Red Hat build of Keycloak 22.0

Securing Applications and Services Guide

Last Updated: 2024-06-03

Abstract

This guide consists of information for securing applications and services using Red Hat build of Keycloak 22.0.

Table of Contents

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message .

CHAPTER 1. PLANNING FOR SECURING APPLICATIONS AND SERVICES

As an OAuth2, OpenID Connect, and SAML compliant server, Red Hat build of Keycloak can secure any application and service as long as the technology stack they are using supports any of these protocols. For more details about the security protocols supported by Red Hat build of Keycloak, consider looking at Server Administration Guide.

Most of the support for some of these protocols is already available from the programming language, framework, or reverse proxy they are using. Leveraging the support already available from the application ecosystem is a key aspect to make your application fully compliant with security standards and best practices, so that you avoid vendor lock-in.

For some programming languages, Red Hat build of Keycloak provides libraries that try to fill the gap for the lack of support of a particular security protocol or to provide a more rich and tightly coupled integration with the server. These libraries are known by **Keycloak Client Adapters**, and they should be used as a last resort if you cannot rely on what is available from the application ecosystem.

1.1. BASIC STEPS TO SECURE APPLICATIONS AND SERVICES

These are the basic steps for securing an application or a service in Red Hat build of Keycloak.

1. Register a client to a realm using one of these options:
    - The Red Hat build of Keycloak Admin Console
    - The client registration service
    - The CLI
2. Enable OpenID Connect or SAML protocols in your application using one these options:
    - Leveraging existing OpenID Connect and SAML support from the application ecosystem
    - Using a Red Hat build of Keycloak Adapter

This guide provides the detailed instructions for these steps. You can find more details in the Server Administration Guide about how to register a client to Red Hat build of Keycloak through the administration console.

1.2. GETTING STARTED

The Red Hat build of Keycloak Quickstarts Repository provides examples about how to secure applications and services using different programming languages and frameworks. By going through their documentation and codebase, you will understand the bare minimum changes required in your application and service in order to secure it with Red Hat build of Keycloak.

Also, see the following sections for recommendations for trusted and well-known client-side implementations for both OpenID Connect and SAML protocols.

1.2.1. OpenID Connect

1.2.1.1. JavaScript (client-side)

- JavaScript

1.2.1.2. Node.js (server-side)

- Node.js

1.2.2. SAML

1.2.2.1. Java

- JBoss EAP

1.3. TERMINOLOGY

These terms are used in this guide:

- **Clients** are entities that interact with Red Hat build of Keycloak to authenticate users and obtain tokens. Most often, clients are applications and services acting on behalf of users that provide a single sign-on experience to their users and access other services using the tokens issued by the server. Clients can also be entities only interested in obtaining tokens and acting on their own behalf for accessing other services.
- **Applications** include a wide range of applications that work for specific platforms for each protocol
- **Client adapters** are libraries that make it easy to secure applications and services with Red Hat build of Keycloak. They provide a tight integration to the underlying platform and framework.
- **Creating a client** and **registering a client** are the same action. **Creating a Client** is the term used to create a client by using the Admin Console. **Registering a client** is the term used to register a client by using the Red Hat build of Keycloak Client Registration Service.
- **A service account** is a type of client that is able to obtain tokens on its own behalf.

CHAPTER 2. USING OPENID CONNECT TO SECURE APPLICATIONS AND SERVICES

This section describes how you can secure applications and services with OpenID Connect using Red Hat build of Keycloak.

2.1. AVAILABLE ENDPOINTS

As a fully-compliant OpenID Connect Provider implementation, Red Hat build of Keycloak exposes a set of endpoints that applications and services can use to authenticate and authorize their users.

This section describes some of the key endpoints that your application and service should be use when interacting with Red Hat build of Keycloak.

2.1.1. Endpoints

The most important endpoint to understand is the **well-known** configuration endpoint. It lists endpoints and other configuration options relevant to the OpenID Connect implementation in Red Hat build of Keycloak. The endpoint is:

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

To obtain the full URL, add the base URL for Red Hat build of Keycloak and replace **{realm-name}** with the name of your realm. For example:

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

Some RP libraries retrieve all required endpoints from this endpoint, but for others you might need to list the endpoints individually.

2.1.1.1. Authorization endpoint

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

The authorization endpoint performs authentication of the end-user. This authentication is done by redirecting the user agent to this endpoint.

For more details see the Authorization Endpoint section in the OpenID Connect specification.

2.1.1.2. Token endpoint

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

The token endpoint is used to obtain tokens. Tokens can either be obtained by exchanging an authorization code or by supplying credentials directly depending on what flow is used. The token endpoint is also used to obtain new access tokens when they expire.

For more details, see the Token Endpoint section in the OpenID Connect specification.

2.1.1.3. Userinfo endpoint

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

The userinfo endpoint returns standard claims about the authenticated user; this endpoint is protected by a bearer token.

For more details, see the Userinfo Endpoint section in the OpenID Connect specification.

2.1.1.4. Logout endpoint

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

The logout endpoint logs out the authenticated user.

The user agent can be redirected to the endpoint, which causes the active user session to be logged out. The user agent is then redirected back to the application.

The endpoint can also be invoked directly by the application. To invoke this endpoint directly, the refresh token needs to be included as well as the credentials required to authenticate the client.

2.1.1.5. Certificate endpoint

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

The certificate endpoint returns the public keys enabled by the realm, encoded as a JSON Web Key (JWK). Depending on the realm settings, one or more keys can be enabled for verifying tokens. For more information, see the Server Administration Guide and the JSON Web Key specification.

