッドで行うことができます。**updateToken** メソッドは、トークンが正常に更新された場合にのみサービスを呼び出し、そうでない場合にはユーザーにエラーを表示することを容易にする Promise を返します。以下に例を示します。

```

keycloak.updateToken(30).then(function() {
  loadData();
}).catch(function() {
  alert('Failed to refresh token');
});

```

2.2.1. セッションステータスの iframe

デフォルトでは、JavaScript アダプターは、Single-Sign Out が発生したかどうかを検出するために使用される非表示の iframe を作成します。これにはネットワークトラフィックは必要ありません。代わりに、特別なステータスクッキーを確認してステータスを取得します。**init** メソッドに渡されるオプションで **checkLoginIframe: false** を設定することで、この機能を無効できます。

このクッキーを直接参照する必要はありません。この形式は変更でき、アプリケーションではなく、Red Hat Single Sign-On サーバーの URL に関連付けられている可能性があります。



警告

Chrome バージョン 80 以降 (2020 年 2 月にリリース)、ステータス iframe は、Red Hat Single Sign-On に設定された SSL / TLS 接続上で特別なクッキーのみを確認できます。非セキュアな接続を使用すると、iframe がステータスをチェックするたびに Red Hat Single Sign-On にリダイレクトされる可能性があります。iframe を無効にしたり、Red Hat のシングルサインオン側で [SSL/TLS を設定](#) することで、この動作を回避できます。

2.2.2. 暗黙的フローおよびハイブリッドフロー

デフォルトでは、JavaScript アダプターは [Authorization Code](#) フローを使用します。

このフローでは、Red Hat Single Sign-On サーバーは、認証トークンではなく、承認コードをアプリケーションに返します。JavaScript アダプターは、ブラウザーがアプリケーションにリダイレクトされた後に、アクセストークンとリフレッシュトークンのコードを交換します。

Red Hat Single Sign-On は、Red Hat Single Sign-On で認証に成功した直後にアクセストークンが送信される [Implicit](#) フローもサポートしています。これは、コードをトークンと交換する追加のリクエストがないため、標準フローよりもパフォーマンスが向上する可能性があります。アクセストークンの有効期限が切れると影響があります。

ただし、URL フラグメントでアクセストークンを送信すると、セキュリティ脆弱性が発生する可能性があります。たとえば、トークンは Web サーバーログやブラウザー履歴によってリークされる可能性があります。

暗黙のフローを有効にするには、Red Hat Single Sign-On 管理コンソールで、クライアントの **Implicit Flow Enabled** フラグを有効にする必要があります。また、`init` メソッドに `implicit` 値を持つパラメーター `flow` を渡す必要があります。

```
keycloak.init({  
  flow: 'implicit'  
})
```

注記 1 つは、アクセストークンのみが提供され、更新トークンはありません。そのため、アクセストークンの期限が切れると、アプリケーションが Red Hat Single Sign-On に再度リダイレクトして新しいアクセストークンを取得する必要があります。

Red Hat Single Sign-On は、[Hybrid](#) フローにも対応しています。

これは、クライアントが管理コンソールで **Standard Flow Enabled** フラグおよび **Implicit Flow Enabled** フラグの両方を有効にしている必要があります。次に、Red Hat Single Sign-On サーバーは、コードとトークンの両方をアプリケーションに送信します。アクセストークンは、コードのアクセスと更新トークンを交換して、すぐに使用できます。暗黙的なフローと同様に、アクセストークンがすぐに利用できるため、ハイブリッドフローはパフォーマンスに適しています。ただし、トークンは URL で依然として送信され、前述のセキュリティの脆弱性は引き続き適用されます。

Hybrid フローの利点の 1 つは、更新トークンがアプリケーションで利用できることです。

ハイブリッドフローの場合は、パラメーター `flow` の値 `hybrid` を `init` メソッドに渡す必要があります。

```
keycloak.init({  
  flow: 'hybrid'  
})
```

2.2.3. Cordova でのハイブリッドアプリケーション

Keycloak は、[Apache Cordova](#) で開発されたハイブリッドモバイルアプリをサポートしています。JavaScript アダプターには、`cordova` および `cordova-native` の 2 つのモードがあります。

デフォルトは `cordova` で、アダプターのタイプが設定されておらず、`window.cordova` が存在する場合は自動的に選択されます。ログインすると、ユーザーが Red Hat Single Sign-On を操作できる [InApp ブラウザー](#) が開き、その後 <http://localhost> リダイレクトしてアプリに戻ります。このため、管理コン

ソールのクライアント設定セクションでこの URL を有効な redirect-uri としてホワイトリスト化する必要があります。

このモードの設定は簡単ですが、いくつかの欠点もあります。

- InApp-Browser はアプリに組み込まれたブラウザーで、電話のデフォルトブラウザーではありません。したがって、設定が異なり、保存されている認証情報は利用できません。
- 特に複雑な内容をレンダリングする場合は、InApp-Browser も遅くなる可能性があります。
- このモードを使用する前に、アプリがログインページをレンダリングするブラウザーを完全に制御できるため、アプリがユーザーの資格情報にアクセスできる可能性があるなど、セキュリティ上の懸念事項を考慮する必要があります。そのため、信頼できないアプリの使用は許可しないでください。

この例のアプリ (<https://github.com/keycloak/keycloak/tree/master/examples/cordova>) で初めてみましょう。

代替モードの **cordova-native** は別のアプローチを取っています。システムのブラウザーを使用してログインページが開きます。ユーザーが認証されると、ブラウザーは特別な URL を使用してアプリにリダイレクトします。そこから、Red Hat Single Sign-On アダプターは、URL からコードまたはトークンを読み取り、ログインを終了できます。

init メソッドにアダプター型の **cordova-native** を渡すことで、ネイティブモードを有効できます。

```
keycloak.init({
  adapter: 'cordova-native'
})
```

このアダプターには、以下の2つのプラグインが必要です。

- **cordova-plugin-browsertab**: システムのブラウザーで Web ページを開くことができます。
- **cordova-plugin-deeplinks**: ブラウザーが特別な URL でアプリにリダイレクトできるようにします。

アプリへのリンクの技術情報は、各プラットフォームや特別な設定によって異なります。詳細は、[deeplinks プラグインのドキュメント](#) の Android と iOS のセクションを参照してください。

アプリを開くためのリンクにはさまざまな種類があります。カスタムスキーム (**myapp://login** または **android-app://com.example.myapp/https/example.com/login**) と **ユニバーサルリンク (iOS)/Deep Links (Android)** です。前者はセットアップが簡単で信頼性が高い傾向がありますが、後者は一意であり、ドメインの所有者のみが登録できるため、セキュリティが強化されます。カスタム URL は iOS で非推奨になりました。最高の信頼性を得るには、ユニバーサルリンクをカスタム URL リンクのあるフォールバックサイトと組み合わせて使用することをお勧めします。

さらに、Keycloak アダプターとの互換性を改善するには、以下の手順が推奨されます。

- iOS のユニバーサルリンクは、**response-mode** を **query** に設定すると、より確実に動作するようです。
- リダイレクト時に Android がアプリの新しいインスタンスを開かないようにするには、次のスニペットを **config.xml** に追加します。

```
<preference name="AndroidLaunchMode" value="singleTask" />
```

ネイティブモードの使い方を示すアプリの例

(<https://github.com/keycloak/keycloak/tree/master/examples/cordova-native>) があります。

2.2.4. 以前のブラウザー

JavaScript アダプターは、Base64 (`window.btoa` および `window.atob`)、HTML5 History API、およびオプシオンで Promise API に依存します。これらが利用できないブラウザーをサポートする必要がある場合 (例: IE9) は、polyfillers を追加する必要があります。

ポリフィルライブラリーの例:

- Base64 - <https://github.com/davidchambers/Base64.js>
- HTML5 History - <https://github.com/devote/HTML5-History-API>
- Promise - <https://github.com/stefanpenner/es6-promise>

2.2.5. JavaScript アダプターリファレンス

2.2.5.1. コンストラクター

```
new Keycloak();  
new Keycloak('http://localhost/keycloak.json');  
new Keycloak({ url: 'http://localhost/auth', realm: 'myrealm', clientId: 'myApp' });
```

2.2.5.2. プロパティ

authenticated

ユーザが認証されている場合は **true**、そうでない場合は **false** です。

token

サービスへのリクエストで **Authorization** ヘッダーで送ることができる base64 エンコードされたトークン。

tokenParsed

解析されたトークンを JavaScript オブジェクトとして実行します。

subject

ユーザー id

idToken

base64 でエンコードされた ID トークン。

idTokenParsed

解析された id トークンを JavaScript オブジェクトとして実行します。

realmAccess

トークンに関連付けられたレルムロール。

resourceAccess

トークンに関連付けられたリソースロール。

refreshToken

新しいトークンの取得に使用できる base64 でエンコードされた更新トークン。

refreshTokenParsed

JavaScript オブジェクトとして解析された更新トークン。

timeSkew

ブラウザー時間と Red Hat Single Sign-On サーバー間の推定時間 (秒単位)。この値は見積りませんが、トークンの有効期限が切れているかどうかを判断します。

responseMode

init で渡される応答モード (デフォルト値は fragment)。

flow

init に渡されるフロー。

adapter

リダイレクトする方法と、その他のブラウザー関連の機能がライブラリーによって処理される方法を上書きできます。利用可能なオプション:

- 「default」 - ライブラリーはリダイレクトにブラウザー api を使用します (これがデフォルトです)。
- 「cordova」 - ライブラリーは InAppBrowser cordova プラグインを使用して keycloak login/registration ページを読み込もうとします (これは、ライブラリーが cordova エコシステムで作業しているときに自動的に使用されます)。
- 「cordova-native」 - ライブラリーは、BrowserTabs cordova プラグインを使用して、電話のシステムブラウザーを使用してログインおよび登録ページを開くを試みます。これには、アプリにリダイレクトするための特別な設定が必要です ([「Cordova でのハイブリッドアプリケーション」](#) を参照)。
- custom - カスタムアダプターを実装することができます (高度なユースケースのみ)。

responseType

ログイン要求で Red Hat Single Sign-On に送信された応答タイプ。これは初期化時に使用されるフロー値に基づいて決定されますが、この値を設定すると上書きできます。

2.2.5.3. メソッド

2.2.5.3.1. init(options)

アダプターを初期化するために呼び出されます。

オプションはオブジェクトです。ここでは、以下のようになります。

- useNonce - 認証応答がリクエストと一致することを確認します (デフォルトは true)。
- onLoad - 読み込み時に実行するアクションを指定します。サポートされている値は、**login-required** または **check-sso** です。
- silentCheckSsoRedirectUri - onLoad が「check-sso」に設定されたかどうかをサイレント認証チェックにリダイレクト URI を設定します。
- token - トークンに初期値を設定します。
- refreshToken - 更新トークンの初期値を設定します。
- idToken - id トークンに初期値を設定します (トークンまたは refreshToken とともにのみ)。

- `timeSkew` - ローカルタイムと Red Hat Single Sign-On サーバー間のスキュー用に初期値を設定します (トークンと `refreshToken` の場合のみ)。
- `checkLoginIframe` - ログイン状態の監視を有効にするかどうかを設定します (既定値は **true**)。
- `checkLoginIframeInterval` - ログイン状態を確認する間隔を設定します (デフォルトは 5 秒です)。
- `responseMode` - ログイン要求時に OpenID Connect 応答モードを Red Hat Single Sign-On サーバーに送信します。有効な値は **query** または **fragment** です。デフォルト値は **fragment** で、認証に成功した後、OpenID Connect パラメーターを URL フラグメントに追加した JavaScript アプリケーションに Red Hat Single Sign-On がリダイレクトすることを意味します。これは一般的には、より安全で、**query** を介して推奨されています。
- `flow` - OpenID Connect フローを設定します。有効な値は、**standard**、**implicit**、または **hybrid** です。
- `enableLogging` - Keycloak からコンソールへのログメッセージを有効にします (デフォルトは **false** です)。
- `pkceMethod` - 使用する Proof Key Code Exchange (PKCE) のメソッド。この値を設定すると、PKCE メカニズムが有効になります。利用可能なオプション:
 - 「S256」 - SHA256 ベースの PKCE メソッド

初期化の完了時に解決する promise を返します。

2.2.5.3.2. login(options)

ログインフォームへのリダイレクト (オプションは `redirectUri` や `prompt` フィールドのある任意のオブジェクト) です。

オプションはオブジェクトです。ここでは、以下のようになります。

- `redirecturi` - ログイン後にリダイレクトする URI を指定します。
- `prompt` - このパラメーターは、Red Hat Single Sign-On サーバー側でログインフローを若干カスタマイズできるようにします。たとえば、値の **login** の場合、ログイン画面の表示を強制します。**prompt** パラメーターの詳細および使用できるすべての値は、「[パラメーター転送セッション](#)」を参照してください。
- `maxAge` - ユーザーがすでに認証されている場合にのみ使用します。ユーザーの認証が発生した時点の最大時間を指定します。ユーザーが **maxAge** よりも長い期間認証されている場合、SSO は無視されるため、再認証が必要になります。
- `loginHint` - ログインフォームの `username/email` フィールドを事前に入力するのに使用されます。
- `scope` - `scope` パラメーターを Red Hat Single Sign-On ログインエンドポイントに転送するのに使用します。スコープのスペースで区切られたリストを使用します。通常、これらは特定のクライアントで定義される「[クライアントスコープ](#)」を参照します。スコープの **openid** は、アダプターによって常にスコープのリストに追加されることに注意してください。例えば、スコープオプションの **address phone** を入力すると、Red Hat Single Sign-On へのリクエストにはスコープパラメーター **scope=openid address phone** が含まれます。

- idpHint - Red Hat Single Sign-On に対して、ログインページの表示を省略し、代わりに指定されたアイデンティティプロバイダーに自動的にリダイレクトするように指示するために使用されます。詳細は、[Identity Providerのドキュメント](#)を参照してください。
- action - 値が **register** の場合は登録ページにリダイレクトされますが、それ以外の場合はログインページにリダイレクトされます。
- locale - [OIDC 1.0 仕様のセクション 3.1.2.1](#) に従って、「ui_localesクエリーパラメーターを設定します。
- kcLocale - UI に必要な Keycloak ロケールを指定します。これは、Keycloak サーバーに cookie を設定し、ユーザーのプロファイルを新しい優先ロケールに更新するよう指示するロケールパラメーターとは異なります。
- cordovaOptions - Cordova in-app-browser (該当する場合) に渡される引数を指定します。**hidden** オプションおよび **location** オプションは、これらの引数の影響を受けません。利用可能なすべてのオプションは<https://cordova.apache.org/docs/en/latest/reference/cordova-plugin-inappbrowser/> で定義されています。使用例: { zoom: "no", hardwareback: "yes" };

2.2.5.3.3. createLoginUrl(options)

ログインフォームに URL を返します (オプションは redirectUri フィールドまたは prompt フィールドのある任意のオブジェクト)。

オプションはオブジェクトで、関数の **login** と同じオプションをサポートします。

2.2.5.3.4. logout(options)

ログアウトにリダイレクトされます。

オプションはオブジェクトです。ここでは、以下のようになります。

- redirectUri - ログアウト後にリダイレクトする URI を指定します。

2.2.5.3.5. createLogoutUrl(options)

ユーザーをログアウトするための URL を返します。

オプションはオブジェクトです。ここでは、以下のようになります。

- redirectUri - ログアウト後にリダイレクトする URI を指定します。

2.2.5.3.6. register(options)

登録フォームにリダイレクトされます。オプション action = 'register' を使用したログインのショートカット

オプションはログイン方法と同じですが、「action」は「register」に設定されています

2.2.5.3.7. createRegisterUrl(options)

登録ページへの URL を返します。オプション action = 'register' を持つ createLoginUrl のショートカット

オプションは createLoginUrl メソッドと同じですが、「action」は「register」に設定されています

2.2.5.3.8. accountManagement()

アカウント管理コンソールにリダイレクトされます。

2.2.5.3.9. createAccountUrl()

アカウント管理コンソールに URL を返します。

2.2.5.3.10. hasRealmRole(role)

トークンに指定のレルムロールがある場合は true を返します。

2.2.5.3.11. hasResourceRole(role, resource)

トークンにリソースの指定されたロールがある場合に true を返します (指定の clientId が使用されていない場合、リソースは任意です)。

2.2.5.3.12. loadUserProfile()

ユーザープロファイルを読み込みます。

プロファイルで解決する promise を返します。

以下に例を示します。

```
keycloak.loadUserProfile()
  .then(function(profile) {
    alert(JSON.stringify(profile, null, " "))
  }).catch(function() {
    alert('Failed to load user profile');
  });
```

2.2.5.3.13. isTokenExpired(minValidity)

トークンの有効期限が切れる前にトークンが minValidity 秒未満の場合は true を返します (minValidity は任意であり、指定されていない場合は 0 が使用されます)。

2.2.5.3.14. updateToken(minValidity)

トークンが minValidity 秒以内に期限切れになると (minValidity は任意で、指定されていない場合は 5 が使用されます)、トークンが更新されます。セッションステータスが iframe が有効になっている場合は、セッションステータスもチェックされます。

トークンが更新されているかどうかを示すブール値で解決する promise を返します。

以下に例を示します。

```
keycloak.updateToken(5)
  .then(function(refreshed) {
    if (refreshed) {
      alert('Token was successfully refreshed');
    } else {
      alert('Token is still valid');
    }
  });
```

```
}).catch(function() {
    alert('Failed to refresh the token, or the session has expired');
});
```

2.2.5.3.15. clearToken()

トークンを含む認証状態を消去します。これは、トークンの更新が失敗した場合など、セッションの期限が切れたことをアプリケーションが検出する場合に役立ちます。

これを呼び出すと、AuthLogout コールバックリスナーが呼び出されます。

2.2.5.4. コールバックイベント

アダプターは、特定のイベントの callback リスナーの設定をサポートします。

以下に例を示します。

```
keycloak.onAuthSuccess = function() { alert('authenticated'); }
```

利用可能なイベントは以下のとおりです。

- onReady(authenticated) - アダプターが初期化されると呼び出されます。
- onAuthSuccess - ユーザーが正常に認証されると呼び出しされます。
- onAuthError - 認証中にエラーが発生した場合に呼び出しされます。
- onAuthRefreshSuccess - トークンの更新時に呼び出しされます。
- onAuthRefreshError - トークンの更新中にエラーが発生した場合に呼び出します。
- onAuthLogout - ユーザーがログアウトした場合に呼び出されます (セッションステータス iframe が有効になっている場合、または Cordova モードの場合にのみ呼び出されます)。
- onTokenExpired - アクセストークンの期限が切れたときに呼び出しされます。更新トークンが利用可能な場合、トークンは updateToken で更新できます。利用できない場合 (つまり暗黙的なフローの場合) は、ログイン画面にリダイレクトして新しいアクセストークンを取得できます。

2.3. NODE.JS アダプター

Red Hat Single Sign-On は、サーバー側の JavaScript アプリケーションを保護するために [Connect](#) 上に構築された Node.js アダプターを提供します。この目的は、[Express.js](#) などのフレームワークと統合できる柔軟性があることでした。

Node.js アダプターを使用するには、まず Red Hat Single Sign-On 管理コンソールでアプリケーションのクライアントを作成する必要があります。アダプターは、パブリック、機密、およびベアラーのみのアクセスタイプをサポートします。どちらを選択しても、ユースケースのシナリオにより異なります。

クライアントが作成されたら、**Installation** タブをクリックして **Format Option** に **Red Hat Single Sign-On OIDC JSON** を選択してから **Download** をクリックします。ダウンロードした **keycloak.json** ファイルはプロジェクトの root ディレクトリーにあるはずです。

2.3.1. インストール

[Node.js](#) がすでにインストールされている場合は、アプリケーションのディレクトリを作成します。

```
mkdir myapp && cd myapp
```

npm init コマンドを使用して、アプリケーションの **package.json** を作成します。依存関係一覧に Red Hat Single Sign-On の接続アダプターを追加するようになりました。

```
"dependencies": {
  "keycloak-connect": "file:keycloak-connect-9.0.7-Final-redhat-00002.tgz"
}
```

2.3.2. 使用法

Keycloak クラスをインスタンス化します。

Keycloak クラスは、アプリケーションとの設定および統合の中心的な場所を提供します。最も単純な作成には引数は含まれません。

```
var session = require('express-session');
var Keycloak = require('keycloak-connect');

var memoryStore = new session.MemoryStore();
var keycloak = new Keycloak({ store: memoryStore });
```

デフォルトでは、これはアプリケーションの主な実行ファイルとともに **keycloak.json** という名前のファイルを見つけ、keycloak 固有の設定を初期化します (公開鍵、レルム名、さまざまな URL)。**keycloak.json** ファイルは Red Hat Single Sign-On 管理コンソールから取得されます。

この方法を使用したインスタンス化により、妥当なデフォルトがすべて使用されます。または、**keycloak.json** ファイルの代わりに、設定オブジェクトを指定することもできます。

```
let kcConfig = {
  clientId: 'myclient',
  bearerOnly: true,
  serverUrl: 'http://localhost:8080/auth',
  realm: 'myrealm',
  realmPublicKey: 'MIIBljANB...'
};

let keycloak = new Keycloak({ store: memoryStore }, kcConfig);
```

アプリケーションは、以下を使用して、ユーザーを優先しているアイデンティティプロバイダーにリダイレクトすることもできます。

```
let keycloak = new Keycloak({ store: memoryStore, idpHint: myIdP }, kcConfig);
```

Web セッションストアの設定

Web セッションを使用して認証のサーバー側の状態を管理する場合は、少なくとも **store** パラメーターで **Keycloak(...)** を初期化し、**express-session** が使用している実際のセッションストアを渡します。

```
var session = require('express-session');
var memoryStore = new session.MemoryStore();
```

```
var keycloak = new Keycloak({ store: memoryStore });
```

カスタムスコープの値の指定

デフォルトでは、スコープ値の **openid** はクエリーパラメーターとして Red Hat Single Sign-On のログイン URL に渡されますが、さらにカスタム値を追加することができます。

```
var keycloak = new Keycloak({ scope: 'offline_access' });
```

2.3.3. ミドルウェアのインストール

インスタンス化したら、ミドルウェアを接続対応のアプリケーションにインストールします。

```
var app = express();

app.use( keycloak.middleware() );
```

