# Red Hat build of Keycloak 24.0

## アプリケーションおよびサービスの保護ガイド

Last Updated: 2024-06-04

# Red Hat build of Keycloak 24.0 アプリケーションおよびサービスの保護ガイド

## 法律上の通知



## 概要

このガイドには、Red Hat build of Keycloak 24.0 のアプリケーションおよびサービスの保護に関する情報が記載されています。

# 目次

# 多様性を受け入れるオープンソースの強化

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

# 第1章 アプリケーションとサービスのセキュア化のプランニング

Red Hat build of Keycloak は、OAuth2、OpenID Connect、および SAML 準拠のサーバーとして、使用しているテクノロジースタックがこれらのプロトコルのいずれかをサポートしている限り、あらゆるアプリケーションとサービスを保護できます。Red Hat build of Keycloak でサポートされているセキュリティープロトコルの詳細は、サーバー管理ガイド を参照してください。

これらのプロトコルの一部は、使用しているプログラミング言語、フレームワーク、またはリバースプロキシーですでにサポートされています。アプリケーションエコシステムから利用できるサポートを活用することは、アプリケーションをセキュリティー標準とベストプラクティスに完全に準拠させ、ベンダーロックインを回避するために重要です。

Red Hat build of Keycloak は、一部のプログラミング言語について、特定のセキュリティープロトコルにおけるサポート不足を解消するため、またはより緊密で充実したサーバーとの統合を提供するために、ライブラリーを提供しています。これらのライブラリーは **Keycloak クライアントアダプター** として知られており、アプリケーションエコシステムで利用できるものがない場合の最終手段として使用するものとします。

## 1.1. アプリケーションとサービスを保護する基本的な手順

以下は、Red Hat build of Keycloak のアプリケーションまたはサービスを保護するための基本的な手順です。

1. 次のオプションのいずれかを使用して、クライアントをレルムに登録します。
   - Red Hat build of Keycloak 管理コンソール
   - クライアント登録サービス
   - CLI
2. 次のオプションのいずれかを使用して、アプリケーションで OpenID Connect または SAML プロトコルを有効にします。
   - アプリケーションエコシステムから OpenID Connect と SAML の既存サポートを活用する
   - Red Hat build of Keycloak アダプターを使用する

このガイドでは、これらのステップを詳細に説明します。管理コンソールを通じて Red Hat build of Keycloak にクライアントを登録する方法の詳細は、サーバー管理ガイド を参照してください。

## 1.2. スタートガイド

Red Hat build of Keycloak クイックスタートリポジトリー には、さまざまなプログラミング言語とフレームワークを使用してアプリケーションとサービスを保護する方法の例があります。ドキュメントとコードベースを確認することで、 Red Hat build of Keycloak を使用してアプリケーションとサービスを保護するために必要な最小限の変更を理解できます。

OpenID Connect プロトコルと SAML プロトコルの両方で信頼される、クライアント側の一般的な実装に関する推奨事項については、次のセクションを参照してください。

### 1.2.1. OpenID Connect

#### 1.2.1.1. JavaScript (クライアント側)

- JavaScript

#### 1.2.1.2. Node.js (サーバー側)

- Node.js

### 1.2.2. SAML

#### 1.2.2.1. Java

- JBoss EAP

## 1.3. 用語

このガイドでは、以下の用語を使用しています。

- **Clients** は、Red Hat build of Keycloak と対話してユーザーを認証し、トークンを取得するエンティティーです。多くの場合、クライアントは、そのユーザーにシングルサインオンエクスペリエンスを提供し、サーバーが発行するトークンを使用して他のサービスにアクセスするユーザーの代わりに機能するアプリケーションおよびサービスです。クライアントは、トークンの取得のみに関心があり、他のサービスにアクセスするために機能するエンティティーである場合もあります。

- **Applications** には、それぞれのプロトコルについて特定のプラットフォームで動作する幅広いアプリケーションが含まれます。

- **Client adapters** は、Red Hat build of Keycloak を使用したアプリケーションとサービスの保護を簡単に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します。

- **Creating a client** と **registering a client** は、同じアクションです。**Creating a Client** は、管理コンソールを使用してクライアントを作成するのに使用する用語です。**Registering a client** は、Red Hat build of Keycloak クライアント登録サービスを使用してクライアントを登録することを意味する用語です。

- **A service account** は、自分自身のためにトークンを取得できるクライアントの一種です。

# 第2章 OPENID CONNECT を使用したアプリケーションとサービスのセキュア化

このセクションでは、Red Hat build of Keycloak を使用して OpenID Connect でアプリケーションとサービスを保護する方法について説明します。

## 2.1. 利用可能なエンドポイント

Red Hat build of Keycloak は、完全に準拠した OpenID Connect プロバイダーの実装として、アプリケーションとサービスがユーザーの認証と認可に使用できるエンドポイントのセットを公開します。

このセクションでは、アプリケーションとサービスが Red Hat build of Keycloak と対話する際に使用する必要がある、いくつかの主要なエンドポイントについて説明します。

### 2.1.1. 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、**well-known** 設定エンドポイントです。Red Hat build of Keycloak 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リスト表示します。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 build of Keycloak 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/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リスト表示しないといけない場合があります。

#### 2.1.1.1. 認可エンドポイント

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

認可エンドポイントはエンドユーザーの認証を実行します。この認証は、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によって行われます。

詳細は、OpenID Connect 仕様の 認可エンドポイント セクションを参照してください。

#### 2.1.1.2. 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、認可コードを調べるか、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。

#### 2.1.1.3. 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ます。このエンドポイントは、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。

#### 2.1.1.4. 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される可能性があり、これにより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。

#### 2.1.1.5. 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。レルム設定に応じて、トークンを検証するために 1 つ以上のキーを有効にできます。詳細は、サーバー管理ガイド および JSON Web キーの仕様 を参照してください。

#### 2.1.1.6. 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。このエンドポイントは、機密クライアントによってのみ呼び出せます。

このエンドポイントで呼び出す方法の詳細については、OAuth 2.0 Token Introspection specification を参照してください。

#### 2.1.1.7. 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 を参照してください。

#### 2.1.1.8. 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。更新トークンを取り消すと、対応するクライアントに対するユーザーの同意も取り消されます。

このエンドポイントで呼び出す方法の詳細については、OAuth 2.0 Token Revocation specification を参照してください。

#### 2.1.1.9. デバイス認可エンドポイント

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

デバイス認可エンドポイントは、デバイスコードとユーザーコードを取得するために使用されます。これは、機密またはパブリッククライアントで呼び出すことができます。

このエンドポイントで呼び出す方法の詳細については、OAuth 2.0 Device Authorization Grant specification を参照してください。