2.1.1.6. Introspection endpoint

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

The introspection endpoint is used to retrieve the active state of a token. In other words, you can use it to validate an access or refresh token. This endpoint can only be invoked by confidential clients.

For more details on how to invoke on this endpoint, see OAuth 2.0 Token Introspection specification.

2.1.1.7. Dynamic Client Registration endpoint

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

The dynamic client registration endpoint is used to dynamically register clients.

For more details, see the Client Registration chapter and the OpenID Connect Dynamic Client Registration specification.

2.1.1.8. Token Revocation endpoint

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

The token revocation endpoint is used to revoke tokens. Both refresh tokens and access tokens are supported by this endpoint. When revoking a refresh token, the user consent for the corresponding client is also revoked.

For more details on how to invoke on this endpoint, see OAuth 2.0 Token Revocation specification.

2.1.1.9. Device Authorization endpoint

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

The device authorization endpoint is used to obtain a device code and a user code. It can be invoked by confidential or public clients.

For more details on how to invoke on this endpoint, see OAuth 2.0 Device Authorization Grant specification.

2.1.1.10. Backchannel Authentication endpoint

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

The backchannel authentication endpoint is used to obtain an auth_req_id that identifies the authentication request made by the client. It can only be invoked by confidential clients.

For more details on how to invoke on this endpoint, see OpenID Connect Client Initiated Backchannel Authentication Flow specification.

Also refer to other places of Red Hat build of Keycloak documentation like Client Initiated Backchannel Authentication Grant section of this guide and Client Initiated Backchannel Authentication Grant section of Server Administration Guide.

2.2. SUPPORTED GRANT TYPES

This section describes the different grant types available to relaying parties.

2.2.1. Authorization code

The Authorization Code flow redirects the user agent to Red Hat build of Keycloak. Once the user has successfully authenticated with Red Hat build of Keycloak, an Authorization Code is created and the user agent is redirected back to the application. The application then uses the authorization code along with its credentials to obtain an Access Token, Refresh Token and ID Token from Red Hat build of Keycloak.

The flow is targeted towards web applications, but is also recommended for native applications, including mobile applications, where it is possible to embed a user agent.

For more details refer to the Authorization Code Flow in the OpenID Connect specification.

2.2.2. Implicit

The Implicit flow works similarly to the Authorization Code flow, but instead of returning an Authorization Code, the Access Token and ID Token is returned. This approach reduces the need for the extra invocation to exchange the Authorization Code for an Access Token. However, it does not include a Refresh Token. This results in the need to permit Access Tokens with a long expiration; however, that approach is not practical because it is very hard to invalidate these tokens. Alternatively, you can require a new redirect to obtain a new Access Token once the initial Access Token has expired. The Implicit flow is useful if the application only wants to authenticate the user and deals with logout itself.

You can instead use a Hybrid flow where both the Access Token and an Authorization Code are returned.

One thing to note is that both the Implicit flow and Hybrid flow have potential security risks as the Access Token may be leaked through web server logs and browser history. You can somewhat mitigate this problem by using short expiration for Access Tokens.

For more details, see the Implicit Flow in the OpenID Connect specification.

2.2.3. Resource Owner Password Credentials

Resource Owner Password Credentials, referred to as Direct Grant in Red Hat build of Keycloak, allows exchanging user credentials for tokens. Using this flow is not recommended unlesss it is essential. Examples where this flow could be useful are legacy applications and command-line interfaces.

The limitations of using this flow include:

- User credentials are exposed to the application
- Applications need login pages
- Application needs to be aware of the authentication scheme
- Changes to authentication flow requires changes to application
- No support for identity brokering or social login
- Flows are not supported (user self-registration, required actions, and so on.)

For a client to be permitted to use the Resource Owner Password Credentials grant, the client has to have the **Direct Access Grants Enabled** option enabled.

This flow is not included in OpenID Connect, but is a part of the OAuth 2.0 specification.

For more details, see the Resource Owner Password Credentials Grant chapter in the OAuth 2.0 specification.