2.3.4. 認証の確認

リソースにアクセスする前にユーザーが認証されていることを確認するには、**keycloak.checkSso()** を使用します。これは、ユーザーがすでにログインしている場合にのみ認証されます。ユーザーがログインしていない場合、ブラウザーは最初のリクエストされた URL にリダイレクトされ、認証されていないままになります。

```
app.get( '/check-sso', keycloak.checkSso(), checkSsoHandler );
```

2.3.5. リソースの保護

簡易認証

リソースにアクセスする前にユーザーを認証する必要があるように強制するには、単に **keycloak.protect()** の非引数バージョンを使用します。

```
app.get( '/complain', keycloak.protect(), complaintHandler );
```

ロールベースの承認

現在のアプリケーションのアプリケーションロールでリソースのセキュリティを保護するには、以下を実行します。

```
app.get( '/special', keycloak.protect('special'), specialHandler );
```

別のアプリケーションのアプリケーションロールでリソースを保護するには、以下を行います。

```
app.get( '/extra-special', keycloak.protect('other-app:special'), extraSpecialHandler );
```

レルムロールでリソースをセキュアにするには、以下を実行します。

```
app.get( '/admin', keycloak.protect( 'realm:admin' ), adminHandler );
```

リソースベースの承認

リソースベースの認証を使用すると、Keycloak で定義された一連のポリシーに基づいて、リソース

とその特定のメソッド/アクション ** を保護できるため、アプリケーションからの認証を外部化できます。これは、リソースを保護するために使用できる **keycloak.enforcer** メソッドを公開することで実現されます。*

```
app.get('/apis/me', keycloak.enforcer('user:profile'), userProfileHandler);
```

keycloak.enforcer メソッドは、**response_mode** 設定オプションの値に応じて 2 つのモードで動作します。

```
app.get('/apis/me', keycloak.enforcer('user:profile', {response_mode: 'token'}), userProfileHandler);
```

response_mode が **token** に設定されている場合は、アプリケーションに送信されたベアラートークンで表されるサブジェクトの代わりにパーミッションがサーバーから取得されます。この場合、新しいアクセストークンは、サーバーによって付与されたアクセス許可を使用して、Keycloak により発行されます。サーバーが予想されるパーミッションを持つトークンに回答しなかった場合、要求は拒否されます。このモードを使用すると、以下のようにリクエストからトークンを取得できるはずです。

```
app.get('/apis/me', keycloak.enforcer('user:profile', {response_mode: 'token'}), function (req, res) {
  var token = req.kauth.grant.access_token.content;
  var permissions = token.authorization ? token.authorization.permissions : undefined;

  // show user profile
});
```

アプリケーションがセッションを使用していて、サーバーからの以前の決定をキャッシュし、更新トークンを自動的に処理する場合は、このモードが推奨されます。このモードは、クライアントとリソースサーバーとして動作するアプリケーションに特に便利です。

response_mode が **permissions** (デフォルトモード) に設定されている場合、サーバーは新しいアクセストークンを発行せずに付与されたパーミッションの一覧のみを返します。このメソッドは、新しいトークンを発行しないだけでなく、以下のように リクエスト を介してサーバーに付与されたパーミッションを公開します。

```
app.get('/apis/me', keycloak.enforcer('user:profile', {response_mode: 'token'}), function (req, res) {
  var permissions = req.permissions;

  // show user profile
});
```

使用中の **response_mode** に関係なく、**keycloak.enforcer** メソッドは、アプリケーションに送信されたベアラートークン内のパーミッションを確認します。ベアラートークンで予想される権限がすでに引き継がれている場合は、サーバーと対話して意思決定を取得する必要はありません。これは、クライアントが、保護されたリソースにアクセスする前に予想されるパーミッションでサーバーからアクセストークンを取得できる場合に特に便利です。そのため、増分認証などの Keycloak Authorization Services が提供する機能を使用し、**keycloak.enforcer** がリソースへのアクセスを強制している場合に、サーバーへの追加のリクエストを回避できます。

デフォルトでは、ポリシーエンフォースャーはアプリケーション (例: **keycloak.json**) に定義された **client_id** を使用して、Keycloak Authorization Services をサポートする Keycloak のクライアントを参照します。この場合、クライアントは実際にはリソースサーバーであることをパブリックにすることはできません。

アプリケーションがパブリッククライアント (フロントエンド) とリソースサーバー (バックエンド) の両方として機能している場合は、次の構成を使用して、適用するポリシーで Keycloak 内の別のクライアントを参照できます。

```
keycloak.enforcer('user:profile', {resource_server_id: 'my-apiserver'})
```

フロントエンドおよびバックエンドを表すために、Keycloak で個別のクライアントを使用することが推奨されます。

保護するアプリケーションが Keycloak 認証サービスで有効になり、**keycloak.json** でクライアント認証情報を定義している場合は、サーバーに追加の要求をプッシュして決定のためにポリシーで利用できるようにすることができます。そのため、プッシュする要求と共に JSON を返す 関数 を想定する **claims** 設定オプションを定義できます。

```
app.get('/protected/resource', keycloak.enforcer(['resource:view', 'resource:write'], {
  claims: function(request) {
    return {
      "http.uri": ["/protected/resource"],
      "user.agent": // get user agent from request
    }
  }
}), function (req, res) {
  // access granted
```

Keycloak を設定してアプリケーションリソースを保護する方法については、『[認証サービスガイド](#)』を参照してください。

高度な認証

URL 自体の一部に基づいてリソースを保護するには、各セクションにロールが存在することを前提としています。

```
function protectBySection(token, request) {
  return token.hasRole( request.params.section );
}

app.get(('/:section/:page', keycloak.protect( protectBySection ), sectionHandler );
```

2.3.6. 追加の URL

明示的なユーザーがトリガーされたログアウト

デフォルトでは、ミドルウェアは **/logout** への呼び出しをキャッチし、Red Hat Single Sign-On 中心のログアウトワークフローを介してユーザーを送信します。これは、**logout** 設定パラメーターを **middleware()** 呼び出しに指定することで変更できます。

```
app.use( keycloak.middleware( { logout: '/logoff' } ));
```

Red Hat Single Sign-On 管理者のコールバック

また、ミドルウェアは、Red Hat Single Sign-On コンソールのコールバックをサポートし、単一のセッションまたはすべてのセッションをログアウトします。デフォルトでは、これらの管理コールバックのタイプは / のルート URL に関連して行われますが、**middleware()** 呼び出しに **admin** パラメーターを指定して変更できます。

```
app.use( keycloak.middleware( { admin: '/callbacks' } ));
```

2.4. その他の OPENID CONNECT ライブラリー

Red Hat Single Sign-On は、通常、使用が簡単で、Red Hat Single Sign-On との統合を改善できるアダプターによって提供されます。ただし、プログラミング言語、フレームワーク、またはプラットフォームでアダプターが利用できない場合は、代わりに汎用 OpenID Connect Relying Party (RP) ライブラリーを使用することを選択できます。本章では、Red Hat Single Sign-On に固有の詳細を説明します。また、特定のプロトコルの詳細は含まれません。詳細は、「[OpenID Connect 仕様](#)」および「[OAuth2 仕様](#)」を参照してください。

2.4.1. エンドポイント

理解すべき最も重要なエンドポイントは、よく知られている 設定エンドポイントです。Red Hat Single Sign-On の OpenID Connect 実装に関連するエンドポイントおよびその他の設定オプションを一覧表示します。エンドポイントは次のとおりです。

```
/realms/{realm-name}/.well-known/openid-configuration
```

完全な URL を取得するには、Red Hat Single Sign-On のベース URL を追加し、**{realm-name}** をレルムの名前に置き換えます。以下に例を示します。

```
http://localhost:8080/auth/realms/master/.well-known/openid-configuration
```

一部の RP ライブラリーは、このエンドポイントから必要なすべてのエンドポイントを取得しますが、その他のライブラリーでは、エンドポイントを個別に一覧表示しないといけない場合があります。

2.4.1.1. 承認エンドポイント

```
/realms/{realm-name}/protocol/openid-connect/auth
```

承認エンドポイントはエンドユーザーの認証を実行します。これは、ユーザーエージェントをこのエンドポイントにリダイレクトすることで行います。

詳細は、OpenID Connect 仕様の「[認証エンドポイント](#)」セクションを参照してください。

2.4.1.2. トークンエンドポイント

```
/realms/{realm-name}/protocol/openid-connect/token
```

トークンエンドポイントは、トークンの取得に使用されます。トークンは、承認コードを調べるか、使用するフローに応じて認証情報を直接指定して取得できます。トークンエンドポイントは、有効期限が切れたときに新しいアクセストークンの取得にも使用されます。

詳細は、OpenID Connect 仕様の [トークンエンドポイント](#) セクションを参照してください。

2.4.1.3. userInfo エンドポイント

```
/realms/{realm-name}/protocol/openid-connect/userinfo
```

userinfo エンドポイントは、認証されたユーザーについての標準要求を返し、ベアラートークンによって保護されます。

詳細は、OpenID Connect 仕様の [userInfo エンドポイント](#) セクションを参照してください。

2.4.1.4. ログアウトエンドポイント

```
/realms/{realm-name}/protocol/openid-connect/logout
```

ログアウトエンドポイントは、認証されたユーザーをログアウトします。

ユーザーエージェントはエンドポイントにリダイレクトできます。この場合は、アクティブなユーザーセッションがログアウトされます。その後、ユーザーエージェントはアプリケーションにリダイレクトされます。

エンドポイントはアプリケーションによって直接呼び出すこともできます。このエンドポイントを直接呼び出すには、更新トークンとクライアントの認証に必要な認証情報を追加する必要があります。

2.4.1.5. 証明書エンドポイント

```
/realms/{realm-name}/protocol/openid-connect/certs
```

証明書エンドポイントは、レルムが有効にした公開鍵を返し、JSON Web Key (JWK) としてエンコードされます。レルム設定によっては、トークンの検証に1つ以上のキーが有効になります。詳細は、『[サーバー管理ガイド](#)』および「[JSON Web Key specification](#)」を参照してください。

2.4.1.6. イントロスペクションエンドポイント

```
/realms/{realm-name}/protocol/openid-connect/token/introspect
```

イントロスペクションエンドポイントは、トークンのアクティブな状態を取得するために使用されます。つまり、これを使用してアクセストークンを検証したり、更新したりできます。これは、機密クライアントでのみ呼び出すことができます。

このエンドポイントでの起動方法についての詳細は、「[OAuth 2.0 Token Introspection specification](#)」を参照してください。

2.4.1.7. 動的クライアント登録エンドポイント

```
/realms/{realm-name}/clients-registrations/openid-connect
```

動的クライアント登録エンドポイントは、クライアントを動的に登録するのに使用されます。

詳細は、「[クライアント登録](#)」および「[OpenID 接続動的クライアント登録仕様](#)」の章を参照してください。

2.4.1.8. トークン失効エンドポイント

```
/realms/{realm-name}/protocol/openid-connect/revoke
```

トークン失効エンドポイントは、トークンの取り消しに使用されます。このエンドポイントでは、更新トークンとアクセストークンの両方がサポートされます。

このエンドポイントで呼び出し方法の詳細は、「[OAuth 2.0 Token Revocation specification](#)」を参照してください。

2.4.2. アクセストークンの検証

Red Hat Single Sign-On が発行するアクセストークンを手動で検証する必要がある場合には、[イントロスペクションエンドポイント](#)を呼び出すことができます。この方法のマイナス面は、Red Hat Single

Sign-On サーバーへのネットワーク呼び出しを行う必要があることです。同時に実行される検証要求が多すぎると、これは遅くなり、サーバーに過負荷がかかる可能性があります。Red Hat Single Sign-On が発行するアクセストークンは、[JSON Web Signature \(JWS\)](#) を使用してデジタル署名およびエンコードされた [JSON Web Tokens \(JWT\)](#) です。この方法でエンコードされるため、発行したレلمの公開鍵を使用してアクセストークンをローカルで検証できます。レلمの公開鍵を検証コードでハードコードするか、JWS 内に埋め込まれたキー ID (KID) で [証明書エンドポイント](#) を使用して公開鍵を検索してキャッシュします。コーディングする言語に応じて、JWS の検証に役立つサードパーティのライブラリーが多数あります。

2.4.3. フロー

2.4.3.1. 承認コード

Authorization Code フローは、ユーザーエージェントを Red Hat Single Sign-On にリダイレクトします。Red Hat Single Sign-On で正常に認証されたら、承認コードが作成され、ユーザーエージェントはアプリケーションにリダイレクトされます。その後、アプリケーションは認証情報と共に承認コードを使用して、Red Hat Single Sign-On からアクセストークン、更新トークン、および ID トークンを取得します。

フローは Web アプリケーションにターゲットとして設定されていますが、ユーザーエージェントを組み込むことができるモバイルアプリケーションなど、ネイティブアプリケーションに推奨されます。

詳細は、OpenID Connect 仕様の [Authorization Code Flow](#) を参照してください。

2.4.3.2. 暗黙的

暗黙的フローリダイレクトは承認コードフローと同じような機能ですが、アクセストークンおよび ID トークンが返される承認コードを返す代わりに、このフローが返されます。これにより、追加の呼び出しでアクセストークンの承認コードを交換する必要がなくなります。ただし、更新トークンは含まれません。その結果、有効期限の長いアクセストークンを許可する必要があります。これは、これらを無効にするのが非常に難しいため、問題があります。または、最初のアクセストークンの期限が切れたら、新しいアクセストークンを取得するために新しいリダイレクトが必要になります。暗黙的フローは、アプリケーションがユーザーを認証し、ログアウト自体を処理する場合に便利です。

また、アクセストークンと承認コードの両方が返されるハイブリッドフローもあります。

注意すべき点の1つは、アクセストークンが Web サーバーのログやブラウザーの履歴から漏洩する可能性があるため、暗黙的なフローとハイブリッドフローの両方に潜在的なセキュリティリスクがあることです。これは、アクセストークンに短い有効期限を使用することでいくらか軽減されます。

詳細については、OpenID Connect 仕様の [「Implicit Flow」](#) を参照してください。

2.4.3.3. リソースオーナーパスワード認証情報

Red Hat Single Sign-On の Direct Grant (Direct Grant) と呼ばれるリソースオーナーパスワード認証情報を使用すると、ユーザー認証情報をトークンと交換できます。絶対に必要な場合を除き、このフローの使用は推奨されません。これは、従来のアプリケーションおよびコマンドラインインターフェースなどで役に立ちます。

このフローの使用には、次のようないくつかの制限があります。

- ユーザーの認証情報がアプリケーションに公開される
- アプリケーションにはログインページが必要です。

- アプリケーションは認証スキームを認識する必要があります。
- 認証フローへの変更にはアプリケーションへの変更が必要です。
- アイデンティティブローカーまたはソーシャルログインはサポートされません。
- フロー (ユーザー自己登録、必要なアクションなど) はサポートされません。

クライアントで Resource Owner Password Credentials の使用を許可するには、クライアントに **Direct Access Grants Enabled** オプションを有効にする必要があります。

このフローは OpenID Connect に含まれず、OAuth 2.0 仕様の一部です。

詳細は、OAuth 2.0 仕様の「[Resource Owner Password Credentials Grant](#)」の章を参照してください。

2.4.3.3.1. CURL の使用例

以下の例は、ユーザー名 **user** およびパスワード **password** を使用して、レルムの **master** ユーザーのアクセストークンを取得する方法を示しています。この例では、機密クライアント **myclient** を使用しています。

```
curl \
  -d "client_id=myclient" \
  -d "client_secret=40cc097b-2a57-4c17-b36a-8fdf3fc2d578" \
  -d "username=user" \
  -d "password=password" \
  -d "grant_type=password" \
  "http://localhost:8080/auth/realms/master/protocol/openid-connect/token"
```

2.4.3.4. クライアント認証情報

クライアント認証情報は、クライアント (アプリケーションおよびサービス) がユーザーに代わってではなく、自身に代わってアクセスを取得する場合に使用されます。たとえば、特定のユーザーではなく、一般的なシステムに変更を適用するバックグラウンドサービスなどに役立ちます。

Red Hat Single Sign-On は、クライアントが秘密または公開鍵/秘密鍵のいずれかを使用して認証するためのサポートを提供します。

このフローは OpenID Connect に含まれず、OAuth 2.0 仕様の一部です。

詳細は、OAuth 2.0 仕様の「[Client Credentials Grant](#)」の章を参照してください。

2.4.4. リダイレクト URI

リダイレクトベースのフローを使用する場合は、クライアントに有効なリダイレクト URI を使用することが重要です。リダイレクト URI は可能な限り具体的にする必要があります。これは特に、クライアント側の (パブリッククライアント) アプリケーションに適用されます。これを行わないと、以下が発生する可能性があります。

- オープンリダイレクト - これにより、攻撃者はドメインから来ているように見えるなりすましリンクを作成できます
- 不正なエントリ - ユーザーがすでに Red Hat Single Sign-On で認証されている場合、攻撃者は、ユーザーの知らないうちにユーザーをリダイレクトすることでアクセスを取得するようにリダイレクト URI が正しく構成されていないパブリッククライアントを使用できます。

Web アプリケーションで実稼働環境では常にすべてのリダイレクト URI に **https** を使用します。http へのリダイレクトを許可しないでください。

いくつかの特別なリダイレクト URI もあります。

http://localhost

このリダイレクト URI はネイティブアプリケーションに役立ち、ネイティブアプリケーションは認証コードの取得に使用できるランダムポートで Web サーバーを作成できます。このリダイレクト URI は任意のポートを許可します。

urn:ietf:wg:oauth:2.0:oob

クライアント (またはブラウザーが利用できない) で Web サーバーを起動できない場合は、特別な **urn:ietf:wg:oauth:2.0:oob** リダイレクト URI を使用できます。このリダイレクト URI を使用すると、Red Hat Single Sign-On は、タイトルとページ上のボックスにコードを含むページを表示します。アプリケーションは、ブラウザーのタイトルが変更されたことを検出するか、ユーザーがコードを手動でアプリケーションにコピーして貼り付けることができます。このリダイレクト URI を使用すると、ユーザーが別のデバイスを使用してアプリケーションに貼り付けるコードを取得することもできます。

第3章 SAML

本セクションでは、Red Hat Single Sign-On クライアントアダプターまたは汎用 SAML プロバイダーライブラリーを使用して SAML でアプリケーションおよびサービスのセキュリティーを保護する方法を説明します。

3.1. JAVA アダプター

Red Hat Single Sign-On には、Java アプリケーションのさまざまなアダプターがあります。正しいアダプターの選択は、ターゲットプラットフォームによって異なります。

3.1.1. 汎用アダプターの設定

Red Hat Single Sign-On がサポートする各 SAML クライアントアダプターは、単純な XML テキストファイルで設定できます。これは次のようになります。

```
<keycloak-saml-adapter xmlns="urn:keycloak:saml:adapter"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:keycloak:saml:adapter
https://www.keycloak.org/schema/keycloak_saml_adapter_1_10.xsd">
  <SP entityID="http://localhost:8081/sales-post-sig/"
    sslPolicy="EXTERNAL"
    nameIDPolicyFormat="urn:oasis:names:tc:SAML:1.1:nameid-format:unspecified"
    logoutPage="/logout.jsp"
    forceAuthentication="false"
    isPassive="false"
    turnOffChangeSessionIdOnLogin="false"
    autodetectBearerOnly="false">
    <Keys>
      <Key signing="true" >
        <KeyStore resource="/WEB-INF/keystore.jks" password="store123">
          <PrivateKey alias="http://localhost:8080/sales-post-sig/" password="test123"/>
          <Certificate alias="http://localhost:8080/sales-post-sig"/>
        </KeyStore>
      </Key>
    </Keys>
    <PrincipalNameMapping policy="FROM_NAME_ID"/>
    <RoleIdentifiers>
      <Attribute name="Role"/>
    </RoleIdentifiers>
    <RoleMappingsProvider id="properties-based-role-mapper">
      <Property name="properties.resource.location" value="/WEB-INF/role-mappings.properties"/>
    </RoleMappingsProvider>
    <IDP entityID="idp"
      signaturesRequired="true">
    <SingleSignOnService requestBinding="POST"
      bindingUrl="http://localhost:8081/auth/realms/demo/protocol/saml"
    />

    <SingleLogoutService
      requestBinding="POST"
      responseBinding="POST"
      postBindingUrl="http://localhost:8081/auth/realms/demo/protocol/saml"
      redirectBindingUrl="http://localhost:8081/auth/realms/demo/protocol/saml"
    />
  </SP>
</keycloak-saml-adapter>
```

```

    <Keys>
      <Key signing="true">
        <KeyStore resource="/WEB-INF/keystore.jks" password="store123">
          <Certificate alias="demo"/>
        </KeyStore>
      </Key>
    </Keys>
  </IDP>
</SP>
</keycloak-saml-adapter>

```

これらの設定スイッチの一部はアダプター固有で、一部の設定がすべてのアダプターで共通となる場合があります。Java アダプターでは、システムプロパティの置き換えとして `${...}` エンクロージャーを使用できます。たとえば `${jboss.server.config.dir}` になります。

3.1.1.1. SP 要素

SP 要素属性の説明は次のとおりです。

```

<SP entityID="sp"
  sslPolicy="ssl"
  nameIDPolicyFormat="format"
  forceAuthentication="true"
  isPassive="false"
  keepDOMAssertion="true"
  autodetectBearerOnly="false">
  ...
</SP>