#### 2.1.1.10. backchannel 認証エンドポイント

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

backchannel 認証エンドポイントは、クライアントによる認証要求を識別する auth_req_id を取得するために使用されます。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呼び出す方法の詳細は、OpenID Connect Client Initiated Backchannel Authentication Flow specification を参照してください。

このガイドの backchannel 認証エンドポイント およびサーバー管理ガイドの クライアントが開始したバックチャンネル認証の許可 セクションなど、Red Hat build of Keycloak ドキュメントの別のパートも参照してください。

## 2.2. サポートされている許可タイプ

このセクションでは、リレーパーティーが利用できる様々な許可タイプについて説明します。

### 2.2.1. 認可コード

認可コードフローは、ユーザーエージェントを Red Hat build of Keycloak にリダイレクトします。Red Hat build of Keycloak でユーザーが正常に認証されると、認可コードが作成され、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認可コードを使用して、Red Hat build of Keycloak 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。

### 2.2.2. 暗黙的

暗黙的フローは認可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認可コードを返す代わりに、このフローが返されます。このアプローチにより、追加の呼び出しでアクセストークンの認可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。そのため、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。しかし、これらのトークンを無効にするのは非常に難しいため、このアプローチは現実的ではありません。あるいは、最初のアクセストークンの有効期限が切れた後、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を要求できます。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。

代わりに、アクセストークンと認可コードの両方が返されるハイブリッドフローを使用できます。

注意すべき点は、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ーリスクがあることです。アクセストークンの有効期限を短くすることで、この問題をある程度軽減できます。

詳細は、OpenID Connect 仕様の 暗黙的フロー を参照してください。

現在の OAuth 2.0 セキュリティーのベストプラクティス によれば、このフローは使用するべきではありません。このフローは、将来の OAuth 2.1 仕様 から削除されます。

### 2.2.3. リソースオーナーパスワード認証情報

Red Hat build of Keycloak では Direct Grant と呼ばれるリソース所有者の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。現在の OAuth 2.0 セキュリティーベストプラクティス によれば、このフローは使用すべきではありません。「デバイス認可グラント」 や 「認可コード」 などの代替方法を推奨します。

このフローの使用には次のような制限があります。

- ユーザーの認証情報が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。
- アイデンティティー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。

このフローのセキュリティー上の懸念点は次のとおりです。

- Red Hat build of Keycloak が認証情報の処理に関与する
- 認証情報の漏洩が発生する可能性のある脆弱な領域が増加する
- ユーザーが認証情報の入力に Red Hat build of Keycloak ではなく別のアプリケーションを信頼するような環境が生まれる

クライアントでリソースオーナーパスワード認証情報の使用を許可するには、クライアントの **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。将来の OAuth 2.1 仕様 からは削除されます。

詳細は、OAuth 2.0 仕様の リソースオーナーパスワード認証情報の付与 の章を参照してください。

#### 2.2.3.1. 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/realms/master/protocol/openid-connect/token"
```

### 2.2.4. クライアントクレデンシャル

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) がユーザーの代わりにではなく、自身の代わりにアクセスを取得する場合に使用されます。たとえば、これらの認証情報は、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 build of Keycloak は、クライアントがシークレットまたは公開鍵/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 クライアント認証情報の付与 の章を参照してください。

### 2.2.5. デバイス認可グラント

デバイス認可グラントは、入力機能が制限されているか、適切なブラウザーがないインターネット接続デバイスで実行されているクライアントによって使用されます。

1. アプリケーションが、Red Hat build of Keycloak にデバイスコードとユーザーコードを提供するよう要求します。
2. Red Hat build of Keycloak が、デバイスコードとユーザーコードを作成します。
3. Red Hat build of Keycloak が、デバイスコードおよびユーザーコードなどの応答をアプリケーションに返します。
4. アプリケーションが、ユーザーにユーザーコードと検証 URI を提供します。ユーザーは検証 URI にアクセスし、別のブラウザーを使用して認証されます。
5. Red Hat build of Keycloak がユーザー認証を完了するまで、アプリケーションが Red Hat build of Keycloak を繰り返しポーリングします。
6. ユーザー認証が完了すると、アプリケーションはデバイスコードを取得します。
7. アプリケーションは認証情報と共にデバイスコードを使用して、Red Hat build of Keycloak 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。

詳細は、OAuth 2.0 デバイス認可グラントの仕様 参照してください。

### 2.2.6. Client Initiated Backchannel Authentication Grant

Client Initiated Backchannel Authentication Grant は、OAuth 2.0 の認可コードグラントのように、ユーザーのブラウザーを介してリダイレクトせずに、OpenID プロバイダーと直接通信することで認可フローを開始するクライアントによって使用されます。

クライアントは、クライアントによる認証要求を識別する auth_req_id を Red Hat build of Keycloak に要求します。Red Hat build of Keycloakは auth_req_id を作成します。

クライアントは、auth_req_id を受信したてからユーザーが認証されるまで、Red Hat build of Keycloak からアクセストークン、更新トークン、および ID トークンを取得するために、auth_req_id と引き換えに Red Hat build of Keycloak を繰り返しポーリングする必要があります。

クライアントが **ping** モードを使用する場合は、トークンエンドポイントを繰り返しポーリングする必要はありませんが、Red Hat build of Keycloak から指定されたクライアント通知エンドポイントに送信される通知を待つことができます。クライアント通知エンドポイントは、Red Hat build of Keycloak 管理コンソールで設定できます。クライアント通知エンドポイントのコントラクトの詳細は、CIBA 仕様を参照してください。

詳細は、OpenID Connect クライアントが開始したバックチャンネル認証フローの仕様 を参照してください。

また、このガイドの バックチャンネル認証エンドポイント やサーバー管理ガイドの クライアントが開始したバックチャンネル認証グラント セクションなど、Red Hat build of Keycloak ドキュメントの別のパートも参照してください。FAPI CIBA 準拠の詳細は、このガイドの FAPI セクション を参照してください。

## 2.3. RED HAT BUILD OF KEYCLOAK JAVA アダプター

### 2.3.1. Red Hat JBoss Enterprise Application Platform

Red Hat build of Keycloak には、Red Hat JBoss Enterprise Application Platform のアダプターは含まれていません。ただし、Red Hat JBoss Enterprise Application Platform にデプロイされた既存アプリケーションの代替があります。

#### 2.3.1.1. 8.0 Beta

Red Hat Enterprise Application Platform 8.0 Beta は、Elytron OIDC クライアントサブシステムを通じてネイティブ OpenID Connect クライアントを提供します。

詳細は、Red Hat JBoss Enterprise Application Platform のドキュメント を参照してください。

#### 2.3.1.2. 6.4 と 7.x

Red Hat JBoss Enterprise Application Platform 6.4 および 7.x にデプロイされた既存アプリケーションは、Red Hat build of Keycloak サーバーと組み合わせて Red Hat Single Sign-On 7.6 のアダプターを利用できます。

詳細は、Red Hat Single Sign-On のドキュメント を参照してください。

### 2.3.2. Spring Boot アダプター

Red Hat build of Keycloak には、Spring Boot 用のアダプターは含まれていません。ただし、Spring Boot でビルドされた既存アプリケーションの代替があります。

Spring Security は、OAuth 2 と OpenID Connect の包括的なサポートを提供します。詳細は、Spring Security のドキュメント を参照してください。

Spring Boot 2.x の場合は、Red Hat Single Sign-On 7.6 の Spring Boot アダプターを Red Hat build of Keycloak サーバーと組み合わせて使用することもできます。詳細は、Red Hat Single Sign-On のドキュメント を参照してください。

## 2.4. RED HAT BUILD OF KEYCLOAK JAVASCRIPT アダプター

Red Hat build of Keycloak には、Web アプリケーションの保護に使用できる、**keycloak-js** と呼ばれるクライアント側の JavaScript ライブラリーが付属しています。このアダプターには、Cordova アプリケーションのビルトインサポートもあります。

### 2.4.1. インストール

アダプターはいくつかの方法で配布されますが、NPM から **keycloak-js** パッケージをインストールすることが推奨されます。