2.2.3.1. Example using CURL

The following example shows how to obtain an access token for a user in the realm **master** with username **user** and password **password**. The example is using the confidential client **myclient**:

```
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. Client credentials

Client Credentials are used when clients (applications and services) want to obtain access on behalf of themselves rather than on behalf of a user. For example, these credentials can be useful for background services that apply changes to the system in general rather than for a specific user.

Red Hat build of Keycloak provides support for clients to authenticate either with a secret or with public/private keys.

This flow is not included in OpenID Connect, but is a part of the OAuth 2.0 specification.

For more details, see the Client Credentials Grant chapter in the OAuth 2.0 specification.

2.2.5. Device Authorization Grant

Device Authorization Grant is used by clients running on internet-connected devices that have limited input capabilities or lack a suitable browser. . The application requests Red Hat build of Keycloak a device code and a user code. . Red Hat build of Keycloak creates a device code and a user code. . Red Hat build of Keycloak returns a response including the device code and the user code to the application. . The application provides the user with the user code and the verification URI. The user accesses a verification URI to be authenticated by using another browser. . The application repeatedly polls Red Hat build of Keycloak until Red Hat build of Keycloak completes the user authorization. . If user authentication is complete, the application obtains the device code. . The application uses the device code along with its credentials to obtain an Access Token, Refresh Token and ID Token from Red Hat build of Keycloak.

For more details, see the OAuth 2.0 Device Authorization Grant specification .

2.2.6. Client Initiated Backchannel Authentication Grant

Client Initiated Backchannel Authentication Grant is used by clients who want to initiate the authentication flow by communicating with the OpenID Provider directly without redirect through the user's browser like OAuth 2.0's authorization code grant.

The client requests from Red Hat build of Keycloak an auth_req_id that identifies the authentication request made by the client. Red Hat build of Keycloak creates the auth_req_id.

After receiving this auth_req_id, this client repeatedly needs to poll Red Hat build of Keycloak to obtain an Access Token, Refresh Token, and ID Token from Red Hat build of Keycloak in return for the auth_req_id until the user is authenticated.

In case that client uses **ping** mode, it does not need to repeatedly poll the token endpoint, but it can wait for the notification sent by Red Hat build of Keycloak to the specified Client Notification Endpoint. The Client Notification Endpoint can be configured in the Red Hat build of Keycloak Admin Console. The details of the contract for Client Notification Endpoint are described in the CIBA specification.

For more details, see OpenID Connect Client Initiated Backchannel Authentication Flow specification.

Also refer to other places of Red Hat build of Keycloak documentation such as Backchannel Authentication Endpoint of this guide and Client Initiated Backchannel Authentication Grant section of Server Administration Guide. For the details about FAPI CIBA compliance, see the FAPI section of this guide.

2.3. RED HAT BUILD OF KEYCLOAK JAVA ADAPTERS

2.3.1. Red Hat JBoss Enterprise Application Platform

Red Hat build of Keycloak does not include any adapters for Red Hat JBoss Enterprise Application Platform. However, there are alternatives for existing applications deployed to Red Hat JBoss Enterprise Application Platform.

2.3.1.1. 8.0 Beta

Red Hat Enterprise Application Platform 8.0 Beta provides a native OpenID Connect client through the Elytron OIDC client subsystem.

For more information, see the Red Hat JBoss Enterprise Application Platform documentation .

2.3.1.2. 6.4 and 7.x

Existing applications deployed to Red Hat JBoss Enterprise Application Platform 6.4 and 7.x can leverage adapters from Red Hat Single Sign-On 7.6 in combination with the Red Hat build of Keycloak server.

For more information, see the Red Hat Single Sign-On documentation .

2.3.2. Spring Boot adapter

Red Hat build of Keycloak does not include any adapters for Spring Boot. However, there are alternatives for existing applications built with Spring Boot.

Spring Security provides comprehensive support for OAuth 2 and OpenID Connect. For more information, see the Spring Security documentation .

Alternatively, for Spring Boot 2.x the Spring Boot adapter from Red Hat Single Sign-On 7.6 can be used in combination with the Red Hat build of Keycloak server. For more information, see the Red Hat Single Sign-On documentation.

2.4. RED HAT BUILD OF KEYCLOAK JAVASCRIPT ADAPTER

Red Hat build of Keycloak comes with a client-side JavaScript library called **keycloak-js** that can be used to secure web applications. The adapter also comes with built-in support for Cordova applications.

2.4.1. Installation

The adapter is distributed in several ways, but we recommend that you install the **keycloak-js** package from NPM:

```
npm install keycloak-js
```

Alternatively, the library can be retrieved directly from the Red Hat build of Keycloak server at **/js/keycloak.js** and is also distributed as a ZIP archive. We are however considering the inclusion of the adapter directly from the Keycloak server as deprecated, and this functionality might be removed in the future.

2.4.2. Red Hat build of Keycloak server configuration

One important thing to consider about using client-side applications is that the client has to be a public client as there is no secure way to store client credentials in a client-side application. This consideration makes it very important to make sure the redirect URIs you have configured for the client are correct and as specific as possible.

To use the adapter, create a client for your application in the Red Hat build of Keycloak Admin Console. Make the client public by toggling **Client authentication** to **Off** on the **Capability config** page.

You also need to configure **Valid Redirect URIs** and **Web Origins**. Be as specific as possible as failing to do so may result in a security vulnerability.

2.4.3. Using the adapter

The following example shows how to initialize the adapter. Make sure that you replace the options passed to the **Keycloak** constructor with those of the client you have configured.

```
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);
}
```

To authenticate, you call the **login** function. Two options exist to make the adapter automatically authenticate. You can pass **login-required** or **check-sso** to the **init()** function.

- **login-required** authenticates the client if the user is logged in to Red Hat build of Keycloak or displays the login page if the user is not logged in.
- **check-sso** only authenticates the client if the user is already logged in. If the user is not logged in, the browser is redirected back to the application and remains unauthenticated.

You can configure a *silent* **check-sso** option. With this feature enabled, your browser will not perform a full redirect to the Red Hat build of Keycloak server and back to your application, but this action will be performed in a hidden iframe. Therefore, your application resources are only loaded and parsed once by the browser, namely when the application is initialized and not again after the redirect back from Red Hat build of Keycloak to your application. This approach is particularly useful in case of SPAs (Single Page Applications).

To enable the *silent* **check-sso**, you provide a **silentCheckSsoRedirectUri** attribute in the init method. Make sure this URI is a valid endpoint in the application; it must be configured as a valid redirect for the client in the Red Hat build of Keycloak Admin Console:

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

The page at the silent check-sso redirect uri is loaded in the iframe after successfully checking your authentication state and retrieving the tokens from the Red Hat build of Keycloak server. It has no other task than sending the received tokens to the main application and should only look like this:

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

Remember that this page must be served by your application at the specified location in **silentCheckSsoRedirectUri** and is *not* part of the adapter.

WARNING

Silent **check-sso** functionality is limited in some modern browsers. Please see the Modern Browsers with Tracking Protection Section.

To enable **login-required** set **onLoad** to **login-required** and pass to the init method:

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

After the user is authenticated the application can make requests to RESTful services secured by Red Hat build of Keycloak by including the bearer token in the **Authorization** header. For example:

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

    return response.json();
}
```