```

entityID

これは、このクライアントの ID です。IdP では、クライアントが通信しているユーザーを判別するためにこの値が必要です。この設定は 必須 です。

sslPolicy

これは、アダプターが強制する SSL ポリシーです。有効な値は **ALL**、**EXTERNAL**、および **NONE** です。**ALL** の場合、すべてのリクエストは HTTPS 経由で受信する必要があります。**EXTERNAL** の場合、非プライベート IP アドレスのみが HTTPS 経由で有線を通過する必要があります。**NONE** の場合、HTTPS 経由のリクエストはありません。この設定は 任意 です。デフォルト値は **EXTERNAL** です。

nameIDPolicyFormat

SAML クライアントは、特定の NameID Subject 形式を要求できます。特定の形式が必要な場合は、この値を入力します。標準の SAML 形式の識別子である **urn:oasis:names:tc:SAML:2.0:nameid-format:transient** する必要があります。この設定は 任意 です。デフォルトでは、特別な形式は要求されません。

forceAuthentication

SAML クライアントは、ユーザーが IdP にすでにログインしていても、再認証することをリクエストできます。有効にするには、これを **true** に設定します。この設定は 任意 です。デフォルト値は **false** です。

isPassive

SAML クライアントは、IdP にログインしていない場合でも、ユーザーの認証が要求されないように要求できます。必要に応じてこれを **true** に設定します。**forceAuthentication** は逆の設定なので、一緒に使用しないでください。この設定は 任意 です。デフォルト値は **false** です。

turnOffChangeSessionIdOnLogin

セッション ID は、一部のプラットフォームで正常にログインしてセキュリティ攻撃ベクトルをプラグインするためにデフォルトで変更されます。これを無効にするには、これを **true** に変更します。オフにしないことが推奨されます。デフォルト値は **false** です。

autodetectBearerOnly

アプリケーションが Web アプリケーションと Web サービス (例: SOAP または REST) の両方に対応する場合は、**true** に設定する必要があります。Web アプリケーションの認証されていないユーザーを Keycloak ログインページにリダイレクトできますが、認証されていない SOAP または REST クライアントに HTTP **401** ステータスコードを送信することができます。ログインページへのリダイレクトは理解できません。Keycloak は、**X-Requested-With**、**SOAPAction**、**Accept** などの一般的なヘッダーに基づいて SOAP クライアントまたは REST クライアントを自動検出します。デフォルト値は **false** です。

logoutPage

これにより、ログアウト後に表示するページが設定されます。ページが <http://web.example.com/logout.html> などの完全な URL の場合、ユーザーは HTTP **302** ステータスコードを使用してログアウト後にそのページにリダイレクトされます。スキーム部分のないリンク (例: **/logout.jsp**) が指定されると、**web.xml** の **security-constraint** 宣言に従って保護領域に存在するかどうかに関係なく、ログアウトの後にページが表示され、ページがデプロイメントコンテキストルートとの関連で解決されます。

keepDOMAssertion

この属性は **true** に設定して、要求に関連付けられた **SamlPrincipal** 内の元の形式でアダプタストアにアサーションの DOM 表現を設定します。アサーションドキュメントは、プリンシパル内の **getAssertionDocument** メソッドを使用して取得できます。これは、署名済みアサーションの再生時に特に便利です。返されるドキュメントは、Red Hat Single Sign-On サーバーが受信した SAML 応答を解析して生成されたドキュメントです。この設定は 任意 で、デフォルト値は **false** です (ドキュメントはプリンシパルに保存されません)。

3.1.1.2. サービスプロバイダーキーのキー要素

IdP で、クライアントアプリケーション (または SP) がすべての要求に署名するか、IdP がアサーションを暗号化する場合は、これを実行するために使用するキーを定義する必要があります。クライアント署名のドキュメントでは、ドキュメントの署名に使用される秘密鍵、および公開鍵または証明書の両方を定義する必要があります。暗号化の場合は、復号に使用する秘密鍵のみを定義する必要があります。

キーを記述する方法は 2 つあります。キーは Java KeyStore 内に格納することも、PEM 形式の **keycloak-saml.xml** にキーを直接コピー/貼り付けることもできます。

```
<Keys>
  <Key signing="true" >
    ...
  </Key>
</Keys>
```

Key 要素には、2 つの任意の属性の **signing** および **encryption** があります。true に設定すると、これらはアダプターにキーの使用目的を通知します。両方の属性が true に設定されていると、キーはドキュメントの署名と暗号化されたアサーションの復号の両方に使用されます。少なくとも 1 つの属性を true に設定する必要があります。

3.1.1.2.1. キーストア要素

Key 要素内で、Java Keystore から鍵と証明書を読み込みます。これは **KeyStore** 要素内で宣言されます。

```
<Keys>
```

```

    <Key signing="true" >
      <KeyStore resource="/WEB-INF/keystore.jks" password="store123">
        <PrivateKey alias="myPrivate" password="test123"/>
        <Certificate alias="myCertAlias"/>
      </KeyStore>
    </Key>
  </Keys>

```

以下は、**KeyStore** 要素で定義されている XML 設定属性です。

file

キーストアへのファイルパス。このオプションは 任意 です。file または resource 属性を設定する必要があります。

resource

KeyStore への WAR リソースパス。これは、ServletContext.getResourceAsStream() へのメソッド呼び出しで使用されるパスです。このオプションは 任意 です。file または resource 属性を設定する必要があります。

password

KeyStore のパスワード。このオプションは 必須 です。

SP がドキュメントの署名に使用するキーを定義する場合は、Java KeyStore 内の秘密鍵および証明書への参照も指定する必要があります。上記の例の **PrivateKey** および **Certificate** 要素は、キーストア内のキーまたは証明書を参照するエイリアスを定義します。キーストアには、秘密鍵にアクセスするために追加のパスワードが必要です。**PrivateKey** 要素で、**password** 属性内にこのパスワードを定義する必要があります。

3.1.1.2.2. キーの PEMS

Key 要素内で、**PrivateKeyPem**、**PublicKeyPem** および **CertificatePem** のサブ要素を使用して、キーと証明書を直接宣言します。これらの要素に含まれる値は PEM キー形式に準拠する必要があります。通常、**openssl** または同様のコマンドラインツールを使用してキーを生成する場合は、このオプションを使用します。

```

<Keys>
  <Key signing="true">
    <PrivateKeyPem>
      2341251234AB31234==231BB998311222423522334
    </PrivateKeyPem>
    <CertificatePem>
      211111341251234AB31234==231BB998311222423522334
    </CertificatePem>
  </Key>
</Keys>

```

3.1.1.3. SP PrincipalNameMapping 要素

この要素は任意です。**HttpServletRequest.getUserPrincipal()** などのメソッドから取得する Java Principal オブジェクトを作成する場合は、**Principal.getName()** メソッドによって返される名前を定義できます。

```

<SP ...>
  <PrincipalNameMapping policy="FROM_NAME_ID"/>
</SP>

```

```
<SP ...>
  <PrincipalNameMapping policy="FROM_ATTRIBUTE" attribute="email" />
</SP>
```

policy 属性は、この値に設定するために使用されるポリシーを定義します。この属性に使用できる値は次のとおりです。

FROM_NAME_ID

このポリシーは SAML サブジェクト値を使用するだけです。これはデフォルト設定です。

FROM_ATTRIBUTE

これにより、サーバーから受信した SAML アサーションで宣言された属性のいずれかから値をプルします。XML 属性 **attribute** 内で使用する SAML の assertion 属性の名前を指定する必要があります。

3.1.1.4. RoleIdentifiers 要素

RoleIdentifiers 要素は、ユーザーから受け取ったアサーション内の SAML 属性を、ユーザーの Java EE Security Context 内のロール識別子として使用する必要があります。

```
<RoleIdentifiers>
  <Attribute name="Role"/>
  <Attribute name="member"/>
  <Attribute name="memberOf"/>
</RoleIdentifiers>
```

デフォルトでは、**Role** 属性値は Java EE ロールに変換されます。一部の IdP は、**member** または **memberOf** 属性アサーションを使用してロールを送信します。1つ以上の **Attribute** 要素を定義して、ロールに変換する必要がある SAML 属性を指定できます。

3.1.1.5. RoleMappingsProvider 要素

RoleMappingsProvider は、SAML アダプターによって使用される SPI 実装 **org.keycloak.adapters.saml.RoleMappingsProvider** の ID および設定を許可するオプションの要素です。

Red Hat Single Sign-On が IDP として使用されると、ビルドされたロールマッパーを使用して、ロールを SAML アサーションに追加する前にマッピングできます。ただし、SAML アダプターを使用して、SAML リクエストをサードパーティの IDP に送信する際に使用できます。この場合は、SP の必要に応じてアサーションから展開されたロールを異なるロールセットにマッピングすることが必要になる場合があります。**RoleMappingsProvider** SPI は、必要なマッピングを実行するのに使用できるプラグ可能なロールマッパーの設定を可能にします。

プロバイダーの設定は以下のようになります。

```
...
<RoleIdentifiers>
  ...
</RoleIdentifiers>
<RoleMappingsProvider id="properties-based-role-mapper">
  <Property name="properties.resource.location" value="/WEB-INF/role-mappings.properties"/>
</RoleMappingsProvider>
```

```
<IDP>
```

```
...
```

```
</IDP>
```

id 属性は、使用するインストール済みプロバイダーを特定します。**Property** サブ要素は複数回使用してプロバイダーの設定プロパティを指定できます。

3.1.1.5.1. プロパティベースのロールマッピングプロバイダー

Red Hat Single Sign-On には、**properties** ファイルを使用してロールマッピングを実行する **RoleMappingsProvider** 実装が含まれています。このプロバイダーは **id properties-based-role-mapper** によって識別され、**org.keycloak.adapters.saml.PropertiesBasedRoleMapper** クラスによって実装されます。

このプロバイダーは、使用される **properties** ファイルの場所を指定するために使用できる 2 つの設定プロパティに依存します。最初に、設定値を使用して、ファイルシステムで **properties** ファイルを見つけることで、**properties.file.location** プロパティが指定されているかどうかを確認します。設定したファイルが見つからない場合、プロバイダーは **RuntimeException** を出力します。以下のスニペットは、**properties.file.configuration** オプションを使用して、ファイルシステムの **/opt/mappers/** ディレクトリーから **roles.properties** ファイルを読み込むプロバイダーの例を示しています。

```
<RoleMappingsProvider id="properties-based-role-mapper">
  <Property name="properties.file.location" value="/opt/mappers/roles.properties"/>
</RoleMappingsProvider>
```

properties.file.location 設定が設定されていない場合、プロバイダーは **properties.resource.location** プロパティをチェックし、設定した値を使用して **WAR** リソースから **properties** ファイルを読み込みます。この設定プロパティがない場合、プロバイダーはデフォルトで **/WEB-INF/role-mappings.properties** からファイルを読み込もうとします。リソースからファイルを読み込めない場合、プロバイダーは **RuntimeException** を出力します。以下のスニペットは、**properties.resource.location** を使用してアプリケーションの **/WEB-INF/conf/** ディレクトリーから **roles.properties** ファイルを読み込むプロバイダーの例を示しています。

```
<RoleMappingsProvider id="properties-based-role-mapper">
  <Property name="properties.resource.location" value="/WEB-INF/conf/roles.properties"/>
</RoleMappingsProvider>
```

properties ファイルには、ロールとプリンシパルの両方をキーとして含めることができ、0 つ以上のロールを値としてコンマで区切ります。呼び出されると、実装はアサーションから抽出したロールのセットを繰り返し処理し、各ロールについて、マッピングが存在するかどうかを確認します。ロールが空ロールにマップする場合、これは破棄されます。1 つ以上の異なるロールのセットにマップすると、これらのロールは結果セットに設定されます。ロールのマッピングが見つからない場合は、結果セットに組み込まれます。

ロールの処理後、アサーションから抽出したプリンシパルにエントリーの **properties** ファイルが含まれるかどうかを確認します。プリンシパルのマッピングが存在する場合は、値として一覧表示されるすべてのロールが結果セットに追加されます。これにより、追加のロールをプリンシパルに割り当てることができます。

たとえば、プロバイダーが以下のプロパティファイルで設定されていると仮定します。

```
roleA=roleX,roleY
roleB=

kc_user=roleZ
```

プリンシパル **kc_user** が、**roleA**、**roleB**、および **roleC** があるアサーションから抽出されると、プリンシパルに割り当てられたロールの最終セットは **roleC**、**roleX**、**roleY**、および **roleZ** になります。なぜなら **roleA** が **roleX** および **roleY** にマッピングされ、**roleB** が空のロールにマッピングされて破棄され、**roleC** がそのまま使用され、追加ロールが **kc_user** プリンシパル (**roleZ**) に追加されるためです。

3.1.1.5.2. 独自のロールマッピングプロバイダーの追加

カスタムロールマッピングプロバイダーを追加するには、単に **org.keycloak.adapters.saml.RoleMappingsProvider** SPI を実装する必要があります。詳細は、『[Server Developer Guide](#)』の「**SAML ロールマッピング SPI**」を参照してください。

3.1.1.6. IDP 要素

IDP 要素のすべては、SP が通信しているアイデンティティプロバイダー (認証トークン) の設定を記述します。

```
<IDP entityID="idp"
  signaturesRequired="true"
  signatureAlgorithm="RSA_SHA1"
  signatureCanonicalizationMethod="http://www.w3.org/2001/10/xml-exc-c14n#">
...
</IDP>
```

以下は、**IDP** 要素宣言内で指定できる属性設定オプションです。

entityID

これは IDP の発行者 ID です。この設定は 必須 です。

signaturesRequired

true に設定すると、クライアントアダプターは IDP に送信するすべてのドキュメントに署名します。また、クライアントは IDP が送信されたドキュメントに署名されることを想定します。このスイッチは、すべての要求および応答タイプのデフォルトを設定しますが、後でこのタイプを詳細に制御できることがわかります。この設定は 任意 で、デフォルトは **false** に設定されます。

signatureAlgorithm

これは、IDP により署名されたドキュメントで使用する必要がある署名アルゴリズムです。使用できる値は、**RSA_SHA1**、**RSA_SHA256**、**RSA_SHA512**、および **DSA_SHA1** です。この設定は 任意 で、デフォルト値は **RSA_SHA256** です。

signatureCanonicalizationMethod

これは、IDP により署名されたドキュメントで使用する必要がある署名の正規化方法です。この設定は 任意 です。デフォルト値は **http://www.w3.org/2001/10/xml-exc-c14n#** で、大抵の IDP に適しています。

metadataUrl

IDP メタデータの取得に使用される URL。現在、これは署名キーと暗号化キーを定期的を取得するためにのみ使用され、SP 側で手動で変更することなく IDP でこれらのキーを循環させることができます。

3.1.1.7. IDP AllowedClockSkew サブ要素

AllowedClockSkew オプションの sub 要素は、IDP と SP の間で許可されるクロックの skew を定義します。デフォルト値は 0 です。

```
<AllowedClockSkew unit="MILLISECONDS">3500</AllowedClockSkew>
```


unit

この要素の値に割り当てられた時間単位を定義することができます。使用できる値は MICROSECONDS、MILLISECONDS、MINUTES、NANOSECONDS、および SECONDS です。これは 任意 です。デフォルト値は **SECONDS** です。

3.1.1.8. IDP SingleSignOnService サブ要素

サブ要素 **SingleSignOnService** は、IDP のログイン SAML エンドポイントを定義します。クライアントアダプターは、ログイン時にこの要素内の設定を介してフォーマットされた IDP に要求を送信します。

```
<SingleSignOnService signRequest="true"
  validateResponseSignature="true"
  requestBinding="post"
  bindingUrl="url"/>
```

この要素で定義できる config 属性を以下に示します。

signRequest

クライアントは認証要求に署名する必要がありますか?この設定は 任意 です。デフォルトは、任意の IDP の **signaturesRequired** 要素の値に設定されます。

validateResponseSignature

クライアントは、IDP が auhtn 要求から返送されたアサーション応答ドキュメントに署名することを期待する必要がありますか?この設定は 任意 です。デフォルトは、任意の IDP の **signaturesRequired** 要素の値に設定されます。

requestBinding

これは IDP との通信に使用される SAML バインディングタイプです。この設定は 任意 です。デフォルト値は **POST** ですが、**REDIRECT** に設定することもできます。

responseBinding

SAML を使用すると、クライアントは、認証応答で使用するバインディングタイプを要求できません。値は **POST** または **REDIRECT** です。この設定は 任意 です。デフォルトでは、クライアントは応答用に特定のバインディングタイプを要求しません。

assertionConsumerServiceUrl

IDP ログインサービスが応答を送信する必要がある Assertion Consumer Service (ACS) の URL。この設定は 任意 です。デフォルトでは、IdP の設定に依存するため、これは未設定です。設定した場合は、/saml (例: <http://sp.domain.com/my/endpoint/for/saml>) で終了する必要があります。このプロパティの値は SAML **AuthnRequest** メッセージの **AssertionConsumerServiceURL** 属性で送信されます。通常、このプロパティには **responseBinding** 属性が伴います。

bindingUrl

これは、クライアントが要求を送信する IDP ログインサービスの URL です。この設定は 必須 です。

3.1.1.9. IDP SingleLogoutService サブ要素

サブ要素 **SingleLogoutService** は、IDP のログアウト SAML エンドポイントを定義します。クライアントアダプターは、ログアウト時にこの要素内の設定を介してフォーマットされた IDP に要求を送信します。

```
<SingleLogoutService validateRequestSignature="true"
  validateResponseSignature="true"
  signRequest="true"
```

```
signResponse="true"
requestBinding="redirect"
responseBinding="post"
postBindingUrl="posturl"
redirectBindingUrl="redirecturl">
```

signRequest

クライアントは、IDP に対して行うログアウト要求に署名する必要がありますか?この設定は 任意 です。デフォルトは、任意の IDP の **signaturesRequired** 要素の値に設定されます。

signResponse

クライアントが、IDP 要求に送信するログアウト応答に署名する必要があるかどうか。この設定は 任意 です。デフォルトは、任意の IDP の **signaturesRequired** 要素の値に設定されます。

validateRequestSignature

クライアントが、IDP からの署名済みログアウト要求ドキュメントを期待する必要があるかどうか。この設定は 任意 です。デフォルトは、任意の IDP の **signaturesRequired** 要素の値に設定されます。

validateResponseSignature

クライアントが、IDPからの署名済みログアウト応答ドキュメントを期待する必要があるか。この設定は 任意 です。デフォルトは、任意の IDP の **signaturesRequired** 要素の値に設定されます。

requestBinding

これは、IDP への SAML 要求の通信に使用される SAML バインディングタイプです。この設定は 任意 です。デフォルト値は **POST** ですが、**REDIRECT** に設定することもできます。

responseBinding

これは、IDP への SAML 応答の通信に使用される SAML バインディングタイプです。値は **POST** または **REDIRECT** です。この設定は 任意 です。デフォルト値は **POST** ですが、**REDIRECT** に設定することもできます。

postBindingUrl

これは、POST バインディングの使用時に IDP のログアウトサービスの URL です。この設定は、**POST** バインディングを使用する場合に 必須 になります。

redirectBindingUrl

これは、REDIRECT バインディングの使用時に IDP のログアウトサービスの URL です。この設定は、REDIRECT バインディングを使用する場合に 必須 になります。

3.1.1.10. IDP キーサブ要素

IDP のキーサブ要素は、IDP によって署名されたドキュメントの検証に使用する証明書またはパブリックキーの定義にのみ使用されます。これは、[SP の Keys 要素](#) と同じ方法で定義されます。ただし、証明書または公開鍵参照を定義するだけで十分です。IDP と SP の両方が Red Hat Single Sign-On サーバーおよびアダプターによって実現される場合、署名の検証にキーを指定する必要はありません。以下を参照してください。

SP と IDP の両方が Red Hat Single Sign-On によって実装されている場合、公開された証明書から IDP 署名検証用の公開鍵を自動的に取得するように SP を設定することができます。これは、キーサブ要素の署名検証キーの宣言をすべて削除することによって行われます。キーサブ要素が空のままになる場合は、完全に省略できます。次に、キーは SP により SAML 記述子から自動的に取得されます。これは、[「IDP SingleSignOnService サブ要素」](#) で指定された SAML エンドポイント URL から派生します。SAML 記述子取得に使用される HTTP クライアントの設定は、通常、追加設定は必要ありませんが、[IDP HttpClient サブ要素](#) で設定することもできます。

署名の検証に複数のキーを指定することもできます。これは、**signing** 属性が **true** に設定されている Keys サブ要素内で複数の Key 要素を宣言することによって行われます。これは、IDP 署名鍵がロー

テートされる場合などに便利です。通常、新しい SAML プロトコルメッセージとアサーションが新しいキーで署名される際に移行期間がありますが、以前のキーで署名されたものは引き続き受け入れる必要があります。

署名検証用のキーの自動取得および追加の静的署名検証キーの定義の両方を行うように Red Hat Single Sign-On を設定することはできません。

```
<IDP entityID="idp">
  ...
  <Keys>
    <Key signing="true">
      <KeyStore resource="/WEB-INF/keystore.jks" password="store123">
        <Certificate alias="demo"/>
      </KeyStore>
    </Key>
  </Keys>
</IDP>
```

3.1.1.11. IDP HttpClient サブ要素

HttpClient オプションのサブ要素は、[enabled](#) の場合に、IDP の SAML 記述子で IDP 署名検証用の公開鍵を含む証明書を自動的に取得するために使用される HTTP クライアントのプロパティを定義します。

```
<HttpClient connectionPoolSize="10"
  disableTrustManager="false"
  allowAnyHostname="false"
  clientKeystore="classpath:keystore.jks"
  clientKeystorePassword="pwd"
  truststore="classpath:truststore.jks"
  truststorePassword="pwd"
  proxyUrl="http://proxy/" />
```

connectionPoolSize

この設定オプションでは、Red Hat Single Sign-On サーバーへプールする接続の数を定義します。これは 任意 です。デフォルト値は **10** です。

disableTrustManager

Red Hat Single Sign-On サーバーに HTTPS が必要で、この設定オプションが **true** に設定されている場合は、トラストストアを指定する必要はありません。この設定は開発時のみ使用してください。これは SSL 証明書の検証を無効にするため、実稼働環境では 使用しないでください。これは 任意 です。デフォルト値は **false** です。

allowAnyHostname

Red Hat Single Sign-On サーバーに HTTPS が必要で、この設定オプションが **true** に設定されている場合、Red Hat Single Sign-On サーバーの証明書はトラストストア経由で検証されますが、ホスト名の検証は行われません。この設定は開発時のみ使用してください。これは SSL 証明書の検証を一部無効にするため、実稼働環境では 使用しないでください。この設定は、テスト環境で役に立ちます。これは 任意 です。デフォルト値は **false** です。

truststore

値は、トラストストアファイルへのファイルパスです。**classpath:** でパスを接頭辞にすると、代わりにデプロイメントのクラスパスからトラストストアが取得されます。Red Hat Single Sign-On サーバーへの送信 HTTPS 通信に使用されます。HTTPS 要求を実行するクライアントでは、通信先のサーバーのホストを確認する方法が必要です。これは、トラストストアが行なうことです。キー