```
npm install keycloak-js
```

ライブラリーは、**/js/keycloak.js** にある Red Hat build of Keycloak サーバーから直接取得することも可能で、さらに ZIP アーカイブとしても配布されます。ただし、Keycloak サーバーから直接アダプターを組み込む方法は非推奨化が検討されており、今後この機能は来削除される可能性があります。

### 2.4.2. Red Hat build of Keycloak サーバーの設定

クライアント側のアプリケーションの使用に関しては、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、クライアントはパブリッククライアントでなければならない点を考慮する必要があります。これにより、クライアント用に設定したリダイレクト 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。

アダプターを使用するには、Red Hat build of Keycloak 管理コンソールででアプリケーションのクライアントを作成します。**Capability config** ページで **Client authentication** を **Off** に切り替えて、クライアントを公開します。

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、セキュリティー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。

### 2.4.3. アダプターの使用

次の例は、アダプターを初期化する方法を示しています。**Keycloak** コンストラクターに渡されるオプションは、必ず設定したクライアントのオプションに置き換えてください。

```
import Keycloak from 'keycloak-js';

const keycloak = new Keycloak({
    url: 'http://keycloak-server${kc_base_path}',
    realm: 'myrealm',
    clientId: 'myapp'
});

try {
    const authenticated = await keycloak.init();
    console.log(`User is ${authenticated ? 'authenticated' : 'not authenticated'}`);
} catch (error) {
    console.error('Failed to initialize adapter:', error);
}
```

認証するには、**login** 関数を呼び出します。アダプターを自動的に認証する場合、2 つのオプションがあります。、**login-required** または **check-sso** を **init()** 関数に渡せます。

- **login-required** は、ユーザーが Red Hat build of Keycloak にログインしている場合はクライアントを認証し、ユーザーがログインしていない場合はログインページを表示します。

- **check-sso** は、ユーザーがすでにログインしている場合に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 build of Keycloak サーバーの完全なリダイレクトを実行せず、アプリケーションには戻りません。しかし、このこのアクションは非表示の iframe で実行されます。したがって、ブラウザーはアプリケーションの初期化時に一度だけアプリケーションリソースをロードして解析し、Red Hat build of Keycloak からアプリケーションにリダイレクトされた後にこれを再び実行することはありません。このアプローチは、SPA (Single Page Applications) の場合に特に役立ちます。

**silent check-sso** を有効にするには、init メソッドで **silentCheckSsoRedirectUri** 属性を指定します。この URI が、アプリケーション内の有効なエンドポイントであることを確認してください。Red Hat build of Keycloak 管理コンソール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: `${location.origin}/silent-check-sso.html`
});
```

認証の状態が正常にチェックされ、Red Hat build of Keycloak 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。

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

このページは、アダプターの **一部ではなく**、**silentCheckSsoRedirectUri** の指定された場所にあるアプリケーションによって提供される必要がある点に注意してください。

**警告**

**サイレント check-sso** 機能は、一部の最新のブラウザーで制限さていれます。トラッキング防止セクションを実装する最新のブラウザー を参照してください。

**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、**Authorization** ヘッダーにベアラートークンを含めることで、アプリケーションは Red Hat build of Keycloak で保護された RESTful サービスへのリクエストを実行できます。以下に例を示します。

```
async function fetchUsers() {
    const response = await fetch('/api/users', {
        headers: {
            accept: 'application/json',
            authorization: `Bearer ${keycloak.token}`
        }
    });

    return response.json();
}
```

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。このトークンを更新するには、**updateToken()** メソッドを呼び出します。このメソッドは、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。

```
try {
    await keycloak.updateToken(30);
} catch (error) {
    console.error('Failed to refresh token:', error);
}

const users = await fetchUsers();
```

**注記**

アクセストークンとリフレッシュトークンは両方ともメモリーに保存され、どの種類のストレージにも永続化されません。したがって、ハイジャック攻撃を防ぐために、これらのトークンは永続化すべきではありません。

### 2.4.4. セッションステータスの iframe

デフォルトでは、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。この iframe はネットワークトラフィックを必要としません。代わりに、ステータスは特別なステータス Cookie を参照して取得されます。**init()** メソッドに渡されるオプションで **checkLoginIframe: false** を設定すると、この機能を無効化できます。

このクッキーを直接参照する必要はありません。その形式は変更される可能性があり、アプリケーションではなく Red Hat build of Keycloak サーバーの URL にも関連付けられます。

**警告**

セッションステータスの iframe 機能は、一部の最新のブラウザーで制限されています。トラッキング防止セクションを実装する最新のブラウザー を参照してください。

### 2.4.5. 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、アダプターは 認可コード フローを使用します。

このフローでは、Red Hat build of Keycloak は、認証トークンではなく認可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更新トークンの **コード** を交換します。

Red Hat build of Keycloak は、Red Hat build of Keycloak で認証が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。このフローは、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が、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、セキュリティー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。

暗黙的フローを有効にするには、Red Hat build of Keycloak 管理コンソールで、クライアントの **Implicit Flow Enabled** フラグを有効にします。また、パラメーター **flow** を **implicit** の値とともに **init** メソッドに渡します。

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

アクセストークンのみが提供され、更新トークンは存在しないことに注意してください。この状況は、アクセストークンの有効期限が切れると、アプリケーションは Red Hat build of Keycloak 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ることを意味します。