One thing to keep in mind is that the access token by default has a short life expiration so you may need to refresh the access token prior to sending the request. You refresh this token by calling the **updateToken()** method. This method returns a Promise, which makes it easy to invoke the service only if the token was successfully refreshed and displays an error to the user if it was not refreshed. For example:

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

const users = await fetchUsers();
```

NOTE

Both access and refresh token are stored in memory and are not persisted in any kind of storage. Therefore, these tokens should never be persisted to prevent hijacking attacks.

2.4.4. Session Status iframe

By default, the adapter creates a hidden iframe that is used to detect if a Single-Sign Out has occurred. This iframe does not require any network traffic. Instead the status is retrieved by looking at a special status cookie. This feature can be disabled by setting **checkLoginIframe: false** in the options passed to the **init()** method.

You should not rely on looking at this cookie directly. Its format can change and it's also associated with the URL of the Red Hat build of Keycloak server, not your application.

WARNING

Session Status iframe functionality is limited in some modern browsers. Please see Modern Browsers with Tracking Protection Section.

2.4.5. Implicit and hybrid flow

By default, the adapter uses the Authorization Code flow.

With this flow, the Red Hat build of Keycloak server returns an authorization code, not an authentication token, to the application. The JavaScript adapter exchanges the **code** for an access token and a refresh token after the browser is redirected back to the application.

Red Hat build of Keycloak also supports the Implicit flow where an access token is sent immediately after successful authentication with Red Hat build of Keycloak. This flow may have better performance than the standard flow because no additional request exists to exchange the code for tokens, but it has implications when the access token expires.

However, sending the access token in the URL fragment can be a security vulnerability. For example the token could be leaked through web server logs and or browser history.

To enable implicit flow, you enable the **Implicit Flow Enabled** flag for the client in the Red Hat build of Keycloak Admin Console. You also pass the parameter **flow** with the value **implicit** to **init** method:

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

Note that only an access token is provided and no refresh token exists. This situation means that once the access token has expired, the application has to redirect to Red Hat build of Keycloak again to obtain a new access token.

Red Hat build of Keycloak also supports the Hybrid flow.

This flow requires the client to have both the **Standard Flow** and **Implicit Flow** enabled in the Admin Console. The Red Hat build of Keycloak server then sends both the code and tokens to your application.

The access token can be used immediately while the code can be exchanged for access and refresh tokens. Similar to the implicit flow, the hybrid flow is good for performance because the access token is available immediately. But, the token is still sent in the URL, and the security vulnerability mentioned earlier may still apply.

One advantage in the Hybrid flow is that the refresh token is made available to the application.

For the Hybrid flow, you need to pass the parameter **flow** with value **hybrid** to the **init** method:

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

2.4.6. Hybrid Apps with Cordova

Red Hat build of Keycloak supports hybrid mobile apps developed with Apache Cordova. The adapter has two modes for this: **cordova** and **cordova-native**:

The default is **cordova**, which the adapter automatically selects if no adapter type has been explicitly configured and **window.cordova** is present. When logging in, it opens an InApp Browser that lets the user interact with Red Hat build of Keycloak and afterwards returns to the app by redirecting to **http://localhost**. Because of this behavior, you whitelist this URL as a valid redirect-uri in the client configuration section of the Admin Console.

While this mode is easy to set up, it also has some disadvantages:

- The InApp-Browser is a browser embedded in the app and is not the phone’s default browser. Therefore it will have different settings and stored credentials will not be available.
- The InApp-Browser might also be slower, especially when rendering more complex themes.
- There are security concerns to consider, before using this mode, such as that it is possible for the app to gain access to the credentials of the user, as it has full control of the browser rendering the login page, so do not allow its use in apps you do not trust.

Use this example app to help you get started:
https://github.com/keycloak/keycloak/tree/master/examples/cordova

The alternative mode is`cordova-native`, which takes a different approach. It opens the login page using the system’s browser. After the user has authenticated, the browser redirects back into the application using a special URL. From there, the Red Hat build of Keycloak adapter can finish the login by reading the code or token from the URL.

You can activate the native mode by passing the adapter type **cordova-native** to the **init()** method:

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

This adapter requires two additional plugins:

- cordova-plugin-browsertab: allows the app to open webpages in the system’s browser
- cordova-plugin-deeplinks: allow the browser to redirect back to your app by special URLs

The technical details for linking to an app differ on each platform and special setup is needed. Please refer to the Android and iOS sections of the deeplinks plugin documentation for further instructions.

Different kinds of links exist for opening apps: * custom schemes, such as **myapp://login** or **android-app://com.example.myapp/https/example.com/login** * Universal Links (iOS)) / Deep Links (Android) . While the former are easier to set up and tend to work more reliably, the latter offer extra security because they are unique and only the owner of a domain can register them. Custom-URLs are deprecated on iOS. For best reliability, we recommend that you use universal links combined with a fallback site that uses a custom-url link.

Furthermore, we recommend the following steps to improve compatibility with the adapter:

- Universal Links on iOS seem to work more reliably with **response-mode** set to **query**
- To prevent Android from opening a new instance of your app on redirect add the following snippet to **config.xml**:

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

There is an example app that shows how to use the native-mode: https://github.com/keycloak/keycloak/tree/master/examples/cordova-native

2.4.7. Custom Adapters

In some situations, you may need to run the adapter in environments that are not supported by default, such as Capacitor. To use the JavaScript client in these environments, you can pass a custom adapter. For example, a third-party library could provide such an adapter to make it possible to reliably run the adapter:

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

const keycloak = new Keycloak();

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

This specific package does not exist, but it gives a pretty good example of how such an adapter could be passed into the client.

It’s also possible to make your own adapter, to do so you will have to implement the methods described in the **KeycloakAdapter** interface. For example the following TypeScript code ensures that all the methods are properly implemented:

```
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,
});
```

Naturally you can also do this without TypeScript by omitting the type information, but ensuring implementing the interface properly will then be left entirely up to you.

2.4.8. Modern Browsers with Tracking Protection

In the latest versions of some browsers, various cookies policies are applied to prevent tracking of the users by third parties, such as SameSite in Chrome or completely blocked third-party cookies. Those policies are likely to become more restrictive and adopted by other browsers over time. Eventually cookies in third-party contexts may become completely unsupported and blocked by the browsers. As a result, the affected adapter features might ultimately be deprecated.

The adapter relies on third-party cookies for Session Status iframe, *silent* **check-sso** and partially also for regular (non-silent) **check-sso**. Those features have limited functionality or are completely disabled based on how restrictive the browser is regarding cookies. The adapter tries to detect this setting and reacts accordingly.

2.4.8.1. Browsers with "SameSite=Lax by Default" Policy

All features are supported if SSL / TLS connection is configured on the Red Hat build of Keycloak side as well as on the application side. For example, Chrome is affected starting with version 84.

2.4.8.2. Browsers with Blocked Third-Party Cookies

Session Status iframe is not supported and is automatically disabled if such browser behavior is detected by the adapter. This means the adapter cannot use a session cookie for Single Sign-Out detection and must rely purely on tokens. As a result, when a user logs out in another window, the application using the adapter will not be logged out until the application tries to refresh the Access Token. Therefore, consider setting the Access Token Lifespan to a relatively short time, so that the logout is detected as soon as possible. For more details, see Session and Token Timeouts.

Silent **check-sso** is not supported and falls back to regular (non-silent) **check-sso** by default. This behavior can be changed by setting **silentCheckSsoFallback: false** in the options passed to the **init** method. In this case, **check-sso** will be completely disabled if restrictive browser behavior is detected.

Regular **check-sso** is affected as well. Since Session Status iframe is unsupported, an additional redirect to Red Hat build of Keycloak has to be made when the adapter is initialized to check the user's login status. This check is different from the standard behavior when the iframe is used to tell whether the user is logged in, and the redirect is performed only when the user is logged out.

An affected browser is for example Safari starting with version 13.1.

2.4.9. API Reference

2.4.9.1. Constructor

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

2.4.9.2. Properties

authenticated

Is **true** if the user is authenticated, **false** otherwise.

token

The base64 encoded token that can be sent in the **Authorization** header in requests to services.

tokenParsed

The parsed token as a JavaScript object.

subject

The user id.

idToken

The base64 encoded ID token.

idTokenParsed

The parsed id token as a JavaScript object.

realmAccess

The realm roles associated with the token.

resourceAccess

The resource roles associated with the token.

refreshToken

The base64 encoded refresh token that can be used to retrieve a new token.

refreshTokenParsed

The parsed refresh token as a JavaScript object.

timeSkew

The estimated time difference between the browser time and the Red Hat build of Keycloak server in seconds. This value is just an estimation, but is accurate enough when determining if a token is expired or not.

responseMode

Response mode passed in init (default value is fragment).

flow

Flow passed in init.

adapter

Allows you to override the way that redirects and other browser-related functions will be handled by the library. Available options:

- "default" - the library uses the browser api for redirects (this is the default)
- "cordova" - the library will try to use the InAppBrowser cordova plugin to load keycloak login/registration pages (this is used automatically when the library is working in a cordova ecosystem)

- "cordova-native" - the library tries to open the login and registration page using the phone's system browser using the BrowserTabs cordova plugin. This requires extra setup for redirecting back to the app (see Section 2.4.6, "Hybrid Apps with Cordova").
- "custom" - allows you to implement a custom adapter (only for advanced use cases)

responseType

Response type sent to Red Hat build of Keycloak with login requests. This is determined based on the flow value used during initialization, but can be overridden by setting this value.

2.4.9.3. Methods

init(options)

Called to initialize the adapter.

Options is an Object, where:

- useNonce - Adds a cryptographic nonce to verify that the authentication response matches the request (default is **true**).
- onLoad - Specifies an action to do on load. Supported values are **login-required** or **check-sso**.
- silentCheckSsoRedirectUri - Set the redirect uri for silent authentication check if onLoad is set to 'check-sso'.
- silentCheckSsoFallback - Enables fall back to regular **check-sso** when *silent* **check-sso** is not supported by the browser (default is **true**).
- token - Set an initial value for the token.
- refreshToken - Set an initial value for the refresh token.
- idToken - Set an initial value for the id token (only together with token or refreshToken).
- scope - Set the default scope parameter to the Red Hat build of Keycloak login endpoint. Use a space-delimited list of scopes. Those typically reference Client scopes defined on a particular client. Note that the scope **openid** will always be added to the list of scopes by the adapter. For example, if you enter the scope options **address phone**, then the request to Red Hat build of Keycloak will contain the scope parameter **scope=openid address phone**. Note that the default scope specified here is overwritten if the **login()** options specify scope explicitly.
- timeSkew - Set an initial value for skew between local time and Red Hat build of Keycloak server in seconds (only together with token or refreshToken).