ストアには、1つ以上の信頼できるホスト証明書または認証局が含まれます。Red Hat Single Sign-On サーバーの SSL キーストアの公開証明書を抽出して、このトラストストアを作成できます。これは、**disableTrustManager** が **true** でない限り 必須 になります。

truststorePassword

トラストストアのパスワード。これは、トラストストア が設定され、トラストストアにパスワードが必要な場合は 必須 になります。

clientKeystore

これはキーストアファイルに対するファイルパスです。このキーストアには、アダプターが Red Hat Single Sign-On サーバーに対して HTTPS を要求する際に双方向 SSL のクライアント証明書が含まれます。これは 任意 です。

clientKeystorePassword

クライアントキーストアおよびクライアントの鍵のパスワード。これは、**clientKeystore** が設定されている場合は **REQUIRED** になります。

proxyUrl

HTTP 接続に使用する HTTP プロキシの URL。これは 任意 です。

3.1.2. JBoss EAP Adapter

JBoss EAP にデプロイされた WAR アプリケーションのセキュリティを保護できるようにするには、Red Hat Single Sign-On アダプターサブシステムをインストールおよび設定する必要があります。

次に、WAR で keycloak 設定ファイル **/WEB-INF/keycloak-saml.xml** を指定し、auth-method を web.xml 内の Keycloak-SAML に変更します。本セクションでは、両方の方法を説明します。

3.1.2.1. アダプターのインストール

各アダプターは、Red Hat Single Sign-On のダウンロードサイトの個別ダウンロードです。

JBoss EAP 7.x にインストールします。

```
$ cd $EAP_HOME
$ unzip rh-sso-saml-eap7-adapter.zip
```

JBoss EAP 6.x にインストールします。

```
$ cd $EAP_HOME
$ unzip rh-sso-saml-eap6-adapter.zip
```

これらの zip ファイルは、JBoss EAP ディストリビューション内に JBoss EAP SAML アダプターに固有の新しい JBoss モジュールを作成します。

モジュールを追加したら、アプリケーションサーバーのサーバー設定 (**domain.xml** または **standalone.xml**) 内で Red Hat Single Sign-On SAML サブシステムを有効にする必要があります。

サーバー設定の変更に役立つ CLI スクリプトがあります。サーバーを起動し、サーバーの bin ディレクトリーからスクリプトを実行します。

JBoss EAP 7.1 以降

```
$ cd $JBOSS_HOME
$ ./bin/jboss-cli.sh -c --file=bin/adapter-elytron-install-saml.cli
```

JBoss EAP 7.0 および EAP 6.4

```
$ cd $JBASS_HOME
$ ./bin/boass-cli.sh -c --file=bin/adaptor-install-saml.cli
```



注記

JBoss EAP 7.1 以上でも従来の非 Elytron アダプターを使用できます。つまり、これらのバージョンでも **adapter-install-saml.cli** を使用できます。ただし、新しい Elytron アダプターを使用することが推奨されます。

スクリプトは、以下に説明するように、拡張機能、サブシステム、およびオプションの security-domain を追加します。

```
<server xmlns="urn:jboss:domain:1.4">

  <extensions>
    <extension module="org.keycloak.keycloak-saml-adaptor-subsystem"/>
    ...
  </extensions>

  <profile>
    <subsystem xmlns="urn:jboss:domain:keycloak-saml:1.1"/>
    ...
  </profile>
```

セキュアな Web 層で作成されたセキュリティコンテキストを、呼び出している EJB (他の EE コンポーネント) に伝播する必要がある場合には、**keycloak** セキュリティドメインを EJB およびその他のコンポーネントと併用する必要があります。それ以外の場合、この設定は任意です。

```
<server xmlns="urn:jboss:domain:1.4">
  <subsystem xmlns="urn:jboss:domain:security:1.2">
    <security-domains>
      ...
      <security-domain name="keycloak">
        <authentication>
          <login-module code="org.keycloak.adapters.jboss.KeycloakLoginModule"
            flag="required"/>
        </authentication>
      </security-domain>
    </security-domains>
```

セキュリティコンテキストは EJB 階層に自動的に伝播されます。

3.1.2.2. JBoss SSO

JBoss EAP は、同じ JBoss EAP インスタンスにデプロイされた web アプリケーションに対するシングルサインオンのサポートが組み込まれています。これは、Red Hat Single Sign-On を使用する場合に有効にしないでください。

3.1.3. RPM からの JBoss EAP アダプターのインストール

RPM から EAP 7 アダプターをインストールします。



注記

Red Hat Enterprise Linux 7 では、チャンネル という用語はリポジトリという用語に置き換えられました。これらの手順では、リポジトリという用語のみが使用されています。

RPM から EAP 7 アダプターをインストールする前に、JBoss EAP 7 リポジトリにサブスクライブする必要があります。

前提条件

1. Red Hat Subscription Manager を使用して、Red Hat Enterprise Linux システムがお使いのアカウントに登録されている必要があります。詳細は、[Red Hat Subscription Management のドキュメント](#) を参照してください。
2. すでに別の JBoss EAP リポジトリにサブスクライブしている場合は、最初にそのリポジトリからサブスクライブを解除する必要があります。

Red Hat Enterprise Linux 6、7 の場合: Red Hat Subscription Manager を使用して、以下のコマンドで JBoss EAP 7.3 リポジトリにサブスクライブします。お使いの Red Hat Enterprise Linux のバージョンに応じて、<RHEL_VERSION> を 6 または 7 に置き換えてください。

```
$ sudo subscription-manager repos --enable=jb-eap-7-for-rhel-<RHEL_VERSION>-server-rpms
```

Red Hat Enterprise Linux 8 の場合: Red Hat Subscription Manager を使用して、以下のコマンドで JBoss EAP 7.3 リポジトリにサブスクライブします。

```
$ sudo subscription-manager repos --enable=jb-eap-7.3-for-rhel-8-x86_64-rpms --enable=rhel-8-for-x86_64-baseos-rpms --enable=rhel-8-for-x86_64-appstream-rpms
```

以下のコマンドを使用して、SAML の EAP 7 アダプターをインストールします。

```
$ sudo yum install eap7-keycloak-saml-adapter-sso7_4
```

または、Red Hat Enterprise Linux 8 に以下のいずれかを使用します。

```
$ sudo dnf install eap7-keycloak-adapter-sso7_4
```



注記

RPM インストールのデフォルトの EAP_HOME パスは /opt/rh/eap7/root/usr/share/wildfly です。

適切なモジュールインストールスクリプトを実行します。

SAML モジュールには、以下のコマンドを入力します。

```
$ $EAP_HOME/bin/jboss-cli.sh -c --file=$EAP_HOME/bin/adapter-install-saml.cli
```

インストールが完了しました。

RPM から EAP 6 アダプターをインストールします。



注記

Red Hat Enterprise Linux 7 では、チャンネル という用語はリポジトリという用語に置き換えられました。これらの手順では、リポジトリという用語のみが使用されています。

RPM から EAP 6 アダプターをインストールする前に、JBoss EAP 6 リポジトリにサブスクライブする必要があります。

前提条件

1. Red Hat Subscription Manager を使用して、Red Hat Enterprise Linux システムがお使いのアカウントに登録されている必要があります。詳細は、[Red Hat Subscription Management のドキュメント](#) を参照してください。
2. すでに別の JBoss EAP リポジトリにサブスクライブしている場合は、最初にそのリポジトリからサブスクライブを解除する必要があります。

Red Hat Subscription Manager を使用して、以下のコマンドを使用して JBoss EAP 6 リポジトリにサブスクライブします。お使いの Red Hat Enterprise Linux のバージョンに応じて、<RHEL_VERSION> を 6 または 7 に置き換えてください。

```
$ sudo subscription-manager repos --enable=jb-eap-6-for-rhel-<RHEL_VERSION>-server-rpms
```

以下のコマンドを使用して、SAML 用の EAP 6 アダプターをインストールします。

```
$ sudo yum install keycloak-saml-adapter-sso7_4-eap6
```



注記

RPM インストールのデフォルトの EAP_HOME パスは /opt/rh/eap6/root/usr/share/wildfly です。

適切なモジュールインストールスクリプトを実行します。

SAML モジュールには、以下のコマンドを入力します。

```
$ $EAP_HOME/bin/jboss-cli.sh -c --file=$EAP_HOME/bin/adapter-install-saml.cli
```

インストールが完了しました。

3.1.3.1. WAR ごとの設定

このセクションでは、WAR パッケージ内に設定ファイルを追加してファイルを編集することにより、WAR を直接保護する方法について説明します。

最初に、WAR の **WEB-INF** ディレクトリーに **keycloak-saml.xml** アダプター設定ファイルを作成する必要があります。この設定ファイルの形式は、「[一般的なアダプター設定](#)」セクションで説明されています。

次に、**web.xml** で **auth-method** を **KEYCLOAK-SAML** に設定する必要があります。また、標準のサーブレットセキュリティーを使用して URL に role-base 制約を指定する必要があります。以下は **web.xml** ファイルの例です。

```

<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_3_0.xsd"
  version="3.0">

  <module-name>customer-portal</module-name>

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Admins</web-resource-name>
      <url-pattern>/admin/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>admin</role-name>
    </auth-constraint>
    <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
  </security-constraint>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Customers</web-resource-name>
      <url-pattern>/customers/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>user</role-name>
    </auth-constraint>
    <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
  </security-constraint>

  <login-config>
    <auth-method>KEYCLOAK-SAML</auth-method>
    <realm-name>this is ignored currently</realm-name>
  </login-config>

  <security-role>
    <role-name>admin</role-name>
  </security-role>
  <security-role>
    <role-name>user</role-name>
  </security-role>
</web-app>

```

auth-method 設定を除くすべての標準サーブレット設定。

3.1.3.2. Red Hat Single Sign-On SAML サブシステムを使用した WAR のセキュリティー保護

Red Hat Single Sign-On で WAR を保護するために、WAR をオープンする必要はありません。代わりに、Red Hat Single Sign-On SAML Adapter Subsystem 経由で外部からセキュリティーを確保することもできます。KEYCLOAK-SAML を **auth-method** として指定する必要はありませんが、**web.xml** で

security-constraints を定義する必要があります。ただし、**WEB-INF/keycloak-saml.xml** ファイルを作成する必要はありません。このメタデータは、サーバーのサブシステム設定 **domain.xml** または **standalone.xml** の XML 内で定義されます。

```
<extensions>
  <extension module="org.keycloak.keycloak-saml-adapter-subsystem"/>
</extensions>

<profile>
  <subsystem xmlns="urn:jboss:domain:keycloak-saml:1.1">
    <secure-deployment name="WAR MODULE NAME.war">
      <SP entityID="APPLICATION URL">
        ...
      </SP>
    </secure-deployment>
  </subsystem>
</profile>
```

secure-deployment name 属性は、セキュリティを保護する WAR を識別します。この値は **web.xml** に **.war** を追加した **module-name** です。残りの設定は、**一般的なアダプター設定** で定義される **keycloak-saml.xml** 設定と同じ XML 構文を使用します。

設定例:

```
<subsystem xmlns="urn:jboss:domain:keycloak-saml:1.1">
  <secure-deployment name="saml-post-encryption.war">
    <SP entityID="http://localhost:8080/sales-post-enc/"
      sslPolicy="EXTERNAL"
      nameIDPolicyFormat="urn:oasis:names:tc:SAML:1.1:nameid-format:unspecified"
      logoutPage="/logout.jsp"
      forceAuthentication="false">
      <Keys>
        <Key signing="true" encryption="true">
          <KeyStore resource="/WEB-INF/keystore.jks" password="store123">
            <PrivateKey alias="http://localhost:8080/sales-post-enc/" password="test123"/>
            <Certificate alias="http://localhost:8080/sales-post-enc/">
          </KeyStore>
        </Key>
      </Keys>
      <PrincipalNameMapping policy="FROM_NAME_ID"/>
      <RoleIdentifiers>
        <Attribute name="Role"/>
      </RoleIdentifiers>
      <IDP entityID="idp">
        <SingleSignOnService signRequest="true"
          validateResponseSignature="true"
          requestBinding="POST"
          bindingUrl="http://localhost:8080/auth/realms/saml-demo/protocol/saml"/>

        <SingleLogoutService
          validateRequestSignature="true"
          validateResponseSignature="true"
          signRequest="true"
          signResponse="true"
          requestBinding="POST"
```



```

responseBinding="POST"
postBindingUrl="http://localhost:8080/auth/realms/saml-demo/protocol/saml"
redirectBindingUrl="http://localhost:8080/auth/realms/saml-demo/protocol/saml"/>
<Keys>
  <Key signing="true" >
    <KeyStore resource="/WEB-INF/keystore.jks" password="store123">
      <Certificate alias="saml-demo"/>
    </KeyStore>
  </Key>
</Keys>
</IDP>
</SP>
</secure-deployment>
</subsystem>

```

3.1.4. Java Servlet Filter Adapter

サーブレットプラットフォーム用のアダプターを持たない Java サーブレットアプリケーションで SAML を使用する場合は、Red Hat Single Sign-On が持つサーブレットフィルターアダプターを使用することを選択できます。このアダプターは、他のアダプターとは異なります。[「一般的なアダプター設定」](#) セクションに定義されているように **/WEB-INF/keycloak-saml.xml** ファイルを指定する必要がありますが、**web.xml** でセキュリティ制約を定義する必要はありません。代わりに、Red Hat Single Sign-On サーブレットフィルターアダプターを使用してフィルターマッピングを定義し、セキュリティを保護する url パターンを保護します。



注記

Backchannel ログアウトの動作は、標準アダプターとは少々異なります。HTTP セッションを無効にする代わりに、セッション ID をログアウトとしてマークします。セッション ID に基づいて http セッションを任意に無効にする方法はありません。



警告

SAML フィルターを使用するクラスター化されたアプリケーションがある場合は、バックチャネルログアウトは現在機能しません。

```

<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_3_0.xsd"
  version="3.0">

  <module-name>customer-portal</module-name>

  <filter>
    <filter-name>Keycloak Filter</filter-name>
    <filter-class>org.keycloak.adapters.saml.servlet.SamlFilter</filter-class>
  </filter>
  <filter-mapping>

```

```

    <filter-name>Keycloak Filter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
</web-app>

```

Red Hat Single Sign-On フィルターには、他のアダプターと同じ設定パラメーターがあります。ただし、これらはコンテキストパラメーターではなく、フィルター init パラメーターとして定義する必要があります。

さまざまな異なる安全な URL パターンと安全でない URL パターンがある場合は、複数のフィルターマッピングを定義できます。



警告

/saml に対応するフィルターマッピングがある。このマッピングは、すべてのサーバーコールバックに対応します。

IdP で SP を登録する場合は、**http[s]://hostname/{context-root}/saml** を Assert Consumer Service URL および Single Logout Service URL として登録する必要があります。

このフィルターを使用するには、この maven アーティファクトを WAR pom に組み込みます。

```

<dependency>
  <groupId>org.keycloak</groupId>
  <artifactId>keycloak-saml-servlet-filter-adapter</artifactId>
  <version>9.0.17.redhat-00001</version>
</dependency>

```

マルチテナントを使用するには、**keycloak.config.resolver** パラメーターをフィルターパラメーターとして渡す必要があります。

```

<filter>
  <filter-name>Keycloak Filter</filter-name>
  <filter-class>org.keycloak.adapters.saml.servlet.SamlFilter</filter-class>
  <init-param>
    <param-name>keycloak.config.resolver</param-name>
    <param-value>example.SamlMultiTenantResolver</param-value>
  </init-param>
</filter>

```

3.1.5. ID プロバイダーを使用した登録

サーブレットベースのアダプタごとに、アサートコンシューマサービス URL およびシングルログアウトサービスに登録するエンドポイントは **/saml** が追加されたサーブレットアプリケーションのベース URL、つまり **https://example.com/contextPath/saml** である必要があります。

3.1.6. ログアウト

Web アプリケーションからログアウトする方法は複数あります。Java EE サーブレットコンテナで

は、`HttpServletRequest.logout()` を呼び出すことができます。他のブラウザアプリケーションでは、ブラウザをセキュリティ制約のある Web アプリケーションの任意の URL に指定し、クエリパラメーター GLO (つまり `http://myapp?GLO=true`) に渡すことができます。ブラウザに SSO セッションがある場合は、ログアウトします。

3.1.6.1. クラスター化された環境でログアウト

SAML アダプターは内部的には、SAML セッションインデックス、プリンシパル名 (既知の場合)、および HTTP セッション ID 間のマッピングを保存します。このマッピングは、分散可能なアプリケーションのクラスター全体で、JBoss アプリケーションサーバーファミリー (WildFly 10/11、EAP 6/7) で維持できます。前提条件として、HTTP セッションをクラスター全体に分散する必要があります (つまり、アプリケーションはアプリケーションの `web.xml` の `<distributable>` タグでマークされます)。

この機能を有効にするには、以下のセクションを `/WEB_INF/web.xml` ファイルに追加します。

EAP 7 (WildFly 10/11) の場合:

```
<context-param>
  <param-name>keycloak.sessionIdMapperUpdater.classes</param-name>
  <param-
value>org.keycloak.adapters.saml.wildfly.infinispan.InfinispanSessionCacheIdMapperUpdater</param-value>
</context-param>
```

EAP 6 の場合:

```
<context-param>
  <param-name>keycloak.sessionIdMapperUpdater.classes</param-name>
  <param-
value>org.keycloak.adapters.saml.jbossweb.infinispan.InfinispanSessionCacheIdMapperUpdater</param-value>
</context-param>
```

デプロイメントのセッションキャッシュの名前が **deployment-cache** の場合、SAML マッピングに使用されるキャッシュの名前が **deployment-cache.ssoCache** になります。キャッシュの名前は、コンテキストパラメーター `keycloak.sessionIdMapperUpdater.infinispan.cacheName` で上書きできます。キャッシュを含むキャッシュコンテナはデプロイメントセッションキャッシュを含むものと同じですが、コンテキストパラメーター `keycloak.sessionIdMapperUpdater.infinispan.containerName` で上書きできます。

デフォルトでは、SAML マッピングキャッシュの設定はセッションキャッシュから派生します。この設定は、他のキャッシュと同じように、サーバーのキャッシュ設定セクションで手動でオーバーライドできます。

現在、信頼できるサービスを提供するには、SAML セッションキャッシュにレプリケートされたキャッシュを使用することが推奨されます。分散キャッシュを使用すると、SAML のログアウト要求が HTTP セッションマッピングへの SAML セッションインデックスにアクセスできないノードに到達する結果となり、ログアウトに失敗する可能性があります。

3.1.6.2. クロス DC シナリオにおけるログアウト

DC の間のシナリオは、WildFly 10 以降および EAP 7 以降にのみ適用されます。

複数のデータセンターにまたがるセッションを処理するには、特別な処理が必要です。以下のシナリオを見てみましょう。

1. ログイン要求は、データセンター 1 のクラスター内で処理されます。
2. 管理者は、特定の SAML セッションに対してログアウト要求を発行し、要求はデータセンター 2 に送信されます。

データセンター 2 は、データセンター 1 (および HTTP セッションを共有する他のすべてのデータセンター) にあるすべてのセッションをログアウトする必要があります。

この場合、[上記](#) の SAML セッションキャッシュを個別のクラスター内だけでなく、[スタンドアロン Infinispan/JDG サーバーを介して](#) すべてのデータセンターにレプリケートする必要があります。

1. スタンドアロン Infinispan/JDG サーバーにキャッシュを追加する必要があります。
2. 各 SAML セッションキャッシュのリモートストアとして、以前のアイテムからのキャッシュを追加する必要があります。

デプロイメント中にリモートストアが SAML セッションキャッシュに存在することが検出されると、変更がないか監視され、それに応じてローカル SAML セッションキャッシュが更新されます。

3.1.7. assertion 属性の取得

正常な SAML ログインの後、アプリケーションコードが SAML アサーションで渡される属性値を取得することをお勧めします。**HttpServletRequest.getUserPrincipal()**

は、**org.keycloak.adapters.saml.SamlPrincipal** と呼ばれる Red Hat Single Sign-On 固有のクラスに型変換できる **Principal** オブジェクトを返します。このオブジェクトを使用すると、raw アサーションを確認でき、属性値を検索するための便利な関数もあります。

```
package org.keycloak.adapters.saml;

public class SamlPrincipal implements Serializable, Principal {
    /**
     * Get full saml assertion
     *
     * @return
     */
    public AssertionType getAssertion() {
        ...
    }

    /**
     * Get SAML subject sent in assertion
     *
     * @return
     */
    public String getSamlSubject() {
        ...
    }

    /**
     * Subject nameID format
     *
     * @return
     */
    public String getNameIDFormat() {
        ...
    }
}
```

```

@Override
public String getName() {
    ...
}

/**
 * Convenience function that gets Attribute value by attribute name
 *
 * @param name
 * @return
 */
public List<String> getAttributes(String name) {
    ...
}

/**
 * Convenience function that gets Attribute value by attribute friendly name
 *
 * @param friendlyName
 * @return
 */
public List<String> getFriendlyAttributes(String friendlyName) {
    ...
}

/**
 * Convenience function that gets first value of an attribute by attribute name
 *
 * @param name
 * @return
 */
public String getAttribute(String name) {
    ...
}

/**
 * Convenience function that gets first value of an attribute by attribute name
 *
 * @param friendlyName
 * @return
 */
public String getFriendlyAttribute(String friendlyName) {
    ...
}

/**
 * Get set of all assertion attribute names
 *
 * @return
 */
public Set<String> getAttributeNames() {
    ...
}

```

```

/**
 * Get set of all assertion friendly attribute names
 *
 * @return
 */
public Set<String> getFriendlyNames() {
    ...
}
}

```

3.1.8. エラー処理

Red Hat Single Sign-On には、サーブレットベースのクライアントアダプターに対するエラー処理機能があります。認証でエラーが発生した場合、クライアントアダプターは **HttpServletResponse.sendError()** を呼び出します。**web.xml** ファイル内に **error-page** を設定して、必要なエラーを処理できます。クライアントアダプターは 400、401、403、および 500 のエラーを出力できます。

```

<error-page>
  <error-code>403</error-code>
  <location>/ErrorHandler</location>
</error-page>