Red Hat build of Keycloak は、Hybrid フローもサポートしています。

このフローでは、クライアントは管理コンソールで **標準フロー** と **暗黙的フロー** の両方を有効にする必要があります。Red Hat build of Keycloak 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ー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'
});
```

### 2.4.6. Cordova でのハイブリッドアプリケーション

Red Hat build of 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ます。アダプターには、**cordova** と **cordova-native** という 2 つのモードがあります。

デフォルトは **cordova** で、アダプタータイプが明示的に設定されておらず、**window.cordova** が存在する場合、アダプターは自動的にこれを選択します。ログインすると、InApp Browser が開き、ユーザーは Red Hat build of Keycloak と対話できるようになります。その後 **http://localhost** にリダイレクトしてアプリに戻ります。この動作のために、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します。

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認証情報にアクセスできる可能性があるなど、セキュリティー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。

代替モードは、異なるアプローチをする `cordova-native` です。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 を使用してアプリケーションにリダイレクトします。そこから、Red Hat build of Keycloak 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 2 つの追加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: * **myapp://login** または **android-app://com.example.myapp/https/example.com/login** などのカスタムスキーム * Universal Links (iOS)/Deep Links (Android) 。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、セキュリティーが強化されます。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ためには、ユニバーサルリンクをカスタム URL リンクを仕様するフォールバックサイトと組み合わせて使用することが推奨されます。

さらに、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" />
```

### 2.4.7. カスタムアダプター

状況によっては、Capacitor など、デフォルトでサポートされていない環境でアダプターを実行する必要がある場合があります。これらの環境で JavaScript クライアントを使用するには、カスタムアダプターを渡すことができます。たとえば、サードパーティーのライブラリーは、確実に実行できるようにするためのアダプターを提供できます。

```
import Keycloak from 'keycloak-js';
import KeycloakCapacitorAdapter from 'keycloak-capacitor-adapter';

const keycloak = new Keycloak();

keycloak.init({
    adapter: KeycloakCapacitorAdapter,
});
```

この特定のパッケージは存在しませんが、そのようなアダプターをクライアントに渡す方法に関する適切な例を示すことができます。

独自のアダプターを作成することもできます。そのためには、**KeycloakAdapter** インターフェイスで説明されているメソッドを実装する必要があります。たとえば、以下の TypeScript コードは、すべてのメソッドが適切に実装されていることを確認します。

```
import Keycloak, { KeycloakAdapter } from 'keycloak-js';

// Implement the 'KeycloakAdapter' interface so that all required methods are guaranteed to be
present.
const MyCustomAdapter: KeycloakAdapter = {
    login(options) {
        // Write your own implementation here.
    }

    // The other methods go here...
};

const keycloak = new Keycloak();

keycloak.init({
    adapter: MyCustomAdapter,
});
```

タイプ情報を省略することで TypeScript を使用せずにこれを実行することもできますが、インターフェイスを確実に適切に実装することは、完全にユーザー次第となります。

### 2.4.8. 追跡保護機能を備えた最新のブラウザー

一部のブラウザーの最新版では、Chrome の SameSite や完全にブロックされたサードパーティーの cookie など、サードパーティーによるユーザーの追跡を防ぐためにさまざまな cookie ポリシーが適用されています。これらのポリシーは、時間の経過とともにさらに制限が厳しくなり、他のブラウザーでも採用される可能性があります。最終的には、サードパーティーコンテキストの Cookie が完全にサポートされなくなり、ブラウザーによってブロックされる可能性があります。その結果、影響を受けるアダプター機能が、最終的に非推奨になる可能性があります。

アダプターは、セッションステータスの iframe、**サイレント check-sso**、および部分的に通常の (非サイレント) **check-sso** についても、サードパーティーの cookie に依存しています。これらの機能は、機能が制限されているか、ブラウザーが cookie に関してどのように制限されているかに基づいて完全に無効になっています。アダプターはこの設定を検出しようとし、それに応じて反応します。

#### 2.4.8.1. SameSite=Lax by Default ポリシーを持つブラウザー

Red Hat build of Keycloak 側およびアプリケーション側で SSL / TLS 接続が設定されている場合、すべての機能がサポートされます。たとえば、Chrome はバージョン 84 以降が影響を受けます。

#### 2.4.8.2. サードパーティーの Cookie がブロックされているブラウザー

セッションステータス iframe はサポートされず、このようなブラウザーの動作がアダプターによって検出されると自動的に無効になります。これは、アダプターは Single Sign-Out 検出にセッション cookie を使用できず、純粋にトークンに依存する必要があることを意味します。その結果、ユーザーが別のウィンドウでログアウトしても、アダプターを使用しているアプリケーションは、アプリケーションがアクセストークンの更新を試行するまでログアウトされません。したがって、ログアウトができるだけ早く検出されるように、アクセストークンの有効期間を比較的短く設定することを検討してください。詳細は、セッションとトークンのタイムアウト を参照してください。

**サイレント check-sso** はサポートされず、デフォルトで通常の (非サイレント) **check-sso** にフォールバックします。この動作は、**init** メソッドに渡されるオプションで、**SilentCheckSsoFallback: false** を設定することで変更できます。この場合、制限的なブラウザー動作が検出されると、**check-sso** は完全に無効になります。

通常の **check-sso** も影響を受けます。セッションステータス iframe がサポートされていないため、アダプターがユーザーのログインステータスをチェックするために初期化される際に、Red Hat build of Keycloak への追加のリダイレクトを行う必要があります。このチェックは、ユーザーがログインしているかどうかを示すために iframe を使用する標準の動作とは異なり、リダイレクトはユーザーがログアウトしている場合にのみ実行されます。

影響を受けるブラウザーは、たとえば、バージョン 13.1 以降の Safari などです。

### 2.4.9. API リファレンス

#### 2.4.9.1. コンストラクター

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

#### 2.4.9.2. プロパティー

**authenticated**

ユーザー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 build of Keycloak サーバー間の推定時間差 (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ことを試みます。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**

ログインリクエストとともに Red Hat build of Keycloak に送信されるレスポンスタイプ。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。

#### 2.4.9.3. メソッド

**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 - 暗号化ナンスを追加して、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。