- checkLoginIframe - Set to enable/disable monitoring login state (default is **true**).
- checkLoginIframeInterval - Set the interval to check login state (default is 5 seconds).
- responseMode - Set the OpenID Connect response mode send to Red Hat build of Keycloak server at login request. Valid values are **query** or **fragment**. Default value is **fragment**, which means that after successful authentication will Red Hat build of Keycloak redirect to JavaScript application with OpenID Connect parameters added in URL fragment. This is generally safer and recommended over **query**.
- flow - Set the OpenID Connect flow. Valid values are **standard**, **implicit** or **hybrid**.

- enableLogging - Enables logging messages from Keycloak to the console (default is **false**).
- pkceMethod - The method for Proof Key Code Exchange (PKCE) to use. Configuring this value enables the PKCE mechanism. Available options:
  - "S256" - The SHA256 based PKCE method
- scope - Used to forward the scope parameter to the Red Hat build of Keycloak login endpoint. Use a space-delimited list of scopes. Those typically reference Client scopes defined on a particular client. Note that the scope **openid** is always added to the list of scopes by the adapter. For example, if you enter the scope options **address phone**, then the request to Red Hat build of Keycloak will contain the scope parameter **scope=openid address phone**.
- messageReceiveTimeout - Set a timeout in milliseconds for waiting for message responses from the Keycloak server. This is used, for example, when waiting for a message during 3rd party cookies check. The default value is 10000.
- locale - When onLoad is 'login-required', sets the 'ui_locales' query param in compliance with section 3.1.2.1 of the OIDC 1.0 specification.

Returns a promise that resolves when initialization completes.

login(options)

Redirects to login form.

Options is an optional Object, where:

- redirectUri - Specifies the uri to redirect to after login.
- prompt - This parameter allows to slightly customize the login flow on the Red Hat build of Keycloak server side. For example enforce displaying the login screen in case of value **login**. See Parameters Forwarding Section for the details and all the possible values of the **prompt** parameter.
- maxAge - Used just if user is already authenticated. Specifies maximum time since the authentication of user happened. If user is already authenticated for longer time than **maxAge**, the SSO is ignored and he will need to re-authenticate again.
- loginHint - Used to pre-fill the username/email field on the login form.
- scope - Override the scope configured in **init** with a different value for this specific login.
- idpHint - Used to tell Red Hat build of Keycloak to skip showing the login page and automatically redirect to the specified identity provider instead. More info in the Identity Provider documentation.
- acr - Contains the information about **acr** claim, which will be sent inside **claims** parameter to the Red Hat build of Keycloak server. Typical usage is for step-up authentication. Example of use **{ values: ["silver", "gold"], essential: true }**. See OpenID Connect specification and Step-up authentication documentation for more details.
- action - If the value is **register**, the user is redirected to the registration page. See Registration requested by client section for more details. If the value is **UPDATE_PASSWORD** or another supported required action, the user will be redirected to the reset password page or the other

required action page. However, if the user is not authenticated, the user will be sent to the login page and redirected after authentication. See Application Initiated Action section for more details.

- locale - Sets the 'ui_locales' query param in compliance with section 3.1.2.1 of the OIDC 1.0 specification.
- cordovaOptions - Specifies the arguments that are passed to the Cordova in-app-browser (if applicable). Options **hidden** and **location** are not affected by these arguments. All available options are defined at https://cordova.apache.org/docs/en/latest/reference/cordova-plugin-inappbrowser/. Example of use: **{ zoom: "no", hardwareback: "yes" }**;

createLoginUrl(options)

Returns the URL to login form.

Options is an optional Object, which supports same options as the function **login** .

logout(options)

Redirects to logout.

Options is an Object, where:

- redirectUri - Specifies the uri to redirect to after logout.

createLogoutUrl(options)

Returns the URL to log out the user.

Options is an Object, where:

- redirectUri - Specifies the uri to redirect to after logout.

register(options)

Redirects to registration form. Shortcut for login with option action = 'register'

Options are same as for the login method but 'action' is set to 'register'

createRegisterUrl(options)

Returns the url to registration page. Shortcut for createLoginUrl with option action = 'register'

Options are same as for the createLoginUrl method but 'action' is set to 'register'

accountManagement()

Redirects to the Account Management Console.

createAccountUrl(options)

Returns the URL to the Account Management Console.

Options is an Object, where:

- redirectUri - Specifies the uri to redirect to when redirecting back to the application.

hasRealmRole(role)

Returns true if the token has the given realm role.

hasResourceRole(role, resource)

Returns true if the token has the given role for the resource (resource is optional, if not specified clientId is used).

loadUserProfile()

Loads the users profile.

Returns a promise that resolves with the profile.

For example:

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

isTokenExpired(minValidity)

Returns true if the token has less than minValidity seconds left before it expires (minValidity is optional, if not specified 0 is used).

updateToken(minValidity)

If the token expires within minValidity seconds (minValidity is optional, if not specified 5 is used) the token is refreshed. If -1 is passed as the minValidity, the token will be forcibly refreshed. If the session status iframe is enabled, the session status is also checked.

Returns a promise that resolves with a boolean indicating whether or not the token has been refreshed.

For example:

```
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()

Clear authentication state, including tokens. This can be useful if application has detected the session was expired, for example if updating token fails.

Invoking this results in onAuthLogout callback listener being invoked.

2.4.9.4. Callback Events

The adapter supports setting callback listeners for certain events. Keep in mind that these have to be set before the call to the **init()** method.

For example:

```
keycloak.onAuthSuccess = () => console.log('Authenticated!');
```

The available events are:

- **onReady(authenticated)** - Called when the adapter is initialized.
- **onAuthSuccess** - Called when a user is successfully authenticated.
- **onAuthError** - Called if there was an error during authentication.
- **onAuthRefreshSuccess** - Called when the token is refreshed.
- **onAuthRefreshError** - Called if there was an error while trying to refresh the token.
- **onAuthLogout** - Called if the user is logged out (will only be called if the session status iframe is enabled, or in Cordova mode).
- **onTokenExpired** - Called when the access token is expired. If a refresh token is available the token can be refreshed with updateToken, or in cases where it is not (that is, with implicit flow) you can redirect to the login screen to obtain a new access token.

2.5. RED HAT BUILD OF KEYCLOAK NODE.JS ADAPTER

Red Hat build of Keycloak provides a Node.js adapter built on top of Connect to protect server-side JavaScript apps - the goal was to be flexible enough to integrate with frameworks like Express.js.

To use the Node.js adapter, first you must create a client for your application in the Red Hat build of Keycloak Admin Console. The adapter supports public, confidential, and bearer-only access type. Which one to choose depends on the use-case scenario.

Once the client is created click the **Installation** tab, select **Red Hat build of Keycloak OIDC JSON** for **Format Option**, and then click **Download**. The downloaded **keycloak.json** file should be at the root folder of your project.

2.5.1. Installation

Assuming you've already installed Node.js, create a folder for your application:

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

Use **npm init** command to create a **package.json** for your application. Now add the Red Hat build of Keycloak connect adapter in the dependencies list:

```
    "dependencies": {
      "keycloak-connect": "file:keycloak-connect-22.0.11+redhat-00001.tgz"
    }
```

2.5.2. Usage

Instantiate a Keycloak class

The **Keycloak** class provides a central point for configuration and integration with your application. The simplest creation involves no arguments.

In the root directory of your project create a file called **server.js** and add the following code:

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

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

Install the **express-session** dependency:

```
npm install express-session
```

To start the **server.js** script, add the following command in the 'scripts' section of the **package.json**:

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

Now we have the ability to run our server with following command:

```
npm run start
```

By default, this will locate a file named **keycloak.json** alongside the main executable of your application, in our case on the root folder, to initialize Red Hat build of Keycloak specific settings such as public key, realm name, various URLs.

In that case a Red Hat build of Keycloak deployment is necessary to access Red Hat build of Keycloak admin console.

Please visit links on how to deploy a Red Hat build of Keycloak admin console with Podman or Docker

Now we are ready to obtain the **keycloak.json** file by visiting the Red Hat build of Keycloak Admin Console → clients (left sidebar) → choose your client → Installation → Format Option → Keycloak OIDC JSON → Download

Paste the downloaded file on the root folder of our project.

Instantiation with this method results in all the reasonable defaults being used. As alternative, it's also possible to provide a configuration object, rather than the **keycloak.json** file:

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

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

Applications can also redirect users to their preferred identity provider by using:

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

Configuring a web session store

If you want to use web sessions to manage server-side state for authentication, you need to initialize the **Keycloak(…)** with at least a **store** parameter, passing in the actual session store that **express-session** is using.

```
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 });
```

Passing a custom scope value

By default, the scope value **openid** is passed as a query parameter to Red Hat build of Keycloak's login URL, but you can add an additional custom value:

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

2.5.3. Installing middleware

Once instantiated, install the middleware into your connect-capable app:

In order to do so, first we have to install Express:

```
npm install express
```

then require Express in our project as outlined below:

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

and configure Keycloak middleware in Express, by adding at the code below:

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

Last but not least, let's set up our server to listen for HTTP requests on port 3000 by adding the following code to **main.js**:

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

2.5.4. Configuration for proxies

If the application is running behind a proxy that terminates an SSL connection Express must be configured per the express behind proxies guide. Using an incorrect proxy configuration can result in invalid redirect URIs being generated.

Example configuration:

```
const app = express();

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

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

2.5.5. Protecting resources

Simple authentication

To enforce that a user must be authenticated before accessing a resource, simply use a no-argument version of **keycloak.protect()**:

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

Role-based authorization

To secure a resource with an application role for the current app:

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

To secure a resource with an application role for a **different** app:

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

To secure a resource with a realm role:

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

Resource-Based Authorization

Resource-Based Authorization allows you to protect resources, and their specific methods/actions,** based on a set of policies defined in Keycloak, thus externalizing authorization from your application. This is achieved by exposing a **keycloak.enforcer** method which you can use to protect resources.*

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

The **keycloak-enforcer** method operates in two modes, depending on the value of the **response_mode** configuration option.

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

If **response_mode** is set to **token**, permissions are obtained from the server on behalf of the subject represented by the bearer token that was sent to your application. In this case, a new access token is issued by Keycloak with the permissions granted by the server. If the server did not respond with a token with the expected permissions, the request is denied. When using this mode, you should be able to obtain the token from the request as follows:

```
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
});
```

Prefer this mode when your application is using sessions and you want to cache previous decisions from the server, as well automatically handle refresh tokens. This mode is especially useful for applications acting as a client and resource server.

If **response_mode** is set to **permissions** (default mode), the server only returns the list of granted permissions, without issuing a new access token. In addition to not issuing a new token, this method exposes the permissions granted by the server through the **request** as follows:

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

    // show user profile
});
```

Regardless of the **response_mode** in use, the **keycloak.enforcer** method will first try to check the permissions within the bearer token that was sent to your application. If the bearer token already carries the expected permissions, there is no need to interact with the server to obtain a decision. This is specially useful when your clients are capable of obtaining access tokens from the server with the expected permissions before accessing a protected resource, so they can use some capabilities provided by Keycloak Authorization Services such as incremental authorization and avoid additional requests to the server when **keycloak.enforcer** is enforcing access to the resource.

By default, the policy enforcer will use the **client_id** defined to the application (for instance, via **keycloak.json**) to reference a client in Keycloak that supports Keycloak Authorization Services. In this case, the client can not be public given that it is actually a resource server.

If your application is acting as both a public client(frontend) and resource server(backend), you can use the following configuration to reference a different client in Keycloak with the policies that you want to enforce:

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

It is recommended to use distinct clients in Keycloak to represent your frontend and backend.