```

クライアントアダプターは、取得可能な **HttpServletRequest** 属性も設定します。属性名は **org.keycloak.adapters.spi.AuthenticationError** です。このオブジェクトは、**org.keycloak.adapters.saml.SamlAuthenticationError** に型変換されます。このクラスは、何が起こったのかを正確に知ることができます。この属性が設定されていない場合、アダプターはエラーコードを担当しませんでした。

```

public class SamlAuthenticationError implements AuthenticationError {
    public static enum Reason {
        EXTRACTION_FAILURE,
        INVALID_SIGNATURE,
        ERROR_STATUS
    }

    public Reason getReason() {
        return reason;
    }

    public StatusResponseType getStatus() {
        return status;
    }
}

```

3.1.9. トラブルシューティング

問題のトラブルシューティングを行う最適な方法として、クライアントアダプターと Red Hat Single Sign-On サーバーの両方で SAML のデバッグをオンにすることができます。ロギングフレームワークを使用して、**org.keycloak.saml** パッケージのログレベルを **DEBUG** に設定します。これをオンにすると、サーバーとの間で送受信される SAML 要求および応答ドキュメントを確認できます。

3.1.10. マルチテナンシー

SAML は [マルチテナント](#) の OIDC と同じ機能を提供します。つまり、1つのターゲットアプリケーション (WAR) は複数の Red Hat Single Sign-On レルムでセキュリティー保護されます。レルムは、同じ Red Hat Single Sign-On インスタンスまたは別のインスタンスに配置できます。

これを実行するには、複数の **keycloak-saml.xml** アダプター設定ファイルが必要です。

異なるアダプター設定ファイルを含む WAR の複数のインスタンスを異なるコンテキストパスにデプロイすることができますが、これは不便である可能性があるため、コンテキストパス以外のものに基づいてレルムを選択することをお勧めします。

Red Hat Single Sign-On では、カスタム設定リゾルバーの設定が可能で、これにより、リクエストごとに使用するアダプター設定を選択できるようになります。SAML では、設定はログイン処理のみに関連し、ユーザーがログインするとセッションが認証され、返された **keycloak-saml.xml** が異なるかどうかは問題ありません。そのため、同じセッションで同じ設定を返すことが、適切な方法になります。

そのためには、**org.keycloak.adapters.saml.SamlConfigResolver** の実装を作成します。以下の例では、**Host** ヘッダーを使用して適切な設定を検索し、アプリケーションの Java クラスパスから関連する要素を読み込みます。

```
package example;

import java.io.InputStream;
import org.keycloak.adapters.saml.SamlConfigResolver;
import org.keycloak.adapters.saml.SamlDeployment;
import org.keycloak.adapters.saml.config.parsers.DeploymentBuilder;
import org.keycloak.adapters.saml.config.parsers.ResourceLoader;
import org.keycloak.adapters.spi.HttpFacade;
import org.keycloak.saml.common.exceptions.ParsingException;

public class SamlMultiTenantResolver implements SamlConfigResolver {

    @Override
    public SamlDeployment resolve(HttpFacade.Request request) {
        String host = request.getHeader("Host");
        String realm = null;
        if (host.contains("tenant1")) {
            realm = "tenant1";
        } else if (host.contains("tenant2")) {
            realm = "tenant2";
        } else {
            throw new IllegalStateException("Not able to guess the keycloak-saml.xml to load");
        }

        InputStream is = getClass().getResourceAsStream("/" + realm + "-keycloak-saml.xml");
        if (is == null) {
            throw new IllegalStateException("Not able to find the file /" + realm + "-keycloak-saml.xml");
        }

        ResourceLoader loader = new ResourceLoader() {
            @Override
            public InputStream getResourceAsStream(String path) {
                return getClass().getResourceAsStream(path);
            }
        };

        try {
```



```

        return new DeploymentBuilder().build(is, loader);
    } catch (ParsingException e) {
        throw new IllegalStateException("Cannot load SAML deployment", e);
    }
}
}
}

```

また、**web.xml** のコンテキストパラメーター **keycloak.config.resolver** で使用する **SamlConfigResolver** 実装を構成する必要があります。

```

<web-app>
...
<context-param>
  <param-name>keycloak.config.resolver</param-name>
  <param-value>example.SamlMultiTenantResolver</param-value>
</context-param>
</web-app>

```

3.2. APACHE HTTPD モジュール MOD_AUTH_MELLON

mod_auth_mellon モジュールは SAML の Apache HTTPD プラグインです。言語/環境が Apache HTTPD をプロキシとして使用することをサポートしている場合は、**mod_auth_mellon** を使用して SAML で Web アプリケーションを保護できます。このモジュールの詳細は、GitHub リポジトリ **mod_auth_mellon** を参照してください。

mod_auth_mellon を設定するには、以下が必要になります。

- アイデンティティプロバイダー (IdP) エンティティ記述子 XML ファイル。Red Hat Single Sign-On または他の SAML IdP への接続を記述します。
- セキュリティ保護するアプリケーションの SAML 接続および設定を記述する SP エンティティ記述子 XML ファイル。
- 秘密鍵 PEM ファイル。これは、アプリケーションがドキュメントの署名に使用するプライベートキーを定義する PEM 形式のテキストファイルです。
- アプリケーションの証明書を定義するテキストファイルである証明書 PEM ファイル。
- mod_auth_mellon** 固有の Apache HTTPD モジュール設定。

3.2.1. Red Hat Single Sign-On での **mod_auth_mellon** の設定

以下の 2 つのホストがあります。

- Red Hat Single Sign-On が実行されているホスト。Red Hat Single Sign-On は SAML アイデンティティプロバイダー (IdP) であるため、**\$idp_host** と呼ばれます。
- Web アプリケーションが実行されているホスト。これは **\$sp_host** と呼ばれます。SAML では、IdP を使用するアプリケーションはサービスプロバイダー (SP) と呼ばれます。

以下のすべての手順は、**root** 権限で **\$sp_host** で実行する必要があります。

3.2.1.1. パッケージのインストール

必要なパッケージをインストールするには、以下が必要です。

- Apache Web Server (httpd)
- Apache の Mellon SAML SP アドオンモジュール
- X509 証明書を作成するツール

必要なパッケージをインストールするには、以下のコマンドを実行します。

```
yum install httpd mod_auth_mellon mod_ssl openssl
```

3.2.1.2. Apache SAML の設定ディレクトリーの作成

Apache の SAML の使用に関連する設定ファイルを1つの場所で維持することを推奨します。

Apache 設定の root /etc/httpd の下に、saml2 という名前の新規ディレクトリーを作成します。

```
mkdir /etc/httpd/saml2
```

3.2.1.3. Mellon サービスプロバイダーの設定

Apache アドオンモジュールの設定ファイルは /etc/httpd/conf.d ディレクトリーにあり、ファイル名の拡張子は .conf になります。/etc/httpd/conf.d/mellon.conf ファイルを作成し、Mellon の設定ディレクティブをこれに配置する必要があります。

Mellon の設定ディレクティブは、大まかに 2 つのクラスの情報に分類できます。

- SAML 認証を保護する URL
- 保護された URL が参照された場合に使用される SAML パラメーター。

Apache 設定ディレクティブは通常、場所と呼ばれる URL 領域内の階層ツリー構造に従います。Mellon が保護するには、URL の場所を1つ以上指定する必要があります。各場所に適用される設定パラメーターを追加する方法に柔軟性があります。必要なパラメーターをすべてロケーションブロックに追加するか、Mellon パラメーターを特定の保護された場所が継承する URL の場所階層にある共通の場所に追加するかのいずれかを実行できます (またはこの 2 つのなんらかの組み合わせを実行)。どの場所が SAML アクションをトリガーしても、SP は同じように動作するのが一般的であるため、ここで使用する設定例では、一般的な Mellon 設定ディレクティブを階層の root に配置してから、Mellon によって保護される特定の場所を最小限のディレクティブで定義できます。このストラテジーでは、保護されている場所ごとに同じパラメーターが重複しないようにします。

この例には保護された場所が1つ (https://\$sp_host/private) しかありません。

Mellon サービスプロバイダーを設定するには、以下の手順を実行します。

1. 以下の内容で /etc/httpd/conf.d/mellon.conf ファイルを作成します。

```
<Location / >
MellonEnable info
MellonEndpointPath /mellon/
MellonSPMetadataFile /etc/httpd/saml2/mellon_metadata.xml
MellonSPPrivateKeyFile /etc/httpd/saml2/mellon.key
MellonSPCertFile /etc/httpd/saml2/mellon.crt
MellonIdPMetadataFile /etc/httpd/saml2/idp_metadata.xml
</Location>
<Location /private >
```

```
AuthType Mellon
MellonEnable auth
Require valid-user
</Location>
```



注記

上記のコードで参照されるファイルの一部は、後の手順で作成されます。

3.2.1.4. サービスプロバイダーメタデータの作成

SAML IdP および SP は、XML 形式の SAML メタデータを交換します。メタデータのスキーマは標準であるため、参加している SAML エンティティが互いのメタデータを消費できるようにします。以下が必要です。

- SP が使用する IdP のメタデータ
- IdP に提供された SP を記述するメタデータ

SAML メタデータのコンポーネントの1つは X509 証明書です。これらの証明書は2つの目的で使用されます。

- SAML メッセージを署名し、メッセージが予期されたパーティーから発信されたことを受信側が証明できるようにします。
- トランスポート中にメッセージを暗号化します (SAML メッセージは通常 TLS で保護されているトランスポートで発生するため、ほとんど使用されません)。

すでに認証局 (CA) を持っている場合は、独自の証明書を使用できます。または、自己署名証明書を生成することもできます。この例では簡単にするために、自己署名証明書が使用されています。

Mellon の SP メタデータは `mod_auth_mellon` のインストール済みバージョンの機能を反映する必要があるため、有効な SP メタデータ XML である必要があり、X509 証明書 (X509 証明書の生成に精通していない場合は、証明書の作成はわかりにくい可能性があります) を含む必要があります。SP メタデータを生成する最も便利な方法は、`mod_auth_mellon` パッケージ (`mellon_create_metadata.sh`) に含まれるツールを使用することです。生成されたメタデータは、テキストファイルであるため、常に後で編集できます。このツールは、X509 キーおよび証明書も作成します。

SAML IdP および SP は、EntityID として知られる一意の名前を使用して識別します。Mellon メタデータ作成ツールを使用するには、以下が必要です。

- EntityID。これは通常 SP の URL であり、多くの場合、SP メタデータを取得できる SP の URL です。
- SP の SAML メッセージが使用される URL。Mellon は `MellonEndPointPath` を呼び出します。

SP メタデータを作成するには、以下の手順を行います。

1. ヘルパーシェルスクリプト変数をいくつか作成します。

```
fqdn=`hostname`
mellon_endpoint_url="https://${fqdn}/mellon"
mellon_entity_id="${mellon_endpoint_url}/metadata"
file_prefix="$(echo "$mellon_entity_id" | sed 's/[^A-Za-z.]/_/g' | sed 's/_/_/g')
```

2. 以下のコマンドを実行して、Mellon メタデータ作成ツールを呼び出します。

```
/usr/libexec/mod_auth_mellon/mellon_create_metadata.sh $mellon_entity_id
$mellon_endpoint_url
```

3. 生成されたファイルを (上記で作成した /etc/httpd/conf.d/mellon.conf ファイルで参照した) 宛先に移動します。

```
mv ${file_prefix}.cert /etc/httpd/saml2/mellon.crt
mv ${file_prefix}.key /etc/httpd/saml2/mellon.key
mv ${file_prefix}.xml /etc/httpd/saml2/mellon_metadata.xml
```

3.2.1.5. Red Hat Single Sign-On ID プロバイダーへの Mellon サービスプロバイダーの追加

前提条件: Red Hat Single Sign-On IdP が \$idp_host にすでにインストールされています。

Red Hat Single Sign-On は、すべてのユーザー、クライアントなどが、いわゆるレルムにグループ化されるマルチテナンシーをサポートします。各レルムは、他のレルムとは独立しています。Red Hat Single Sign-On で既存のレルムを使用できますが、この例では test_realm という新しいレルムを作成し、そのレルムを使用する方法が示されています。

これらの操作はすべて、Red Hat Single Sign-On 管理 Web コンソールを使用して実行されます。\$idp_host の管理者のユーザー名およびパスワードが必要です。

以下の手順を実行してください。

1. 管理コンソールを開き、管理者のユーザー名とパスワードを入力してログオンします。
管理コンソールにログインすると、既存のレルムがあります。Red Hat Single Sign-On が最初に設定されると、root レルム (master) がデフォルトで作成されます。以前に作成されたレルムは、管理コンソールの左上隅のドロップダウンリストに一覧表示されます。
2. レルムドロップダウンリストから **Add レルム** を選択します。
3. Name フィールドに **test_realm** と入力して、**Create** をクリックします。

3.2.1.5.1. レルムのクライアントとしての Mellon Service Provider の追加

Red Hat Single Sign-On SAML SP はクライアントと呼ばれます。SP を追加するには、レルムの Clients セクションにいる必要があります。

1. 左側の Clients メニュー項目をクリックし、右上隅の **Create** をクリックして新規クライアントを作成します。

3.2.1.5.2. Mellon SP クライアントの追加

Mellon SP クライアントを追加するには、以下の手順を実行します。

1. クライアントプロトコルを SAML に設定します。Client Protocol ドロップダウンリストから、**saml** を選択します。
2. 上記で作成されたすべての SP メタデータファイル (/etc/httpd/saml2/mellon_metadata.xml) を提供します。ブラウザーが実行している場所に応じて、ブラウザーがファイルを見つけられるように、SP メタデータを \$sp_host からブラウザーが実行されているマシンにコピーする必要があります。
3. **Save** をクリックします。

3.2.1.5.3. Mellon SP クライアントの編集

設定を提案するクライアント設定パラメーターは複数あります。

- 「Force POST Binding」が On であることを確認します。
- paosResponse を有効なリダイレクト URI 一覧に追加します。
 1. 「Valid Redirect URIs」の postResponse URL をコピーし、「+」のすぐ下の空の add text フィールドに貼り付けます。
 2. 「postResponse」を「paosResponse」に変更します。(SAML ECP には paosResponse URL が必要です。)
 3. 下部の **Save** をクリックします。

多くの SAML SP は、グループのユーザーのメンバーシップに基づいて承認を決定します。Red Hat Single Sign-On IdP はユーザーグループ情報を管理できますが、IdP が SAML 属性として提供するように設定されていない限り、ユーザーのグループは提供しません。

ユーザーのグループを SAML 属性として提供するように IdP を設定するには、以下の手順を実行します。

1. クライアントの Mappers タブをクリックします。
2. Mappers ページの右上隅にある **Create** をクリックします。
3. Mapper Type ドロップダウンリストから **Group list** を選択します。
4. Name を "group list" に設定します。
5. SAML 属性名を「グループ」に設定します。
6. **Save** をクリックします。

残りの手順は \$sp_host で実行されます。

3.2.1.5.4. アイデンティティプロバイダーメタデータの取得

IdP でレلمを作成したので、それに関連付けられた IdP メタデータを取得して、Mellon SP がそれを認識するようになる必要があります。以前に作成した /etc/httpd/conf.d/mellon.conf ファイルでは、MellonIdPMetadataFile は /etc/httpd/saml2/idp_metadata.xml として指定されますが、これまでのところ、そのファイルは \$sp_host に存在していませんでした。このファイルを取得するには、IdP から取得します。

1. \$idp_host を正しい値に置き換えて、IdP からファイルを取得します。

```
curl -k -o /etc/httpd/saml2/idp_metadata.xml \
https://$idp_host/auth/realms/test_realm/protocol/saml/descriptor
```

Mellon が完全に設定されるようになりました。

2. Apache 設定ファイルの構文チェックを実行するには、以下を行います。

```
apachectl configtest
```



注記

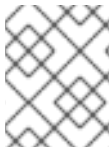
Configtest は apachectl の -t 引数と同じです。設定テストでエラーが表示される場合には、次に進む前に修正してください。

3. Apache サーバーを再起動します。

```
systemctl restart httpd.service
```

これで、Red Hat Single Sign-On を test_realm の SAML IdP として、mod_auth_mellon を SAML SP として設定し、**\$idp_host** に対して認証することで URL \$sp_host/protected (およびその下のすべてのもの) を保護することができました。

第4章 DOCKER レジストリーの設定



注記

Docker 認証はデフォルトで無効になっています。有効にするには、「[プロファイル](#)」を参照してください。

本セクションでは、Red Hat Single Sign-On を認証サーバーとして使用するように Docker レジストリーを設定する方法を説明します。

Docker レジストリーのセットアップおよび設定方法の詳細は、『[Docker Registry Configuration Guide](#)』を参照してください。

4.1. DOCKER レジストリー設定ファイルのインストール

高度な Docker レジストリー設定を使用しているユーザーの場合は、通常、独自のレジストリー設定ファイルを提供することをお勧めします。Red Hat Single Sign-On Docker プロバイダーは、**Registry Config File** フォーマットオプションを介してこのメカニズムをサポートします。このオプションを選択すると、以下のような出力が生成されます。

```
auth:
  token:
    realm: http://localhost:8080/auth/realms/master/protocol/docker-v2/auth
    service: docker-test
    issuer: http://localhost:8080/auth/realms/master
```

この出力は、既存のレジストリー設定ファイルにコピーできます。ファイルの設定方法は、[レジストリー設定ファイル](#) の仕様を参照してください、または [基本的な例](#) から始めてください。



警告

rootcertbundle フィールドを Red Hat Single Sign-On レalmの公開証明書の場合で設定することを忘れないでください。認証設定は、この引数なしでは機能しません。

4.2. DOCKER レジストリー環境変数オーバーライドインストール

多くの場合、開発または POC Docker レジストリーに単純な環境変数オーバーライドを使用することが適しています。通常、このアプローチは実稼働環境での使用は推奨されませんが、急場しのぎの方法でレジストリーを起動する必要がある場合には役立ちます。クライアントのインストールタブから **Variable Override** フォーマットオプションを使用するだけで、出力は以下のようになります。

```
REGISTRY_AUTH_TOKEN_REALM: http://localhost:8080/auth/realms/master/protocol/docker-v2/auth
REGISTRY_AUTH_TOKEN_SERVICE: docker-test
REGISTRY_AUTH_TOKEN_ISSUER: http://localhost:8080/auth/realms/master
```


**警告**

REGISTRY_AUTH_TOKEN_ROOTCERTBUNDLE を、Red Hat Single Sign-On レルムの公開証明書で上書きする設定を忘れないでください。認証設定は、この引数なしでは機能しません。

4.3. DOCKER COMPOSE YAML ファイル

**警告**

このインストール方法は、Red Hat Single Sign-On サーバーに対して docker レジストリーを認証する簡単な方法になります。これは開発のみを目的としており、本番環境または本番環境のような環境で使用しないでください。

zip ファイルインストールメカニズムは、Red Hat Single Sign-On サーバーが Docker レジストリーとどのように対話するかを理解したい開発者向けのクイックスタートを提供します。設定するには、以下を実施します。

1. 必要なレルムから、クライアント設定を作成します。この時点で、Docker レジストリーはありません。クイックスタートがその部分を処理します。
2. インストールタブから「Docker Compose YAML」オプションを選択し、.zip ファイルをダウンロードします。
3. 目的の場所にアーカイブを展開して、ディレクトリーを開きます。
4. **docker-compose up** で Docker レジストリーを起動します。

**注記**

HTTP Basic 認証フローはフォームを表示しないため、'master' 以外のレルムで Docker レジストリークライアントを設定することをお勧めします。

上記の設定が行われ、keycloak サーバーおよび Docker レジストリーが実行されると、Docker 認証が正常に行われるはずです。

```
[user ~]# docker login localhost:5000 -u $username
Password: *****
Login Succeeded
```

第5章 クライアント登録

アプリケーションまたはサービスが Red Hat Single Sign-On を使用するには、Red Hat Single Sign-On でクライアントを登録する必要があります。管理者は管理コンソール (または管理 REST エンドポイント) でこれを実行できますが、クライアントは Red Hat Single Sign-On クライアント登録サービスを介して登録することもできます。

クライアント登録サービスは、Red Hat Single Sign-On Client Representations、OpenID Connect Client Meta Data および SAML Entity Descriptors の組み込みサポートを提供します。クライアント登録サービスのエンドポイントは `/auth/realms/<realm>/clients-registrations/<provider>` です。

サポートされている組み込み プロバイダー は以下のとおりです。

- デフォルト - Red Hat Single Sign-On Client Representation (JSON)
- インストール - Red Hat Single Sign-On Adapter Configuration (JSON)
- openid-connect - OpenID Connect Client Metadata Description (JSON)
- saml2-entity-descriptor - SAML エンティティ記述子 (XML)

以下のセクションでは、異なるプロバイダーを使用する方法を説明します。

5.1. 認証

クライアント登録サービスを呼び出すには、通常トークンが必要です。トークンは、ベアラートークン、初期アクセストークン、または登録アクセストークンにすることができます。トークンなしで新しいクライアントを登録する別の方法もありますが、その場合はクライアント登録ポリシーを設定する必要があります (以下を参照)。

5.1.1. ベアラートークン

ベアラートークンは、ユーザーまたは Service Account の代わりに発行できます。エンドポイントを呼び出すには、以下のパーミッションが必要です (詳細は『[サーバー管理ガイド](#)』を参照)。

- create-client または manage-client - クライアントを作成するため
- view-client または manage-client - クライアントを表示するため
- manage-client - クライアントを更新または削除する

ベアラートークンを使用してクライアントを作成する場合は、**create-client** ロールのサービスアカウントからトークンを使用することが推奨されます (詳細は『[サーバー管理ガイド](#)』を参照)。

5.1.2. 初期アクセストークン

新しいクライアントを登録するための推奨されるアプローチは、初期アクセストークンを使用することです。初期アクセストークンは、クライアントの作成にのみ使用でき、有効期限を設定できるほか、作成できるクライアントの数に制限を設定できます。

初期アクセストークンは管理コンソールを使用して作成できます。新しい初期アクセストークンを作成するには、最初に管理コンソールでレルムを選択し、左側のメニューで **Realm Settings** をクリックし、ページに表示されるタブの **Client Registration** をクリックします。最後に **Initial Access Tokens** サブタブをクリックします。

これで、既存の初期アクセストークンを確認できるようになります。アクセスがあれば、不要になったトークンを削除できます。トークンの作成時にのみ、トークンの値を取得できます。新規トークンを作成するには、**Create** をクリックします。オプションで、トークンの有効期間を追加できるようになりました。また、トークンを使用して作成できるクライアントの数も追加できます。**Save** をクリックすると、トークンの値が表示されます。