- silentCheckSsoFallback: **サイレント check-sso** がブラウザーでサポートされない場合に、通常の **check-sso** へのフォールバックを有効にします (デフォルトは **true**)。
- 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)。
- scope - デフォルトのスコープパラメーターを Red Hat build of Keycloak ログインエンドポイントに設定します。スコープのスペースで区切られたリストを使用します。これらは通常、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。たとえば、スコープオプションの **address phone** を入力すると、Red Hat build of Keycloak 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。**login()** オプションでスコープを明示的に指定すると、ここで指定したデフォルトのスコープが上書きされることに注意してください。
- timeSkew - ローカルタイムと Red Hat build of Keycloak 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 (デフォルト値は **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 build of Keycloak 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 build of Keycloak 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) 方式。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 方式 (デフォルト)。
  - false - PKCE は無効です。
- acrValues - 認証コンテキストクラス参照を参照し、クライアントが必要なアシュアランスレベル要件 (認証メカニズムなど) を宣言できるようにする **acr_values** パラメーターを生成します。Section 4. acr_values request values and level of assurance in OpenID Connect MODRNA Authentication Profile 1.0 を参照してください。
- messageReceiveTimeout: Keycloak サーバーからのメッセージ応答を待つタイムアウトをミリ秒単位で設定します。これは、たとえば、サードパーティーの Cookie チェック中に、メッセージを待機するときに使用されます。デフォルト値は 10000 です。
- locale - onLoad が 'login-required' の場合、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。

初期化の完了時に解決する promise を返します。

**login(options)**

ログインフォームにリダイレクトします。

options は任意のオブジェクトです。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 - Red Hat build of Keycloak 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、値が **login** の場合にログイン画面を表示するように強制します。
- 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 - **init** で設定されたスコープを、この特定ログインの別の値でオーバーライドします。
- idpHint - Red Hat build of Keycloak 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ー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、アイデンティティープロバイダーのドキュメント を参照してください。
- acr: **acr** 要求についての情報が含まれます。これは、**claims** パラメーター内で Red Hat build of Keycloak サーバーに送信されます。一般的な用途は、段階的な認証です。使用例: **{ values: ["silver", "gold"], essential: true }**詳細は、OpenID Connect 仕様と ステップアップ認証のドキュメント を参照してください。
- acrValues - 認証コンテキストクラス参照を参照し、クライアントが必要なアシュアランスレベル要件 (認証メカニズムなど) を宣言できるようにする **acr_values** パラメーターを生成します。Section 4. acr_values request values and level of assurance in OpenID Connect MODRNA Authentication Profile 1.0 を参照してください。
- action - 値が **register** の場合、ユーザーは登録ページにリダイレクトされます。詳細は、クライアントから要求される登録に関するセクション を参照してください。値が **UPDATE_PASSWORD** またはサポートされている別の必須アクションの場合、ユーザーはパスワードのリセットページまたはその他の必須アクションページにリダイレクトされます。ただし、ユーザーが認証されていない場合は、ユーザーはログインページへ移動され、認証後にリダイレクトされます。詳細は、アプリケーションが開始したアクション セクションを参照してください。
- 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。
- 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" }**;

**createLoginUrl(options)**

ログインフォームへの URL を返します。

options は任意のオブジェクトで、関数 **login** と同じオプションをサポートします。

**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。

**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。

**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

**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

**accountManagement()**

アカウントコンソールにリダイレクトします。

**createAccountUrl(options)**

アカウントコンソールへの 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: アプリケーションにリダイレクトする際にリダイレクトする URI を指定します。

**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。

**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)。

**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。

```
try {
    const profile = await keycloak.loadUserProfile();
    console.log('Retrieved user profile:', profile);
} catch (error) {
    console.error('Failed to load user profile:', error);
}
```

**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)。

**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。-1 が minValidity として渡された場合、トークンは強制的に更新されます。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。

```
try {
    const refreshed = await keycloak.updateToken(5);
    console.log(refreshed ? 'Token was refreshed' : 'Token is still valid');
} catch (error) {
    console.error('Failed to refresh the token:', error);
}
```

**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、onAuthLogout コールバックリスナーが呼び出されます。

#### 2.4.9.4. 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。これらは **init()** メソッドを呼び出す前に設定する必要があることに注意してください。

以下に例を示します。

```
keycloak.onAuthSuccess = () => console.log('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。

## 2.5. RED HAT BUILD OF KEYCLOAK NODE.JS アダプター

Red Hat build of Keycloak 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。これは、Express.js などのフレームワークと統合できる柔軟性を備えることが目的でした。

Node.js アダプターを使用するには、まず Red Hat build of Keycloak 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、右上の **Action** をクリックし、**Download adapter config** を選択します。Format で **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、プロジェクトのルートフォルダーにあります。

### 2.5.1. 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ーを作成します。

```
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。次に、依存関係リストに Red Hat build of Keycloak 接続アダプターを追加します。

```
"dependencies": {
    "keycloak-connect": "file:keycloak-connect-24.0.0+redhat-00001.tgz"
}
```

### 2.5.2. 使用方法

**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。

プロジェクトのルートディレクトリーに **server.js** という名前のファイルを作成し、以下のコードを追加します。

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

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

- **express-session** 依存関係をインストールします。

```
npm install express-session
```

**server.js** スクリプトを起動するには、**package.json** の 'scripts' セクションに以下のコマンドを追加します。

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "node server.js"
},
```

これで、以下のコマンドでサーバーを実行できるようになりました。

```
npm run start
```

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 (ここではルートフォルダー)、Red Hat build of 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 など)。

この場合、Red Hat build of Keycloak 管理コンソールにアクセスするために Red Hat build of Keycloak デプロイメントが必要です。

Podman または Docker を使用して Red Hat build of Keycloak 管理コンソールをデプロイする方法の詳細は、それぞれのリンクを参照してください。

これで、Red Hat build of Keycloak Admin Console → clients (左サイドバー) → choose your client → Installation → Format Option → Keycloak OIDC JSON → Download に移動して、**keycloak.json** ファイルを取得できます。

ダウンロードしたファイルをプロジェクトのルートディレクトリーに貼り付けます。

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。

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

const 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ープロバイダーにリダイレクトすることもできます。

```
const 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。

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

// Configure session
app.use(
  session({
    secret: 'mySecret',
    resave: false,
    saveUninitialized: true,
    store: memoryStore,
  })
);

const 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 build of Keycloak 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。

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

### 2.5.3. 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。

これを行うには、まず Express をインストールする必要があります。

```
npm install express
```

次に、以下で説明されているようにプロジェクトに Express が必要になります。

```
const express = require('express');
const app = express();
```

また、以下のコードを追加して Express で Keycloak ミドルウェアを設定します。

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

最後に、以下のコードを **main.js** に追加して、3000 ポートで HTTP 要求をリッスンするようにサーバーをセットアップしましょう。

```
app.listen(3000, function () {
  console.log('App listening on port 3000');
});
```

### 2.5.4. プロキシーの設定

SSL 接続を終了するプロキシーの背後でアプリケーション実行されている場合は、express behind proxies ガイドに従って、Express を設定する必要があります。誤ったプロキシー設定を使用すると、無効なリダイレクト URI が生成されることがあります。