このトークンは後で取得できないため、今すぐこのトークンをコピーして貼り付けることが重要です。コピー/貼り付けを忘れた場合は、このトークンを削除して別のトークンを作成してください。

トークン値は、クライアント登録サービスを呼び出すときに、リクエストの Authorization ヘッダーに追加することにより、標準のベアラートークンとして使用されます。以下に例を示します。

```
Authorization: bearer eyJhbGciOiJSUz...
```

5.1.3. 登録アクセストークン

クライアント登録サービス経由でクライアントを作成する場合、応答には登録アクセストークンが含まれます。登録アクセストークンは、後でクライアント設定を取得するアクセスを提供しますが、クライアントを更新または削除するためのアクセスも提供します。登録アクセストークンは、ベアラートークンまたは初期アクセストークンと同じ方法でリクエストに含まれます。登録アクセストークンは1回だけ有効です。使用すると、応答に新しいトークンが含まれます。

クライアント登録サービスの外部でクライアントが作成された場合、それに関連付けられた登録アクセストークンはありません。管理コンソールから作成できます。これは、特定のクライアントのトークンを失った場合にも便利です。新しいトークンを作成するには、管理コンソールでクライアントを見つけ、**Credentials** をクリックします。次に、**Generate registration access token** をクリックします。

5.2. RED HAT SINGLE SIGN-ON REPRESENTATIONS

デフォルトのクライアント登録プロバイダーは、クライアントの作成、取得、更新、および削除に使用できます。Red Hat Single Sign-On Client Representation 形式を使用します。これは、たとえばプロトコルマッパーの設定など、管理コンソールから設定できるのとまったく同じように、クライアントを設定するためのサポートを提供します。

クライアントを作成するには、クライアント表現 (JSON) を作成し、**/auth/realms/<realm>/clients-registrations/default** への HTTP POST 要求を実行します。

登録アクセストークンも含まれる Client Representation を返します。後で設定を取得したり、クライアントを更新または削除したりする場合は、登録アクセストークンをどこかに保存する必要があります。

クライアント表現を取得するには、**/auth/realms/<realm>/clients-registrations/default/<client id>** に対して HTTP GET リクエストを実行します。

また、新しい登録アクセストークンも返します。

クライアント表示を更新するには、更新されたクライアント表示で HTTP PUT 要求を実行します。これは、**/auth/realms/<realm>/clients-registrations/default/<client id>** です。

また、新しい登録アクセストークンも返します。

クライアント表現を削除するには、**/auth/realms/<realm>/clients-registrations/default/<client id>** への HTTP DELETE 要求を実行します。

5.3. RED HAT SINGLE SIGN-ON アダプターの設定

インストール クライアント登録プロバイダーを使用して、クライアントのアダプター設定を取得できます。トークン認証の他に、HTTP Basic 認証を使用してクライアント認証情報で認証することもできます。これには、リクエストに以下のヘッダーが含まれます。

```
Authorization: basic BASE64(client-id + ':' + client-secret)
```

アダプター設定を取得するには、HTTP GET 要求を `/auth/realms/<realm>/clients-registrations/install/<client id>` に実行します。

パブリッククライアントには認証は必要ありません。これは、JavaScript アダプターの場合、上記の URL を使用して、Red Hat Single Sign-On から直接クライアント設定をロードできることを意味します。

5.4. OPENID CONNECT 動的クライアント登録

Red Hat Single Sign-On は、[OAuth 2.0 動的クライアント登録プロトコル](#) および [OAuth 2.0 動的クライアント登録管理プロトコル](#) を拡張する [OpenID Connect 動的クライアント登録](#) を実装します。

Red Hat Single Sign-On でクライアントを登録するために使用するエンドポイントは、`/auth/realms/<realm>/clients-registrations/openid-connect[/<client id>]` です。

このエンドポイントは、レルムの OpenID Connect Discovery エンドポイントである `/auth/realms/<realm>/well-known/openid-configuration` にも含まれています。

5.5. SAML エンティティ記述子

SAML エンティティ記述子エンドポイントは、SAML v2 エンティティ記述子を使用したクライアントの作成のみをサポートします。クライアントの取得、更新、または削除はサポートされません。これらの操作では、Red Hat Single Sign-On 表現エンドポイントを使用する必要があります。クライアントを作成する場合、登録アクセストークンを含む作成されたクライアントに関する詳細が記載された Red Hat Single Sign-On Client Representation が返されます。

クライアントを作成するには、SAML エンティティ記述子を使用して HTTP POST 要求を `/auth/realms/<realm>/clients-registrations/saml2-entity-descriptor` に実行します。

5.6. CURL の使用例

以下の例では、CURL を使用して clientId **myclient** でクライアントを作成します。**eyJhbGciOiJSUz...** を、適切な初期アクセストークンまたはベアータークンに置き換える必要があります。

```
curl -X POST \
  -d '{"clientId": "myclient"}' \
  -H "Content-Type: application/json" \
  -H "Authorization: bearer eyJhbGciOiJSUz..." \
  http://localhost:8080/auth/realms/master/clients-registrations/default
```

5.7. JAVA クライアント登録 API の使用例

クライアント登録 Java API により、Java を使用したクライアント登録サービスが容易になります。使用するには、Maven からの **org.keycloak:keycloak-client-registration-api:>VERSION<** の依存関係を追加します。

クライアント登録の使用に関する詳細な手順は、JavaDocs を参照してください。以下は、クライアントを作成する例です。**eyJhbGciOiJSUz...** を、適切な初期アクセストークンまたはベアータークンに置き換える必要があります。

```
String token = "eyJhbGciOiJSUz...";

ClientRepresentation client = new ClientRepresentation();
client.setClientId(CLIENT_ID);

ClientRegistration reg = ClientRegistration.create()
    .url("http://localhost:8080/auth", "myrealm")
    .build();

reg.auth(Auth.token(token));

client = reg.create(client);

String registrationAccessToken = client.getRegistrationAccessToken();
```

5.8. クライアント登録ポリシー

現在、Red Hat Single Sign-On では、クライアント登録サービスを介して新しいクライアントを登録する 2 つの方法をサポートしています。

- 認証された要求 - 新しいクライアントを登録する要求には、上記のように **Initial Access Token** または **Bearer Token** のいずれかが含まれている必要があります。
- 特定の要求 - 新しいクライアントを登録する要求にトークンを含める必要はありません。

匿名のクライアント登録要求は非常に興味深く強力な機能ですが、誰もが制限なしに新しいクライアントを登録できることは、通常は望まれません。このため、クライアント登録ポリシー **SPI** があります。これは、新しいクライアントを登録できるユーザーとその条件を制限する方法を提供します。

Red Hat Single Sign-On 管理コンソールでは、**Client Registration** タブをクリックして、**Client Registration Policies** サブタブをクリックします。ここでは、匿名要求に対して、デフォルトで設定されているポリシーと、認証された要求に対して設定されているポリシーを確認できます。



注記

匿名要求 (トークンなしの要求) は、新しいクライアントの作成 (登録) のためだけに許可されます。したがって、匿名要求を介して新しいクライアントを登録すると、応答には登録アクセストークンが含まれます。これは、特定のクライアントの読み取り、更新、または削除要求に使用する必要があります。ただし、匿名登録からこの登録アクセストークンを使用する場合も、匿名ポリシーが適用されます。そのため、たとえば、**Trusted Hosts** ポリシーがある場合は、更新クライアントのリクエストも Trusted Host から取得する必要があります。たとえば、クライアントの更新時や、**Consent Required** が存在する場合には、**Consent Required** を無効にすることはできません。

現在、以下のポリシー実装があります。

- Trusted Hosts Policy: 信頼されたホストおよび信頼済みドメインの一覧を設定できます。クライアント登録サービスへの要求は、それらのホストまたはドメインからのみ送信できます。信頼できない IP から送信されたリクエストは拒否されます。新たに登録したクライアントの URL も、信頼できるホストまたはドメインを使用する必要があります。たとえば、信頼されてい

いホストを参照するクライアントの **Redirect URI** を設定することはできません。デフォルトでは、ホワイトリストに登録されているホストがないため、匿名クライアントの登録は事実上無効になっています。

- **Consent Required Policy:** 新しく登録されたクライアントでは、**Consent Allowed** スイッチが有効になります。したがって、認証が成功した後、ユーザーがパーミッション (クライアントスコープ) を承認する必要があるときに、常に同意画面が表示されます。これは、ユーザーが承認しない限り、クライアントが個人情報へのアクセスやユーザーのパーミッションを持たないことを意味します。
- **Protocol Mappers Policy:** ホワイトリスト化されたプロトコルマッパー実装のリストを設定できます。新規クライアントに、ホワイトリストに記載されていないプロトコルマッパーが含まれている場合は、登録や更新を行うことができません。このポリシーは認証されたリクエストにも使用されるため、認証されたリクエストであっても、使用できるプロトコルマッパーに関していくつかの制限がある点に注意してください。
- **Client Scope Policy:** 新しく登録または更新されたクライアントで使用する **Client Scopes** をホワイトリスト化できます。デフォルトではホワイトリスト化されたスコープはありません。デフォルトでは、**Realm Default Client Scopes** として定義されるクライアントスコープのみがホワイトリスト化されます。
- **Full Scope Policy:** 新しく登録されたクライアントでは、**Full Scope Allowed** が無効になります。つまり、指定されたレルムロールや、他のクライアントのクライアントロールはありません。
- **Max Clients Policy:** レルム内の現在のクライアント数が指定の制限と同じかそれより多い場合、登録を拒否します。匿名登録では、デフォルトで 200 になります。
- **Client Disabled Policy:** 新たに登録されたクライアントが無効になります。これは、管理者が新しく登録されたすべてのクライアントを手動で承認して有効にする必要があることを意味します。このポリシーは、匿名登録でもデフォルトで使用されません。

第6章 クライアント登録 CLI

クライアント登録 CLI は、アプリケーション開発者が、Red Hat Single Sign-On と統合する際に、セルフサービス方式で新しいクライアントを設定するためのコマンドラインインターフェイス (CLI) ツールです。これは、Red Hat Single Sign-On クライアント登録 REST エンドポイントと対話する目的で設計されています。

Red Hat Single Sign-On を使用できるようにするには、すべてのアプリケーションでクライアント設定を作成または取得する必要があります。通常、一意のホスト名でホストされる新規アプリケーションごとに新規クライアントを設定します。アプリケーションが Red Hat Single Sign-On と対話する場合、アプリケーションはクライアント ID で自身を識別するため、Red Hat Single Sign-On はログインページ、シングルサインオン (SSO) セッション管理、およびその他のサービスを提供できます。

クライアント登録 CLI を使用してコマンドラインからアプリケーションクライアントを設定できます。また、これをシェルスクリプトで使用できます。

特定のユーザーが **Client Registration CLI** を使用できるようにするために、Red Hat Single Sign-On の管理者は通常、管理コンソールを使用して適切なロールで新規ユーザーを設定するか、新しいクライアントとクライアントシークレットを設定してクライアント登録 REST API へのアクセスを付与します。

6.1. クライアント登録 CLI で使用する新しい一般ユーザーの設定

1. 管理コンソール (例: <http://localhost:8080/auth/admin>) に **admin** としてログインします。
2. 管理するレルムを選択します。
3. 既存のユーザーを使用する場合は、そのユーザーを選択して編集します。それ以外の場合は、新しいユーザーを作成します。
4. **Role Mappings > Client Roles > realm-management** を選択します。マスターレルムを使用している場合は、**NAME-realm** を選択します。**NAME** はターゲットレルムの名前に置き換えます。他のレルムへのアクセスをマスターレルムのユーザーに付与できます。
5. **Available Roles > manage-client** を選択して、完全セットのクライアント管理パーミッションを付与します。別のオプションとして、読み取り専用 **view-clients** または **create-client** を選択して、新規クライアントを作成します。



注記

これらのパーミッションにより、[初期アクセストークン](#) または [登録アクセストークン](#) を使用せずに操作を実行することができます。

realm-management ロールをユーザーに割り当てることはできません。この場合、ユーザーは引き続きクライアント登録 CLI でログインできますが、初期アクセストークンなしでは使用することはできません。トークンなしで操作を実行しようとすると、**403 Forbidden** エラーが発生します。

管理者は、**Realm Settings > Client Registration > Initial Access Token** メニューの Admin Console から、初期アクセストークンを実行できます。

6.2. クライアント登録 CLI で使用するクライアントの設定

デフォルトでは、サーバーはクライアント登録 CLI を **admin-cli** クライアントとして認識します。これは、新しいレلمごとに自動的に設定されます。ユーザー名を使用してログインする場合、追加のクライアント設定は必要ありません。

1. クライアント登録 CLI に別のクライアント設定を使用する場合は、新しいクライアントを作成します (例: **reg-cli**)。
2. **Standard Flow Enabled** の設定を **Off** に切り替えます。
3. クライアントの **Access Type** を **Confidential** として設定し、**Credentials > ClientId and Secret** を選択してセキュリティを強化します。



注記

Credentials タブで **Client Id and Secret** または **Signed JWT** のいずれかを設定できます。

4. 管理コンソールの **Clients** セクションで編集するクライアントを選択して、クライアントに関連付けられたサービスアカウントを使用する場合は、サービスアカウントを有効にします。
 - a. **Settings** で **Access Type** を **Confidential** に変更し、**Service Accounts Enabled** 設定を **On** に切り替えて、**Save** をクリックします。
 - b. **Service Account Roles** をクリックし、必要なロールを選択してサービスアカウントのアクセスを設定します。選択するロールの詳細は、[「クライアント登録 CLI で使用する新しい一般ユーザーの設定」](#)を参照してください。
5. サービスアカウントの代わりに通常のユーザーアカウントを使用する場合は、**Direct Access Grants Enabled** の設定を **On** に切り替えます。
6. クライアントが **Confidential** として設定されている場合は、**--secret** オプションを使用して **kcreg config credentials** を実行する際に設定されたシークレットを指定します。
7. **kcreg config credentials** の実行時に使用する **clientId** (例: **--client reg-cli**)を指定します。
8. サービスアカウントを有効にすると、**kcreg config credentials** の実行時にユーザーの指定を省略し、クライアントシークレットまたはキーストア情報のみを提供します。

6.3. クライアント登録 CLI のインストール

クライアント登録 CLI は Red Hat Single Sign-On Server ディストリビューション内にパッケージ化されています。**bin** ディレクトリー内に実行スクリプトを見つけることができます。Linux スクリプトは **kcreg.sh** と呼ばれ、Windows スクリプトは **kcreg.bat** と呼ばれます。

ファイルシステム上の任意の場所からクライアントを使用できるようにクライアントを設定する際に、Red Hat Single Sign-On サーバーディレクトリーを **PATH** に追加します。

たとえば、以下のようになります。

- Linux:

```
$ export PATH=$PATH:$KEYCLOAK_HOME/bin
$ kcreg.sh
```

- Windows:

```
c:\> set PATH=%PATH%;%KEYCLOAK_HOME%\bin
c:\> kcreg
```

KEYCLOAK_HOME は、Red Hat Single Sign-On Server ディストリビューションが展開されたディレクトリーを指定します。

6.4. クライアント登録 CLI の使用

1. 認証情報を使用してログインし、認証されたセッションを開始します。
2. クライアント登録 **REST** エンドポイントでコマンドを実行します。
たとえば、以下ようになります。

- Linux:

```
$ kcreg.sh config credentials --server http://localhost:8080/auth --realm demo --user user
--client reg-cli
$ kcreg.sh create -s clientId=my_client -s 'redirectUri=["http://localhost:8980/myapp/*"]'
$ kcreg.sh get my_client
```

- Windows:

```
c:\> kcreg config credentials --server http://localhost:8080/auth --realm demo --user user
--client reg-cli
c:\> kcreg create -s clientId=my_client -s "redirectUri=["http://localhost:8980/myapp/*"]"
c:\> kcreg get my_client
```



注記

実稼働環境では、**https:** で Red Hat Single Sign-On にアクセスする必要があります。これにより、トークンをネットワークスニファーに公開しないようにする必要があります。

3. Java のデフォルト証明書トラストストアに含まれる信頼される認証局 (CA) のいずれかによってサーバーの証明書が発行されていない場合は、**truststore.jks** ファイルを準備し、クライアント登録 CLI にこれを使用するように指示します。
たとえば、以下ようになります。

- Linux:

```
$ kcreg.sh config truststore --trustpass $PASSWORD ~/.keycloak/truststore.jks
```

- Windows:

```
c:\> kcreg config truststore --trustpass %PASSWORD%
%HOMEPATH%\keycloak\truststore.jks
```

6.4.1. ログイン

1. クライアント登録 CLI でログインするときに、サーバーエンドポイント URL およびレルムを指定します。

2. ユーザー名またはクライアント ID を指定します。これにより、特別なサービスアカウントが使用されます。ユーザー名を使用する場合は、指定したユーザーのパスワードを使用する必要があります。クライアント ID を使用する場合は、パスワードの代わりにクライアントシークレットまたは署名済み **JWT** を使用します。

ログイン方法に関係なく、ログインするアカウントには、クライアント登録操作を実行するための適切な権限が必要です。マスター以外のレルムのアカウントには、同じレルム内のクライアントを管理するためのパーミッションのみを持つことができる点に注意してください。異なるレルムを管理する必要がある場合は、異なるレルムで複数のユーザーを設定するか、**master** レルムで1つのユーザーを作成し、異なるレルムでクライアントを管理するロールを追加できます。

クライアント登録 CLI を使用してユーザーを設定することはできません。管理コンソールの Web インターフェースまたは Admin Client CLI を使用してユーザーを設定します。詳細は、『[サーバー管理ガイド](#)』を参照してください。

kcreg が正常にログインすると、認証トークンを受け取り、プライベート設定ファイルに保存します。これにより、後続の呼び出しにトークンを使用できます。設定ファイルの詳細は、『[代替設定の使用](#)』を参照してください。

クライアント登録 CLI の使用方法についての詳細は、組み込みヘルプを参照してください。

たとえば、以下のようになります。

- Linux:

```
$ kcreg.sh help
```

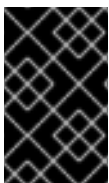
- Windows:

```
c:\> kcreg help
```

認証セッションの開始に関する詳細は、**kcreg config credentials --help** を参照してください。

6.4.2. 代替設定の使用

デフォルトでは、クライアント登録 CLI は、ユーザーのホームディレクトリーにあるデフォルトの場所 (`./keycloak/kcreg.config`) で設定ファイルを自動的に維持します。**--config** オプションを使用して、異なるファイルまたは場所を指定して、複数の認証セッションを並行して管理することができます。単一のスレッドから、単一の設定ファイルに関連付けられる操作を実行するための最も安全な方法です。



重要

システム上の他のユーザーには、設定ファイルを表示しないでください。設定ファイルには、非公開にしておく必要のあるアクセストークンとシークレットが含まれています。

すべてのコマンドで **--no-config** オプションを使用すると、シークレットを設定ファイル内に格納しないようにすることができます。各 **kcreg** 呼び出しですべての認証情報を指定します。

6.4.3. 初期アクセスおよび登録アクセストークン

使用したい Red Hat Single Sign-On サーバーでアカウントを設定していない開発者は、クライアント登録 CLI を使用できます。これは、レルム管理者が開発者に初期アクセストークンを発行した場合にのみ可能です。これらのトークンを発行および配布する方法と時期を決定するのは、レルム管理者の責任に

なります。レルム管理者は、初期アクセストークンの最大有効期間と、それを使用して作成できるクライアントの総数を制限できます。

開発者に初期アクセストークンがある場合、開発者は **kcreg config credentials** で認証せずにこれを使用して新規クライアントを作成することができます。初期アクセストークンは、設定ファイルに保存するか、**kcreg create** コマンドの一部として指定できます。

たとえば、以下のようになります。

- Linux:

```
$ kcreg.sh config initial-token $TOKEN
$ kcreg.sh create -s clientId=myclient
```

または

```
$ kcreg.sh create -s clientId=myclient -t $TOKEN
```

- Windows:

```
c:\> kcreg config initial-token %TOKEN%
c:\> kcreg create -s clientId=myclient
```

または

```
c:\> kcreg create -s clientId=myclient -t %TOKEN%
```

初期アクセストークンを使用する場合、サーバーの応答には、新しく発行された登録アクセストークンが含まれます。そのクライアントの後続の操作は、そのトークンで認証することで実行する必要があります。これは、そのクライアントに対してのみ有効です。

クライアント登録 CLI は、プライベート設定ファイルを自動的に使用して、このトークンを保存し、関連付けられたクライアントで使用します。同じ設定ファイルがすべてのクライアント操作に使用される限り、開発者はこの方法で作成されたクライアントの読み取り、更新、または削除を行うために認証する必要はありません。

初期アクセスおよび登録アクセストークンの詳細は、[「クライアント登録」](#)を参照してください。

クライアント登録 CLI でトークンを設定する方法についての詳細は、**kcreg config initial-token --help** コマンドおよび **kcreg config registration-token --help** コマンドを実行してください。

6.4.4. クライアント設定の作成

通常、認証情報を使用して認証するか、初期アクセストークンを設定した後の最初のタスクは、新しいクライアントを作成することです。多くの場合、準備した JSON ファイルをテンプレートとして使用し、一部の属性を設定したり、上書きすることをお勧めします。

以下の例は、JSON ファイルの読み取り、含まれる可能性のあるクライアント ID の上書き、その他の属性の設定、正常な作成後に設定を標準出力に出力する方法を示しています。

- Linux:

```
$ kcreg.sh create -f client-template.json -s clientId=myclient -s baseUrl=/myclient -s 'redirectUri=
["/myclient/*"]' -o
```

- Windows:

```
C:\> kcreg create -f client-template.json -s clientId=myclient -s baseUrl=/myclient -s "redirectUris=[\"/myclient/*\"]" -o
```

kcreg create コマンドの詳細は、**kcreg create --help** を実行します。

kcreg attrs を使用して利用可能な属性を一覧表示できます。多くの設定属性は、有効性または一貫性についてチェックされないことに注意してください。適切な値を指定することが推奨されます。テンプレートに id フィールドを使用せず、**kcreg create** コマンドに引数として指定しないでください。

6.4.5. クライアント設定の取得

kcreg get コマンドを使用して、既存のクライアントを取得できます。

たとえば、以下のようになります。

- Linux:

```
$ kcreg.sh get myclient
```

- Windows:

```
C:\> kcreg get myclient
```

クライアント設定をアダプター設定ファイルとして取得することもできます。これは Web アプリケーションでパッケージ化できます。

たとえば、以下のようになります。

- Linux:

```
$ kcreg.sh get myclient -e install > keycloak.json
```

- Windows:

```
C:\> kcreg get myclient -e install > keycloak.json
```

kcreg get コマンドの詳細は、**kcreg get --help** コマンドを実行してください。

6.4.6. クライアント設定の変更

クライアント設定の更新方法は 2 つあります。

1つの方法は、現在の設定を取得してファイルに保存し、編集してサーバーにポストバックした後に、完全に新しい状態をサーバーに送信することです。

たとえば、以下のようになります。

- Linux:

```
$ kcreg.sh get myclient > myclient.json
$ vi myclient.json
$ kcreg.sh update myclient -f myclient.json
```

- Windows:

```
C:\> kcreg get myclient > myclient.json
C:\> notepad myclient.json
C:\> kcreg update myclient -f myclient.json
```

2つ目の方法は、現在のクライアントをフェッチし、そのクライアントのフィールドを設定または削除して、1つのステップでポストバックします。

たとえば、以下ようになります。

- Linux:

```
$ kcreg.sh update myclient -s enabled=false -d redirectUri
```

- Windows:

```
C:\> kcreg update myclient -s enabled=false -d redirectUri
```

適用する変更のみを含むファイルを使用することもできるため、引数として値を多く指定する必要はありません。この場合は、**--merge** を指定して、JSON ファイルを完全かつ新規の設定として処理するのではなく、既存の設定に適用する属性セットとして処理する必要があることをクライアント登録 CLI に指示します。

たとえば、以下ようになります。

- Linux:

```
$ kcreg.sh update myclient --merge -d redirectUri -f mychanges.json
```

- Windows:

```
C:\> kcreg update myclient --merge -d redirectUri -f mychanges.json
```

kcreg update コマンドの詳細は、**kcreg update --help** コマンドを実行してください。

6.4.7. クライアント設定の削除

クライアントを削除するには、以下の例を使用します。

- Linux:

```
$ kcreg.sh delete myclient
```

- Windows:

```
C:\> kcreg delete myclient
```

kcreg delete コマンドの詳細は、**kcreg delete --help** コマンドを実行してください。

6.4.8. 無効な登録アクセストークンの更新

--no-config モードを使用して作成、読み取り、更新、および削除 (CRUD) 操作を実行すると、クライアント登録 CLI はユーザーの登録アクセストークンを処理できません。この場合、クライアントに対して最近発行した Registration Access Token の追跡できなくなる可能性があり、**manage-clients** パーミッションを持つアカウントで認証を行わずに、そのクライアントでの CRUD 操作を追加で実行できなくなります。

パーミッションがある場合は、クライアントの新しい登録アクセストークンを発行し、標準出力に出力したり、任意の設定ファイルに保存したりできます。それ以外の場合は、レルム管理者に、クライアント用の新しい登録アクセストークンを発行して送信するよう依頼する必要があります。**--token** オプションを使用して、これをすべての CRUD コマンドに渡すことができます。**kcreg config registration-token** コマンドを使用して新規トークンを設定ファイルに保存し、クライアント登録 CLI にこの時点から自動的に処理させることもできます。

kcreg update-token コマンドの詳細は、**kcreg update-token --help** コマンドを実行してください。

6.5. トラブルシューティング

- Q: ログイン時にエラーが表示されます: **Parameter client_assertion_type is missing [invalid_client]**
A: このエラーは、クライアントが **Signed JWT** トークン認証情報で設定されていることを意味します。つまり、ログイン時に **--keystore** パラメーターを使用する必要があります。

第7章 トークンの交換



注記

トークンの交換は テクノロジープレビュー であるため、完全にサポートされていません。この機能はデフォルトでは無効になっています。

-Dkeycloak.profile=preview または **-Dkeycloak.profile.feature.token_exchange=enabled** でサーバーの起動を有効にするには、以下を行います、詳細は「[プロファイル](#)」を参照してください。



注記

トークンの交換を使用するには、**admin_fine_grained_authz** 機能も有効にする必要があります。[プロファイル](#)を確認してください。

Red Hat Single Sign-On では、トークン交換は、一連の認証情報またはトークンを使用して、まったく異なるトークンを取得するプロセスです。クライアントは、信頼性の低いアプリケーションで呼び出したい場合があるため、現在のトークンをダウングレードしたい場合があります。クライアントは、Red Hat Single Sign-On トークンをリンクされたソーシャルプロバイダーアカウント用に保存されているトークンと交換したい場合があります。他の Red Hat Single Sign-On レルムまたは外部 IDP によって作成された外部トークンを信頼することをお勧めします。クライアントでは、ユーザーの権限を借用しなければならない場合があります。以下は、トークン交換に関する Red Hat Single Sign-On の現在の機能の概要です。

- クライアントは、特定のクライアント用に作成された既存の Red Hat Single Sign-On トークンを、別のクライアントを対象とする新しいトークンと交換できます。
- クライアントは、既存の Red Hat Single Sign-On トークンを外部トークン (リンクされた Facebook アカウントなど) と交換できます。
- クライアントは、外部トークンを、Red Hat Single Sign-On トークンと交換できます。
- クライアントはユーザーの権限を借用できます。

Red Hat Single Sign-On でのトークン交換は、IETF での [OAuth トークン交換](#) 仕様の実装は非常に緩やかです。この仕様を少し拡張し、一部を無視して、他の部分は大まかに解釈しました。これは、レルムの OpenID Connect トークンエンドポイントにおける単純な付与タイプ呼び出しです。

```
/auth/realms/{realm}/protocol/openid-connect/token
```

これは、フォームパラメーター (**application/x-www-form-urlencoded**) を入力として受け付け、出力は交換を要求したトークンのタイプによって異なります。トークン交換はクライアントエンドポイントであるため、リクエストは呼び出し元のクライアントに対して認証情報を提供する必要があります。パブリッククライアントは、クライアント識別子を form パラメーターとして指定します。機密クライアントは、フォームパラメータを使用して、クライアント ID とシークレット、Basic 認証、または管理者がレルムで設定したクライアント認証フローを渡すこともできます。以下は、フォームパラメーターの一覧です。

client_id

必須の可能性あり。このパラメーターは、認証にフォームパラメーターを使用するクライアントに必要です。Basic 認証、クライアント JWT トークン、またはクライアント証明書認証を使用している場合は、このパラメーターを指定しないでください。

client_secret

必須の可能性あり。このパラメーターは、認証にフォームパラメーターを使用し、認証情報としてクライアントシークレットを使用するクライアントに必要です。レルムのクライアント呼び出しが別の手段で認証されている場合は、このパラメーターは指定しないでください。

grant_type

必須。パラメーターの値は **urn:ietf:params:oauth:grant-type:token-exchange** にする必要があります。

subject_token

任意です。リクエストが行われている当事者の ID を表すセキュリティトークン。これは、既存のトークンを新しいトークンと交換する場合に必要です。

subject_issuer

任意です。**subject_token** の発行者を特定します。トークンが現在のレルムから送信される場合や、発行者が **subject_token_type** から判断できる場合は空白のままにすることができます。それ以外の場合は指定する必要があります。有効な値は、レルムに設定された ID プロバイダーのエイリアスです。または、特定の ID プロバイダーによって設定された発行者要求 ID です。

subject_token_type

任意です。このパラメーターは、**subject_token** パラメーターで渡されるトークンのタイプです。**subject_token** がレルムから取得され、アクセストークンである場合、デフォルトは **urn:ietf:params:oauth:token-type:access_token** に設定されます。外部トークンの場合には、**subject_issuer** の要件に応じてこのパラメーターを指定する必要がある場合とそうでない場合があります。

requested_token_type

任意です。このパラメーターは、クライアントが交換するトークンのタイプを表します。現在、oauth および OpenID Connect トークンタイプのみがサポートされます。このデフォルト値が **urn:ietf:params:oauth:token-type:refresh_token** であるかどうかによって異なります。この場合、応答内でアクセストークンと更新トークンの両方が返されます。その他の適切な値は、**urn:ietf:params:oauth:token-type:access_token** および **urn:ietf:params:oauth:token-type:id_token** です。

audience

任意です。このパラメーターは、新しいトークンを作成するターゲットクライアントを指定します。

requested_issuer

任意です。このパラメーターは、クライアントが外部プロバイダーによって作成されたトークンを必要とすることを指定します。レルム内で設定された ID プロバイダーのエイリアスである必要があります。

requested_subject

任意です。これは、クライアントが別のユーザーになりすます場合のユーザー名またはユーザー ID を指定します。

scope

実装ない。このパラメーターは、クライアントが要求する OAuth および OpenID Connect スコープのターゲットセットを表します。現時点では実装されていませんが、Red Hat Single Sign-On がスコープ全般をより適切にサポートするようになると実装されます。



注記

現在、OpenID Connect および OAuth の交換のみをサポートします。SAML ベースのクライアントおよびアイデンティティプロバイダーのサポートは、ユーザーの需要に応じて今後追加される可能性があります。

交換呼び出しからの正常な応答は、クライアントが要求する **requested-token-type** と

requested_issuer に依存するコンテンツタイプを含む HTTP 200 応答コードを返します。OAuth が要求するトークンタイプは、[OAuth Token Exchange](#) 仕様説明されているように JSON ドキュメントを返します。

```
{
  "access_token" : "....",
  "refresh_token" : "....",
  "expires_in" : "...."
}
```

更新トークンを要求するクライアントは、応答でアクセストークンと更新トークンの両方を返します。アクセストークンタイプのみを要求するクライアントは、応答でアクセストークンのみを取得します。**requested_issuer** パラメーターを介して外部発行者を要求するクライアントの有効期限情報は、含まれる場合とそうでない場合があります。

通常、エラー応答は 400 HTTP 応答コードカテゴリに分類されますが、エラーの重大度に応じて他のエラーステータスコードが返される場合があります。エラー応答には、**requested_issuer** に依存するコンテンツが含まれる場合があります。OAuth ベースの交換では、以下のように JSON ドキュメントが返される場合があります。

```
{
  "error" : "...."
  "error_description" : "...."
}
```

交換タイプに応じて、追加のエラークレームを返すことができます。たとえば、ユーザーに ID プロバイダーへのリンクがない場合に、OAuth ID プロバイダーには追加の **account-link-url** 要求が含まれる場合があります。このリンクは、クライアントが開始したリンク要求に使用できます。



注記

トークンの交換の設定には、きめ細かな管理パーミッションに関する知識が必要です (詳細は、『[サーバー管理ガイド](#)』を参照)。交換にクライアントパーミッションを付与する必要があります。これについては、この章で後ほど説明します。

この章の残りの部分では、セットアップ要件について説明し、さまざまな交換シナリオの例を示します。分かりやすくするために、現在のレلمで作成された最小のトークンを 内部 トークンとして呼び出し、外部レلمまたは ID プロバイダーによって作成されたトークンを 外部 トークンと呼ぶことにします。

7.1. 内部トークンから内部トークンへの交換

内部のトークンからトークンへの交換では、特定のクライアントに作成された既存のトークンがあり、このトークンを別のターゲットクライアントに作成された新しいトークンと交換する必要があります。これを実行する理由これは通常、クライアントが自身のために作成したトークンを持っており、アクセストークン内で異なるクレームとパーミッションを必要とする他のアプリケーションに追加の要求を行う必要がある場合に発生します。このタイプの交換が必要となるかもしれないその他の理由は、アプリが信頼性の低いアプリで呼び出す必要があり、現在のアクセストークンを伝播したくない場合の「パーミッションのダウングレード」を実行する必要がある場合です。

7.1.1. Exchange のパーミッションの付与

別のクライアントのトークンを交換する必要のあるクライアントは、管理コンソールで承認される必要があります。交換するパーミッションを付与するクライアントに、**token-exchange** の細かいパーミッションを定義する必要があります。

ターゲットクライアントパーミッション

The screenshot shows the 'Target-client' configuration page in the Red Hat Single Sign-On console. The left sidebar contains a navigation menu with 'Master' at the top, followed by 'Configure' (with sub-items: Realm Settings, Clients, Client Scopes, Roles) and 'Manage' (with sub-items: Groups, Users, Sessions, Events, Import, Export). The main content area shows the 'Clients > target-client' breadcrumb, the 'Target-client' title with a trash icon, and a set of tabs: Settings, Roles, Client Scopes, Mappers, Scope, Revocation, and Sessions. The 'Permissions' tab is selected. Below the tabs, there is a 'Permissions Enabled' section with a toggle switch currently set to 'OFF'.

Permissions Enabled スイッチを ON に切り替えます。

ターゲットクライアントパーミッション

The screenshot shows the 'Target-client' configuration page with the 'Permissions Enabled' switch turned ON. Below the switch, a table lists the permissions for this client.

scope-name	Description	Actions
view	Policies that decide if an administrator can view this client	Edit
manage	Policies that decide if an administrator can manage this client	Edit
configure	Reduced management permissions for administrator. Cannot set scope, template, or protocol mappers.	Edit
map-roles	Policies that decide if an administrator can map roles defined by this client	Edit
map-roles-client-scope	Policies that decide if an administrator can apply roles defined by this client to the client scope of another client	Edit
map-roles-composite	Policies that decide if an administrator can apply roles defined by this client as a composite to another role	Edit
token-exchange	Policies that decide which clients are allowed exchange tokens for a token that is targeted to this client.	Edit

ページに **token-exchange** リンクが表示されます。それをクリックして、パーミッションの定義を開始します。このページが作成されます。

ターゲットクライアント交換のパーミッション設定

Master
▼

Configure

- Realm Settings
- Clients**
- Client Scopes
- Roles
- Identity Providers
- User Federation
- Authentication

Manage

- Groups
- Users
- Sessions
- Events
- Import
- Export

Clients > master-realm > Authorization > Permissions > token-exchange.permission.client.930a9653-8635-4fae-8465-921320ea1425

Token-exchange.permission.client.930a9653-8635-4fae-8465-921320ea1425

Name * ?

Description ?

Resource ?

Scopes * ?

Apply Policy ?

Decision Strategy ?

このパーミッションのポリシーを定義する必要があります。**Authorization** リンクをクリックし、**Policies** タブに移動し、**Client** ポリシーを作成します。

クライアントポリシーの作成

Master
▼

Configure

- Realm Settings
- Clients**
- Client Scopes
- Roles
- Identity
- Providers
- User Federation
- Authentication

Manage

- Groups
- Users
- Sessions
- Events
- Import
- Export

Clients > master-realm > Authorization > Policies > Add Client Policy

Add Client Policy

Name * ?

Description ?

Clients * ?

clientId	Actions
starting-client	<input type="button" value="Remove"/>

Logic ?

ここでは、トークン交換を要求する認証されたクライアントである、開始クライアントを入力します。このポリシーを作成したら、ターゲットクライアントの **token-exchange** パーミッションに戻り、定義したクライアントポリシーを追加します。

クライアントポリシーの適用

Master

Configure

- Realm Settings
- Clients
- Client Scopes
- Roles
- Identity Providers
- User Federation
- Authentication

Manage

- Groups
- Users
- Sessions
- Events
- Import
- Export

Clients > master-realm > Authorization > Permissions > token-exchange.permission.client.930a9653-8635-4fae-8465-921320ea1425

Token-exchange.permission.client.930a9653-8635-4fae-8465-921320ea1425

token-exchange.permission.client.930a9653-8635-4fae-8

Name *

Description

Resource

client.resource.930a9653-8635-4fae-8465-921320ea1425

Scopes *

token-exchange

Select existing policy... Create Policy...

No policies assigned.

Apply Policy

Unanimous Decision Strategy

Save Cancel

これでクライアントを呼び出すパーミッションがある。これを正しく行わないと、エクスチェンジを作成しようとすると、403 Forbidden 応答が返されます。

7.1.2. リクエストの作成

クライアントが、既存のトークンを他のクライアントをターゲットにしたトークンと交換する場合は、**audience** パラメーターを使用する必要があります。このパラメーターは、管理コンソールで設定したターゲットクライアントのクライアント識別子である必要があります。

```
curl -X POST \
  -d "client_id=starting-client" \
  -d "client_secret=geheim" \
  --data-urlencode "grant_type=urn:ietf:params:oauth:grant-type:token-exchange" \
  -d "subject_token=..." \
  --data-urlencode "requested_token_type=urn:ietf:params:oauth:token-type:refresh_token" \
  -d "audience=target-client" \
  http://localhost:8080/auth/realms/myrealm/protocol/openid-connect/token
```

subject_token パラメーターは、ターゲットレルムのアクセストークンである必要があります。**requested_token_type** パラメーターが更新トークンタイプである場合、応答にはアクセストークン、更新トークン、および有効期限が含まれます。以下は、この呼び出しから返される JSON 応答の例です。

```
{
  "access_token": "...",
  "refresh_token": "...",
  "expires_in": 3600
}
```

7.2. 内部トークンから外部トークンへの交換

レルムトークンを、外部 ID プロバイダーによって作成された外部トークンと交換できます。この外部 ID プロバイダーは、管理コンソールの **ID** プロバイダー セクション内で設定する必要があります。現時点で、OAuth/OpenID Connect ベースの外部 ID プロバイダーのみがサポートされます。これには、すべてのソーシャルプロバイダーが含まれます。Red Hat Single Sign-On は、外部プロバイダーへのバックチャネル交換を実行しません。そのため、アカウントがリンクされていない場合は、外部トークンを取得できなくなります。これらの条件の1つの外部トークンを取得できるようにするには、以下の条件を満たしている必要があります。

- ユーザーは、少なくとも1回、外部 ID プロバイダーを使用してログインしている必要があります。
- ユーザーは、ユーザーアカウントサービスを使用して、外部 ID プロバイダーとリンクされている必要があります。
- このユーザーアカウントは、「[クライアント開始アカウントリンク](#)」API を使用して外部の ID プロバイダー経由でリンクされました。

最後に、外部 ID プロバイダーがトークンを保存するように設定されている必要があります。または、上記のアクションの1つが、交換する内部トークンと同じユーザーセッションで実行されている必要があります。

アカウントがリンクされていない場合、交換応答には、アカウントを確立するために使用できるリンクが含まれます。詳細は、「[リクエストの作成](#)」セクションで説明されています。

7.2.1. Exchange のパーミッションの付与

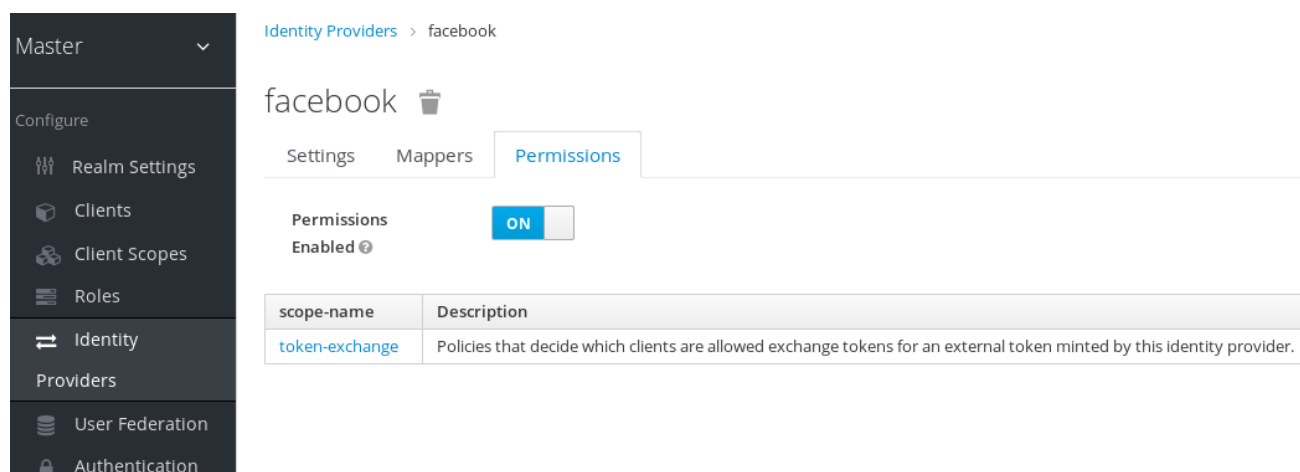
内部から外部へのトークン交換要求は、外部 ID プロバイダーとトークンを交換できるパーミッションを呼び出し元のクライアントに付与するまで、403、Forbidden 応答で拒否されます。クライアントにパーミッションを付与するには、ID プロバイダーの設定ページの **Permissions** タブに移動する必要があります。

ID プロバイダーパーミッション

The screenshot shows the 'Identity Providers' configuration page for 'facebook'. The left sidebar contains a menu with options: Master, Configure, Realm Settings, Clients, Client Scopes, Roles, Identity Providers (selected), User Federation, and Authentication. The main content area shows the 'facebook' provider with tabs for Settings, Mappers, and Permissions (selected). Under the 'Permissions' tab, there is a section 'Permissions Enabled' with a toggle switch currently set to 'OFF'.