設定例:

```
const app = express();

app.set( 'trust proxy', true );

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

### 2.5.5. 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 );
```

**ロールベースの認可**

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ーを保護するには、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 );
```

**リソースベースの認可**

リソースベースの認可を使用すると、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可を外部化できます。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応答しなかった場合、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) {
  const token = req.kauth.grant.access_token.content;
  const 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リストのみを返します。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: 'permissions'}), function (req, res) {
  const 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、増分認可などの 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サ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、次の設定を使用して、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 認可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の詳細は、認可サービスガイド を参照してください。

**高度な認可**

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 );
```

高度なログイン設定:

デフォルトでは、クライアントがベアラーのみでない限り、承認されていないすべてのリクエストは Red Hat build of Keycloak のログインページにリダイレクトされます。ただし、機密またはパブリッククライアントは、閲覧可能なエンドポイントと API エンドポイントの両方をホストする場合があります。認証されていない API 要求でのリダイレクトを防ぎ、代わりに HTTP 401 を返すようにするには、redirectToLogin 関数をオーバーライドします。

たとえば、このオーバーライドでは URL に /api/ が含まれているかどうかを確認し、ログインリダイレクトを無効にします。

```
Keycloak.prototype.redirectToLogin = function(req) {
const apiReqMatcher = /\/api\//i;
return !apiReqMatcher.test(req.originalUrl || req.url);
};
```

### 2.5.6. 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 build of Keycloak 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' } ));
```

ユーザーがトリガーするログアウトが呼び出されると、クエリーパラメーター **redirect_url** を渡すことができます。

```
https://example.com/logoff?
redirect_url=https%3A%2F%2Fexample.com%3A3000%2Flogged%2Fout
```

次に、このパラメーターは OIDC ログアウトエンドポイントのリダイレクト URL として使用され、ユーザーは **https://example.com/logged/out** にリダイレクトされます。

**Red Hat build of Keycloak 管理者コールバック**

ミドルウェアは、Red Hat build of Keycloak コンソールのコールバックもサポートし、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' } );
```

### 2.5.7. 完全な例

Node.js アダプターの完全な使用例は、Keycloak quickstarts for Node.js にあります。

## 2.6. FINANCIAL-GRADE API (FAPI) サポート

Red Hat build of Keycloak を使用すると、管理者はクライアントが以下の仕様に準拠していることを簡単に確認できます。

- Financial-grade API Security Profile 1.0 - Part 1: Baseline
- Financial-grade API Security Profile 1.0 - Part 2: Advanced
- Financial-grade API: Client Initiated Backchannel Authentication Profile (FAPI CIBA)
- FAPI 2.0 Security Profile (Draft)
- FAPI 2.0 Message Signing (Draft)

このコンプライアンスは、Red Hat build of Keycloak サーバーが、これらの仕様に記載されている認可サーバーの要件を検証することを意味します。Red Hat build of Keycloak アダプターには FAPI に対する特別なサポートがないため、クライアント (アプリケーション) 側で必要な検証は、引き続き手動で行うか、その他のサードパーティーソリューションを介して実行する必要がある場合があります。

### 2.6.1. FAPI クライアントプロファイル

クライアントが FAPI に準拠していることを確認するには、サーバー管理ガイド の説明に従って、レルムでクライアントポリシーを設定し、FAPI サポート用のグローバルクライアントプロファイルにリンクします。このプロファイルは、各レルムで自動的に使用可能になります。クライアントが準拠する必要のある FAPI プロファイルに基づいて、**fapi-1-baseline** または **fapi-1-advanced** プロファイルのいずれかを使用できます。また、FAPI 2 Draft 仕様に準拠させる場合は、プロファイル **fapi-2-security-profile** または **fapi-2-message-signing** を使用できます。

Pushed Authorization Request (PAR) を使用する場合は、クライアントが **fapi-1-baseline** プロファイルおよび **fapi-1-advanced** の両方を PAR 要求に使用することが推奨されます。具体的には、 **fapi-1-baseline** プロファイルには、**pkce-enforcer** エグゼキューターが含まれています。これにより、クライアントはセキュリティーで保護された S256 アルゴリズムで PKCE を確実に使用します。これは、PAR 要求を使用しない限り、FAPI Advanced クライアントには必須ではありません。

FAPI に準拠する方法で CIBA を使用する場合は、クライアントが **fapi-1-advanced** および **fapi-ciba** クライアントプロファイルの両方を使用していることを確認してください。**fapi-ciba** プロファイルには CIBA 固有のエクゼキューターのみが含まれるため、**fapi-1-advanced** プロファイル、または要求されたエクゼキューターを含む他のクライアントプロファイルを使用する必要があります。FAPI CIBA 仕様の要件を実施する場合は、機密クライアントまたは証明書をバインドしたアクセストークンの適用など、より多くの要件が必要になります。

### 2.6.2. Open Finance Brasil Financial-grade API Security Profile

Red Hat build of Keycloak は、Open Finance Brasil Financial-grade API Security Profile 1.0 Implementers Draft 3 に準拠しています。このプロファイルの一部の要件は、FAPI 1 Advanced 仕様よりも厳格です。そのため、一部の要件を適用するには、より厳格な方法で クライアントポリシー を設定する必要がある場合があります。以下の場合は、特にその必要があります。

- クライアントが PAR を使用しない場合は、暗号化された OIDC 要求オブジェクトが使用されていることを確認してください。これは、**Encryption Required** を有効にして設定された **secure-request-object** エグゼキューターを持つクライアントプロファイルを使用して実現できます。

- JWS の場合は、クライアントが **PS256** アルゴリズムを使用していることを確認してください。JWE の場合、クライアントは **A256GCM** による **RSA-OAEP** を使用する必要があります。これらのアルゴリズムが適用可能なすべての クライアント設定 でこれを設定する必要がある場合があります。

### 2.6.3. Australia Consumer Data Right (CDR) Security Profile

Red Hat build of Keycloak は、Australia Consumer Data Right Security Profile に準拠しています。

Australia CDR セキュリティープロファイルを適用する場合、**fapi-1-advanced** プロファイルを使用する必要があります。Australia CDR セキュリティープロファイルは FAPI 1.0 Advanced セキュリティープロファイルに基づいているためです。クライアントが PAR も適用する場合は、クライアントが RFC 7637 Proof Key for Code Exchange (PKCE) を適用することを確認してください。Australia CDR セキュリティープロファイルでは、PAR を適用するときに PKCE を適用することが要求されるためです。これは、**pkce-enforcer** エクゼキューターでクライアントプロファイルを使用することで実現できます。

### 2.6.4. TLS に関する考慮事項

機密情報が交換されるため、すべての対話は TLS (HTTPS) で暗号化される必要があります。さらに、使用される暗号スイートおよび TLS プロトコルバージョンの FAPI 仕様にはいくつかの要件があります。これらの要件に合致するには、許可された暗号を設定することを検討してください。これは、**https-protocols** および **https-cipher-suites** オプションで設定できます。Red Hat build of Keycloak はデフォルトで **TLSv1.3** を使用し、このデフォルト設定は変更する必要がないこともあります。ただし、何らかの理由で下位の TLS バージョンにフォールバックする必要がある場合は、暗号化を調整する必要がある場合があります。詳細は、TLS の設定 の章を参照してください。

## 2.7. OAUTH 2.1 のサポート

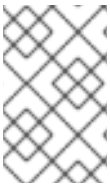
Red Hat build of Keycloak を使用すると、管理者はクライアントが以下の仕様に準拠していることを簡単に確認できます。

- The OAuth 2.1 Authorization Framework - ドラフト仕様

このコンプライアンスは、Red Hat build of Keycloak サーバーが、これらの仕様に記載されている認可サーバーの要件を検証することを意味します。Red Hat build of Keycloak アダプターには OAuth 2.1 に対する特別なサポートがないため、クライアント (アプリケーション) 側で必要な検証は、引き続き手動で行うか、その他のサードパーティーソリューションを介して実行する必要がある場合があります。

### 2.7.1. OAuth 2.1 クライアントプロファイル

クライアントが OAuth 2.1 に準拠していることを確認するには、サーバー管理ガイド の説明に従って、レルムでクライアントポリシーを設定し、OAuth 2.1 サポート用のグローバルクライアントプロファイルにリンクします。このプロファイルは、各レルムで自動的に使用可能になります。機密クライアントには **oauth-2-1-for-confidential-client** プロファイル、パブリッククライアントには **oauth-2-1-for-public-client** プロファイルを使用できます。



**注記**

OAuth 2.1 仕様はまだドラフト段階であり、将来変更される可能性があります。したがって、Red Hat build of Keycloak の組み込み OAuth 2.1 クライアントも変更される可能性があります。

**注記**

パブリッククライアントに OAuth 2.1 プロファイルを使用する場合、サーバー管理ガイドで説明されているように、DPoP プレビュー機能を使用することを推奨します。DPoP は、アクセストークンとリフレッシュトークンをクライアントのキーペアのパブリック部分とバインドするためです。このバインディングは、攻撃者が盗まれたトークンを使用するのを防ぎます。

## 2.8. 推奨事項

このセクションでは、Red Hat build of Keycloak を使用してアプリケーションを保護する場合のいくつかの推奨事項について説明します。

### 2.8.1. アクセストークンの検証

Red Hat build of Keycloak が発行したアクセストークンを手動で検証する必要がある場合は、イントロスペクションエンドポイント を呼び出せます。このアプローチの欠点は、Red Hat build of Keycloak サーバーに対して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、サーバーが過負荷になる可能性があります。Red Hat build of Keycloak が発行した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、発行したレルム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。レルム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、多くのサードパーティーライブラリーが存在し、JWS の検証に使用できます。

### 2.8.2. 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、必ずクライアントに有効なリダイレクト URI を使用してください。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
- 不正なエントリー - ユーザーが Red Hat build of Keycloak ですでに認証されている場合、攻撃者はリダイレクト URI が正しく設定されていないパブリッククライアントを使用し、ユーザーが知らないうちにユーザーをリダイレクトしてアクセスを取得できる可能性があります。

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://127.0.0.1**

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可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。OAuth 2.0 for Native Apps にあるとおり、**localhost** の使用は

**推奨されません**。代わりに、IP リテラル **127.0.0.1** を使用する必要があることに注意してください。

**urn:ietf:wg:oauth:2.0:oob**

クライアントで Web サーバーを起動できない場合 (またはブラウザーが使用できない場合)、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 build of Keycloak 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、ユーザーは別のデバイスを使用してコードを取得し、アプリケーションに貼り付けることができます。

# 第3章 SAML を使用したアプリケーションとサービスのセキュア化

このセクションでは、Red Hat build of Keycloak 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を保護する方法を説明します。

## 3.1. RED HAT BUILD OF KEYCLOAK JAVA アダプター

Red Hat build of Keycloak には、Java アプリケーション用のさまざまなアダプターが付属しています。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. Red Hat JBoss Enterprise Application Platform

#### 3.1.1.1. 8.0 Beta

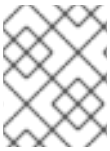
Red Hat build of Keycloak は、Red Hat Enterprise Application Platform 8.0 Beta 用に SAML アダプターを提供します。ただし、現在ドキュメントは利用できず、近日中に追加される予定です。

#### 3.1.1.2. 6.4 と 7.x

Red Hat JBoss Enterprise Application Platform 6.4 および 7.x にデプロイされた既存アプリケーションは、Red Hat build of Keycloak サーバーと組み合わせて Red Hat Single Sign-On 7.6 のアダプターを利用できます。