Permissions Enabled スイッチを true に切り替えます。

ID プロバイダーパーミッション



Identity Providers > facebook

facebook

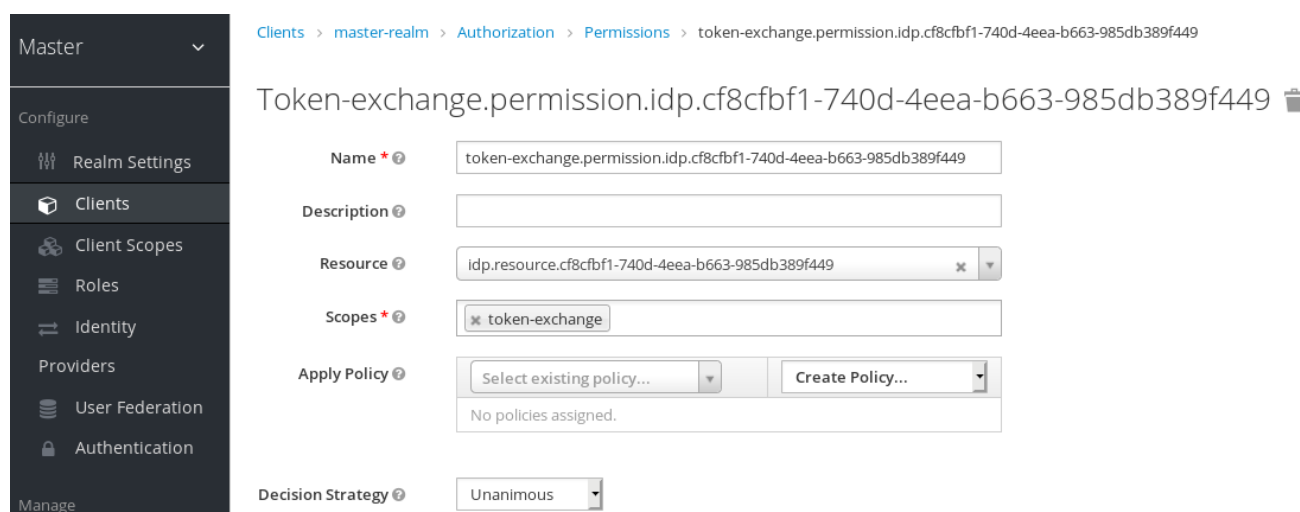
Settings Mappers Permissions

Permissions Enabled ☒

scope-name	Description
token-exchange	Policies that decide which clients are allowed exchange tokens for an external token minted by this identity provider.

ページに **token-exchange** リンクが表示されます。それをクリックして、パーミッションの定義を開始します。このページが作成されます。

アイデンティティプロバイダー交換のパーミッション設定



Clients > master-realm > Authorization > Permissions > token-exchange.permission.idp.cf8cfbf1-740d-4eea-b663-985db389f449

Token-exchange.permission.idp.cf8cfbf1-740d-4eea-b663-985db389f449

Name * token-exchange.permission.idp.cf8cfbf1-740d-4eea-b663-985db389f449

Description

Resource idp.resource.cf8cfbf1-740d-4eea-b663-985db389f449

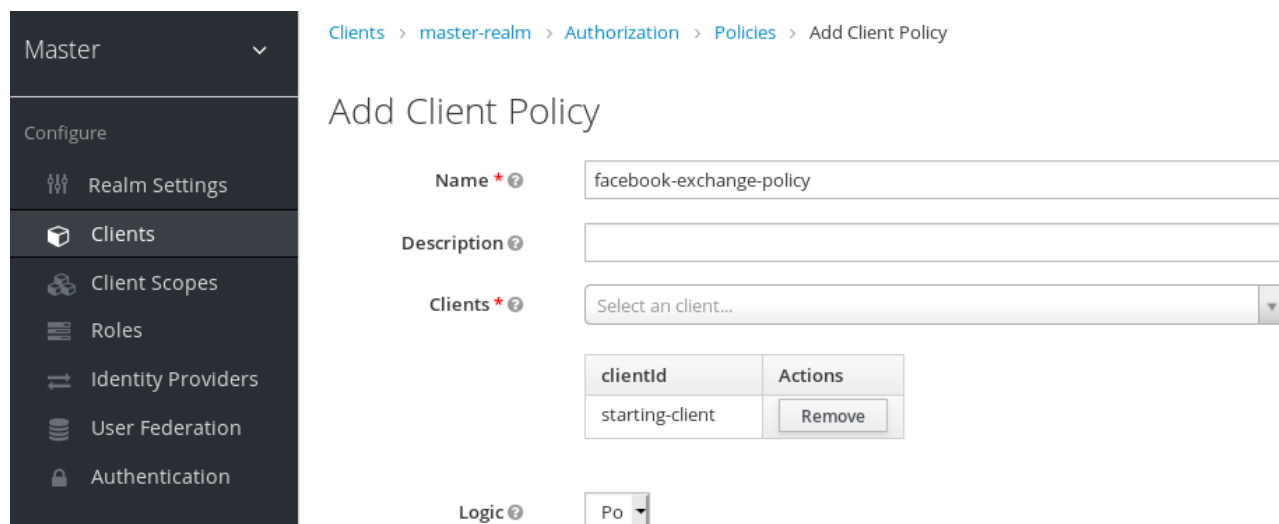
Scopes * token-exchange

Apply Policy Select existing policy... Create Policy... No policies assigned.

Decision Strategy Unanimous

このパーミッションのポリシーを定義する必要があります。**Authorization** リンクをクリックし、**Policies** タブに移動し、**Client** ポリシーを作成します。

クライアントポリシーの作成



Clients > master-realm > Authorization > Policies > Add Client Policy

Add Client Policy

Name * facebook-exchange-policy

Description

Clients * Select an client...

clientId	Actions
starting-client	Remove

Logic Po

ここでは、トークン交換を要求する認証されたクライアントである、開始クライアントを入力します。このポリシーを作成したら、ID プロバイダーの **token-exchange** パーミッションに戻り、定義したクライアントポリシーを追加します。

クライアントポリシーの適用

Token-exchange.permission.client.b38f5216-8f0b-436f-9c77-07b1156b875e

Name *

Description

Resource

Scopes *

Apply Policy

Name	Description	Actions
facebook-exchange-policy		Remove

これでクライアントを呼び出すパーミッションがある。これを正しく行わないと、エクスチェンジを作成しようとする、403 Forbidden 応答が返されます。

7.2.2. リクエストの作成

クライアントが既存の内部トークンを外部に交換する場合には、**requested_issuer** パラメーターを指定する必要があります。このパラメーターは、設定済みのアイデンティティプロバイダーのエイリアスである必要があります。

```
curl -X POST \
  -d "client_id=starting-client" \
  -d "client_secret=geheim" \
  --data-urlencode "grant_type=urn:ietf:params:oauth:grant-type:token-exchange" \
  -d "subject_token=..." \
  --data-urlencode "requested_token_type=urn:ietf:params:oauth:token-type:access_token" \
  -d "requested_issuer=google" \
  http://localhost:8080/auth/realms/myrealm/protocol/openid-connect/token
```

subject_token パラメーターは、ターゲットレルムのアクセストークンである必要があります。**requested_token_type** パラメーターは **urn:ietf:params:oauth:token-type:access_token** であるか、空欄のままにする必要があります。現時点では、他の要求されたトークンタイプはサポートされません。以下は、この呼び出しから返される正常な JSON 応答の例です。

```
{
  "access_token": "...",
  "expires_in": 3600
  "account-link-url": "https://..."
}
```

外部 ID プロバイダーが何らかの理由でリンクされていない場合、以下の JSON ドキュメントで HTTP 400 応答コードが表示されます。

```
{
  "error": "...",
  "error_description": "..."
}
```

```
"account-link-url" : "https://..."
```

```
}
```

error 要求は、**token_expired** または **not_linked** のいずれかになります。クライアントが「[クライアント開始アカウントリンク](#)」を実行できるように、**account-link-url** 要求が提供されます。ほとんどの(またはすべての) プロバイダーは、ブラウザー OAuth プロトコルを使用したリンクを必要とします。**account-link-url** では、単に **redirect_uri** クエリーパラメーターを追加し、ブラウザーに転送してリンクを実行できます。

7.3. 外部トークンから内部トークンへの交換

内部トークンの外部 ID プロバイダーが作成した外部トークンを信頼し、交換できます。これはレルム間をブリッジしたり、ソーシャルプロバイダーからのトークンを信頼したりするために使用できます。新しいユーザーが存在しない場合はレルムにインポートされるという点で、ID プロバイダーのブラウザーログインと同様に機能します。



注記

外部トークン交換に関する現在の制限として、外部トークンが既存のユーザーにマップされている場合、既存のユーザーが外部 ID プロバイダーへのアカウントリンクを既に持っていない限り、交換が許可されない点があります。

交換が完了すると、レルム内にユーザーセッションが作成され、**requested_token_type** パラメーターの値に応じてアクセストークンおよび更新トークンが送信されます。この新しいユーザーセッションは、タイムアウトするまで、またはこの新しいアクセストークンを渡すレルムのログアウトエンドポイントを呼び出すまで、アクティブなままである点に注意してください。

これらのタイプの変更には、管理コンソールで構成済みのアイデンティティプロバイダーが必要でした。



注記

SAML アイデンティティプロバイダーは現時点ではサポートされていません。Twitter トークンも交換できません。

7.3.1. Exchange のパーミッションの付与

外部トークン交換を行う前に、呼び出し元のクライアントに交換を行うためのパーミッションを与える必要があります。このパーミッションは、[内部から外部へのパーミッションが付与されている](#) のと同じ方法で付与されます。

呼び出し以外のクライアントを参照する値を持つ **audience** パラメーターを指定する場合、**audience** パラメーターに固有のターゲットクライアントに交換するための呼び出しクライアントのパーミッションも付与する必要があります。これを行う方法は、このセクションで [上述](#) されています。

7.3.2. リクエストの作成

subject_token_type は、**urn:ietf:params:oauth:token-type:access_token** または **urn:ietf:params:oauth:token-type:jwt** のいずれかである必要があります。タイプが **urn:ietf:params:oauth:token-type:access_token** の場合は、**subject_issuer** パラメーターを指定する必要があります。これは設定された ID プロバイダーのエイリアスでなければなりません。タイプが **urn:ietf:params:oauth:token-type:jwt** の場合、プロバイダーは、プロバイダーのエイリアスである JWT 内の発行者 要求、またはプロバイダー設定の登録済み発行者を介して照合されます。

検証のために、トークンがアクセストークンである場合、プロバイダーのユーザー情報サービスが呼び出されてトークンが検証されます。呼び出しに成功すると、アクセストークンが有効であることを意味します。サブジェクトトークンが JWT であり、プロバイダーで署名検証が有効になっている場合は、それが試行されます。それ以外の場合は、デフォルトでユーザー情報サービスも呼び出してトークンを検証します。

デフォルトでは、作成された内部トークンは、呼び出し元のクライアントを使用して、呼び出し元のクライアント用に定義されたプロトコルマッパーを使用してトークンの内容を判別します。または、**audience** パラメーターを使用して別のターゲットクライアントを指定することもできます。

```
curl -X POST \
  -d "client_id=starting-client" \
  -d "client_secret=geheim" \
  --data-urlencode "grant_type=urn:ietf:params:oauth:grant-type:token-exchange" \
  -d "subject_token=..." \
  -d "subject_issuer=myOidcProvider" \
  --data-urlencode "subject_token_type=urn:ietf:params:oauth:token-type:access_token" \
  -d "audience=target-client" \
  http://localhost:8080/auth/realms/myrealm/protocol/openid-connect/token
```

requested_token_type パラメーターが更新トークンタイプである場合、応答にはアクセストークン、更新トークン、および有効期限が含まれます。以下は、この呼び出しから返される JSON 応答の例です。

```
{
  "access_token": "...",
  "refresh_token": "...",
  "expires_in": 3600
}
```

7.4. 権限借用

内部トークンと外部トークン交換の場合、クライアントは、ユーザーに代わって別のユーザーになりますように要求できます。たとえば、サポートエンジニアが問題をデバッグできるように、ユーザーになります必要がある管理アプリケーションがある場合などです。

7.4.1. Exchange のパーミッションの付与

サブジェクトトークンが表すユーザーには、他のユーザーになりすますためのパーミッションが必要です。このパーミッションを有効にする方法は、『[サーバー管理ガイド](#)』を参照してください。これは、ロールまたは粒度の細かい管理者権限を介して実行できます。

7.4.2. リクエストの作成

requested_subject パラメーターを追加で指定しない限り、他の章の説明どおりに要求を行います。このパラメーターの値は、ユーザー名またはユーザー ID である必要があります。

```
curl -X POST \
  -d "client_id=starting-client" \
  -d "client_secret=geheim" \
  --data-urlencode "grant_type=urn:ietf:params:oauth:grant-type:token-exchange" \
  -d "subject_token=..." \
  --data-urlencode "requested_token_type=urn:ietf:params:oauth:token-type:access_token" \
```

```
-d "audience=target-client" \  
-d "requested_subject=wburke" \  
http://localhost:8080/auth/realms/myrealm/protocol/openid-connect/token
```

7.5. ダイレクトの NAKED IMPERSONATION

subject_token を指定せずに内部トークン交換要求を行うことができます。これは、クライアントがレルム内の任意のユーザーになりすますことができるため、クライアントに多くの信頼を置くことから、ダイレクトの Naked Impersonation と呼ばれます。交換するサブジェクトトークンを取得できないアプリケーションをブリッジするために、これが必要になる場合があります。たとえば、LDAP と直接ログインを実行するレガシーアプリケーションを統合している場合があります。この場合、レガシーアプリケーションはユーザー自身を認証できますが、トークンを取得することはできません。



警告

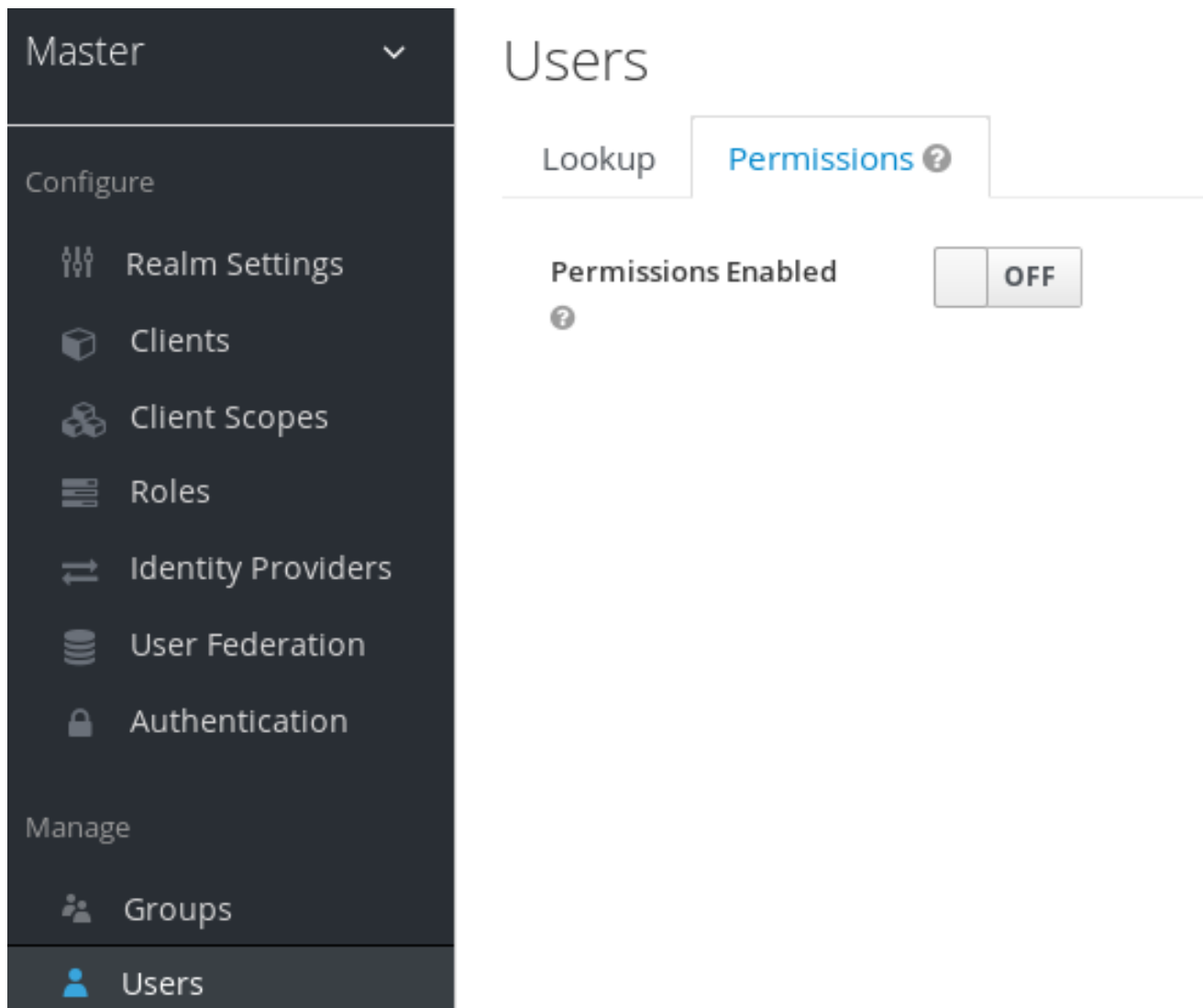
クライアントにダイレクトの Naked Impersonation を有効にすることは非常に危険です。クライアントの認証情報が盗まれた場合、そのクライアントは、システム内の任意のユーザーになりすますことができます。

7.5.1. Exchange のパーミッションの付与

audience パラメーターを指定すると、呼び出し元のクライアントにはクライアントへの交換パーミッションが必要です。これを設定する方法については、この章の前半で説明しています。

さらに、呼び出し元のクライアントには、ユーザーになりすますためのアクセス許可を付与する必要があります。管理コンソールで、**Users** 画面に移動し、**Permissions** タブをクリックします。

ユーザー権限



Permissions Enabled スイッチを true に切り替えます。

ID プロバイダーパーミッション

The screenshot shows the same Keycloak interface, but the 'Permissions Enabled' toggle is now set to 'ON'. Below the toggle, there is a table with the following data:

scope-name	Description	Actions
view	Policies that decide if an administrator can view all users in realm	Edit
manage	Policies that decide if an administrator can manage all users in the realm	Edit
map-roles	Policies that decide if administrator can map roles for all users	Edit
manage-group-membership	Policies that decide if an administrator can manage group membership for all users in the realm. This is used in conjunction with specific group policy	Edit
impersonate	Policies that decide if administrator can impersonate other users	Edit
user-impersonated	Policies that decide which users can be impersonated. These policies are applied to the user being impersonated.	Edit

ページに 偽装 リンクが表示されるはずですが。それをクリックして、パーミッションの定義を開始します。このページが作成されます。

ユーザーのなりすましパーミッションの設定

Master ▾


Configure


- Realm Settings
- Clients**
- Client Scopes
- Roles
- Identity
- Providers
- User Federation
- Authentication



Manage


Clients > master-realm > Authorization > Permissions > user-impersonated.permission.users


User-impersonated.permission.users

Name *  user-impersonated.permission.users


Description 

Resource  Users  ▾

Scopes * 

Apply Policy  ▾ ▾

No policies assigned.

Decision Strategy  Unanimous ▾

このパーミッションのポリシーを定義する必要があります。**Authorization** リンクをクリックし、**Policies** タブに移動し、**Client** ポリシーを作成します。

クライアントポリシーの作成

Master ▾


Configure


- Realm Settings
- Clients**
- Client Scopes
- Roles
- Identity
- Providers
- User Federation
- Authentication


Manage

Clients > master-realm > Authorization > Policies > Add Client Policy


Add Client Policy

Name *  client-impersonators

Description 

Clients *  ▾

clientId	Actions
starting-client	<input type="button" value="Remove"/>

Logic  P ▾

ここでは、トークン交換を要求する認証されたクライアントである、開始クライアントを入力します。このポリシーを作成したら、ユーザーの偽装パーミッションに戻り、定義したクライアントポリシーを追加します。

クライアントポリシーの適用

Master
▼

Configure

Realm Settings

Clients

Client Scopes

Roles

Identity

Providers

User Federation

Authentication

Manage

Groups

Clients > master-realm > Authorization > Permissions > user-impersonated.permission.users

User-impersonated.permission.users

Name * ⓘ user-impersonated.permission.users

Description ⓘ

Resource ⓘ Users

Scopes * ⓘ user-impersonated

Apply Policy ⓘ

Select existing policy...

Create

Decision Strategy ⓘ Unanimous

クライアントには、ユーザーの権限を借用できるパーミッションがあります。これを正しく行わないと、このタイプの交換を行おうとすると、403 Forbidden 応答が返されます。



注記

パブリッククライアントは、ダイレクトの Naked Impersonation を行うことはできません。

7.5.2. リクエストの作成

要求を行うには、**requested_subject** パラメーターを指定します。これは、有効なユーザーのユーザー名またはユーザー ID である必要があります。必要に応じて、**audience** パラメーターを指定することもできます。

```
curl -X POST \
  -d "client_id=starting-client" \
  -d "client_secret=geheim" \
  --data-urlencode "grant_type=urn:ietf:params:oauth:grant-type:token-exchange" \
  -d "requested_subject=wburke" \
  http://localhost:8080/auth/realms/myrealm/protocol/openid-connect/token
```

7.6. サービスアカウントでアクセス許可モデルを展開

クライアントに交換のパーミッションを与える場合、必ずしもすべてのクライアントに対してそれらのパーミッションを手動で有効にする必要はありません。クライアントにサービスアカウントが関連付けられている場合、ロールを使用してパーミッションをグループ化し、クライアントのサービスアカウントにロールを割り当てることで交換パーミッションを割り当てることができます。たとえば、**naked-exchange** ロールと、そのロールを持つサービスアカウントが、そのままの交換を実行できるように定義することができます。

7.7. 交換の脆弱性

トークン交換を許可し始める際に、認識し、注意しなければならないさまざまなことがあります。

1つ目はパブリッククライアントです。パブリッククライアントは、交換を実行するためのクライアント認証情報を持っていないか、またはこれを必要としません。有効なトークンを持つ Note は、パブリッククライアントになりすまして、パブリッククライアントが実行可能な交換を実行できます。レルムが管理するクライアントで信頼できないクライアントが存在する場合、パブリッククライアントはパーミッションモデルに脆弱性を開放する可能性があります。この理由により、ダイレクトの Naked 交換は、パブリッククライアントを許可せず、呼び出し元のクライアントがパブリックの場合にエラーを表示して中止します。

Facebook や Google などが提供するソーシャルトークンをレルムトークンと交換することができま
す。これらのソーシャル Web サイトで、偽のアカウントを作成することは難しくないため、交換トークンで許可されていることには注意し、慎重に対応してください。デフォルトのロール、グループ、およびアイデンティティプロバイダーマッパーを使用して、外部のソーシャルユーザーに割り当てられた属性とロールを制御します。

ダイレクトの Naked の交換は非常に危険です。クライアントの認証情報をリークしない呼び出し元のクライアントに多大な信頼を置いています。これらの認証情報がリークされる場合、盗む側はシステム内の誰かになりすますことができます。これは、既存のトークンを持つ機密クライアントと直接対照的です。認証には、アクセストークンとクライアント認証情報の2つの要素があり、1人のユーザーのみを処理します。したがって、ダイレクトの Naked の交換は控えめに使用してください。