詳細は、Red Hat Single Sign-On のドキュメント を参照してください。

# 第4章 RED HAT BUILD OF KEYCLOAK を使用するための DOCKER レジストリー設定



**注記**

Docker 認証はデフォルトで無効になっています。有効化する場合は、機能の有効化と無効化 の章を参照してください。

このセクションでは、Red Hat build of Keycloak 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推奨します。Red Hat build of Keycloak 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/realms/master/protocol/docker-v2/auth
    service: docker-test
    issuer: http://localhost:8080/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に関する詳細は、レジストリー設定ファイル仕様 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 build of Keycloak レルムの公開鍵の場所に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。クライアントの詳細から **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/realms/master/protocol/docker-v2/auth
REGISTRY_AUTH_TOKEN_SERVICE: docker-test
REGISTRY_AUTH_TOKEN_ISSUER: http://localhost:8080/realms/master
```

**警告**

**REGISTRY_AUTH_TOKEN_ROOTCERTBUNDLE** を、Red Hat build of Keycloak レルムの公開鍵の場所でオーバーライド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

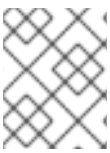
**警告**

このインストール方法は、Red Hat build of Keycloak 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 build of Keycloak 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。

**手順**

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. **Action** メニューから "Docker Compose YAML" オプションを選択し、 **Download adapter config** オプションを選択して 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デプロイメントして、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推奨します。

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 クライアント登録サービスの使用

アプリケーションまたはサービスが Red Hat build of Keycloak を使用するには、Red Hat build of Keycloak 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 build of Keycloak 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 build of Keycloak Client Representations、OpenID Connect Client Meta Data および SAML Entity Descriptors のビルトイン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 **/realms/<realm>/clients-registrations/<provider>** です。

サポートされているビルトイン **providers** は以下のとおりです。

- default - Red Hat build of Keycloak Client Representation (JSON)
- install - Red Hat build of Keycloak Adapter Configuration (JSON)
- openid-connect - OpenID Connect Client Metadata Description (JSON)
- saml2-entity-descriptor - SAML Entity Descriptor (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. AUTHENTICATION

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** ロールのみを持つサービスアカウントのトークンを使用することを推奨します (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 **Client** をクリックし、ページに表示されるタブの **Initial access token** 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。

デフォルトでは、登録アクセストークンのローテーションは有効になっています。これは、登録アクセストークンが 1 回だけ有効であることを意味します。トークンが使用されると、レスポンスには新しいトークンが含まれます。登録アクセストークンのローテーションは、クライアントポリシー を使用して無効化できます。

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 BUILD OF KEYCLOAK の表現

**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 build of Keycloak 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、**/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、**/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、**/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、**/realms/<realm>/clients-registrations/default/<client id>** への HTTP DELETE 要求を実行します。

## 5.3. RED HAT BUILD OF KEYCLOAK のアダプター設定

**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 **/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 build of Keycloak 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 build of Keycloak は、OAuth 2.0 Dynamic Client Registration Protocol および OAuth 2.0 Dynamic Client Registration Management Protocol を拡張する OpenID Connect Dynamic Client Registration を実装します。

Red Hat build of Keycloak 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、**/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 **/realms/<realm>/.well-known/openid-configuration** にも含まれています。

## 5.5. SAML エンティティー記述子

SAML エンティティー記述子エンドポイントは、SAML v2 エンティティー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 build of Keycloak 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 build of Keycloak Client Representation が返されます。

クライアントを作成するには、SAML エンティティー記述子を使用して HTTP POST 要求を **/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ラート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/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ラート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", "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

**注記**

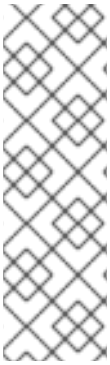
現在の計画では、クライアント登録ポリシーを削除し、代わりに サーバー管理ガイド に記載されているクライアントポリシーを採用する予定です。クライアントポリシーはより柔軟であり、より多くのユースケースに対応します。

現在、Red Hat build of Keycloak 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 build of Keycloak 管理コンソールで、**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リストを設定できます。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、信頼されていな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できる **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 CLI を使用したクライアント登録の自動化

クライアント登録 CLI は、アプリケーション開発者が、Red Hat build of Keycloak 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 build of Keycloak クライアント登録 REST エンドポイントと対話するように特別に設計されています。

Red Hat build of Keycloak 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 build of Keycloak と対話する場合、Red Hat build of Keycloak がログインページ、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るように、アプリケーションはクライアント ID で自身を識別します。

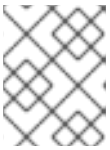
クライアント登録 CLI を使用してコマンドラインからアプリケーションクライアントを設定できます。また、これをシェルスクリプトで使用できます。

特定のユーザーが **Client Registration CLI** を使用できるようにするには、通常、Red Hat build of Keycloak 管理者は、管理コンソールを使用して、適切なロールを持つ新しい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、Client Registration REST API へのアクセスを許可します。

## 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

**手順**

1. 管理コンソール (例: http://localhost:8080/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 Mapping**、**Assign role** を選択します。オプションリストから、**Filter by clients** をクリックします。検索バーに **manage-clients** と入力します。ロールを選択します。または、マスターレルムを使用している場合は、**NAME-realm** のロールを選択します。**NAME** はターゲットレルムの名前です。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. **Assign** をクリックすると、フルセットのクライアント管理パーミッションが付与されます。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。
6. **Available Roles**、**manage-client** を選択して、フル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、管理コンソールの **Initial Access Token** タブ Clients エリアから初期アクセストークンを発行できます。

## 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、新しいレルム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。

**手順**

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、クライアントを作成します (例: **reg-cli**)。
2. **Standard Flow Enable** チェックボックスをオフにします。
3. **Client authentication** を **On** に切り替えてセキュリティーを強化します。
4. 使用するアカウントの種類を選択します。
    a. 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、**Service accounts roles** をオンにします。
    b. 通常のユーザーアカウントを使用する場合は、**Direct access grants** をオンにします。
5. **Next** をクリックします。
6. **Save** をクリックします。
7. **Credentials** タブをクリックします。
**Client Id and Secret** または **Signed JWT** のいずれかを設定します。
8. サービスアカウントロールを使用している場合は、**Service Account Roles** タブをクリックします。
サービスアカウントのアクセス権を設定するためのロールを選択します。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。
9. **Save** をクリックします。

**kcreg config credentials** を実行するときは、**--secret** オプションを使用して、設定したシークレットを指定します。

- **kcreg config credentials** の実行時に使用する **clientId** (例: **--client reg-cli**) を指定します。
- 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 build of Keycloak サーバー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 build of Keycloak 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 build of Keycloak 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

**手順**

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. **Client Registration REST** エンドポイントでコマンドを実行します。
   たとえば、以下のようになります。

   - Linux:

   ```
   $ kcreg.sh config credentials --server http://localhost:8080 --realm demo --user user --client reg-cli
   $ kcreg.sh create -s clientId=my_client -s 'redirectUris=["http://localhost:8980/myapp/*"]'
   $ kcreg.sh get my_client
   ```

   - Windows:

   ```
   c:\> kcreg config credentials --server http://localhost:8080 --realm demo --user user --client reg-cli
   c:\> kcreg create -s clientId=my_client -s "redirectUris=[\"http://localhost:8980/myapp/*\"]"
   c:\> kcreg get my_client
   ```

   **注記**

   実稼働環境では、**https:** で Red Hat build of Keycloak にアクセスする必要があります。そうすることで、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、以下のようになります。

   - 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

**手順**

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 インターフェイスまたは 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、認可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については、「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
```

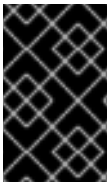
- Windows:

```
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、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

使用する Red Hat build of Keycloak 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、以下を実行します。

```
$ kcreg.sh create -s clientId=myclient -t $TOKEN
```

- Windows:

```
c:\> kcreg config initial-token %TOKEN%
c:\> kcreg create -s clientId=myclient
```

または、以下を実行します。

```
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、上書きすることを推奨します。

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 'redirectUris=["/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リスト表示できます。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、以下のようになります。

- Linux:

```
$ kcreg.sh update myclient -s enabled=false -d redirectUris
```

- Windows:

```
C:\> kcreg update myclient -s enabled=false -d redirectUris
```

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、以下のようになります。

- Linux:

```
$ kcreg.sh update myclient --merge -d redirectUris -f mychanges.json
```

- Windows:

```
C:\> kcreg update myclient --merge -d redirectUris -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。