If the application you are protecting is enabled with Keycloak authorization services and you have defined client credentials in **keycloak.json**, you can push additional claims to the server and make them available to your policies in order to make decisions. For that, you can define a **claims** configuration option which expects a **function** that returns a JSON with the claims you want to push:

```
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
```

For more details about how to configure Keycloak to protected your application resources, please take a look at the Authorization Services Guide.

Advanced authorization

To secure resources based on parts of the URL itself, assuming a role exists for each section:

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

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

Advanced Login Configuration:

By default, all unauthorized requests will be redirected to the Red Hat build of Keycloak login page unless your client is bearer-only. However, a confidential or public client may host both browsable and API endpoints. To prevent redirects on unauthenticated API requests and instead return an HTTP 401, you can override the redirectToLogin function.

For example, this override checks if the URL contains /api/ and disables login redirects:

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

2.5.6. Additional URLs

Explicit user-triggered logout

By default, the middleware catches calls to **/logout** to send the user through a Red Hat build of Keycloak-centric logout workflow. This can be changed by specifying a **logout** configuration parameter to the **middleware()** call:

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

When the user-triggered logout is invoked a query parameter **redirect_url** can be passed:

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

This parameter is then used as the redirect url of the OIDC logout endpoint and the user will be redirected to **https://example.com/logged/out**.

Red Hat build of Keycloak Admin Callbacks

Also, the middleware supports callbacks from the Red Hat build of Keycloak console to log out a single session or all sessions. By default, these type of admin callbacks occur relative to the root URL of / but can be changed by providing an **admin** parameter to the **middleware()** call:

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

2.5.7. Complete example

A complete example using the Node.js adapter usage can be found in Keycloak quickstarts for Node.js

2.6. FINANCIAL-GRADE API (FAPI) SUPPORT

Red Hat build of Keycloak makes it easier for administrators to make sure that their clients are compliant with these specifications:

- 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)

This compliance means that the Red Hat build of Keycloak server will verify the requirements for the authorization server, which are mentioned in the specifications. Red Hat build of Keycloak adapters do not have any specific support for the FAPI, hence the required validations on the client (application) side may need to be still done manually or through some other third-party solutions.

2.6.1. FAPI client profiles

To make sure that your clients are FAPI compliant, you can configure Client Policies in your realm as described in the Server Administration Guide and link them to the global client profiles for FAPI support, which are automatically available in each realm. You can use either **fapi-1-baseline** or **fapi-1-advanced** profile based on which FAPI profile you need your clients to conform with.

In case you want to use Pushed Authorization Request (PAR), it is recommended that your client use both the **fapi-1-baseline** profile and **fapi-1-advanced** for PAR requests. Specifically, the **fapi-1-baseline** profile contains **pkce-enforcer** executor, which makes sure that client use PKCE with secured S256 algorithm. This is not required for FAPI Advanced clients unless they use PAR requests.

In case you want to use CIBA in a FAPI compliant way, make sure that your clients use both **fapi-1-advanced** and **fapi-ciba** client profiles. There is a need to use the **fapi-1-advanced** profile, or other client profile containing the requested executors, as the **fapi-ciba** profile contains just CIBA-specific executors. When enforcing the requirements of the FAPI CIBA specification, there is a need for more requirements, such as enforcement of confidential clients or certificate-bound access tokens.

2.6.2. Open Finance Brasil Financial-grade API Security Profile

Red Hat build of Keycloak is compliant with the Open Finance Brasil Financial-grade API Security Profile 1.0 Implementers Draft 3. This one is stricter in some requirements than the FAPI 1 Advanced specification and hence it may be needed to configure Client Policies in the more strict way to enforce some of the requirements. Especially:

- If your client does not use PAR, make sure that it uses encrypted OIDC request objects. This can be achieved by using a client profile with the **secure-request-object** executor configured with **Encryption Required** enabled.

- Make sure that for JWS, the client uses the **PS256** algorithm. For JWE, the client should use the **RSA-OAEP** with **A256GCM**. This may need to be set in all the Client Settings where these algorithms are applicable.

2.6.3. TLS considerations

As confidential information is being exchanged, all interactions shall be encrypted with TLS (HTTPS). Moreover, there are some requirements in the FAPI specification for the cipher suites and TLS protocol versions used. To match these requirements, you can consider configure allowed ciphers. This configuration can be done by setting the **https-protocols** and **https-cipher-suites** options. Red Hat build of Keycloak uses **TLSv1.3** by default and hence it is possibly not needed to change the default settings. However it may be needed to adjust ciphers if you need to fall back to lower TLS version for some reason. For more details, see Configuring TLS chapter.

2.7. RECOMMENDATIONS

This section describes some recommendations when securing your applications with Red Hat build of Keycloak.

2.7.1. Validating access tokens

If you need to manually validate access tokens issued by Red Hat build of Keycloak, you can invoke the Introspection Endpoint. The downside to this approach is that you have to make a network invocation to the Red Hat build of Keycloak server. This can be slow and possibly overload the server if you have too many validation requests going on at the same time. Red Hat build of Keycloak issued access tokens are JSON Web Tokens (JWT) digitally signed and encoded using JSON Web Signature (JWS). Because they are encoded in this way, you can locally validate access tokens using the public key of the issuing realm. You can either hard code the realm's public key in your validation code, or lookup and cache the public key using the certificate endpoint with the Key ID (KID) embedded within the JWS. Depending on what language you code in, many third party libraries exist and they can help you with JWS validation.

2.7.2. Redirect URIs

When using the redirect based flows, be sure to use valid redirect uris for your clients. The redirect uris should be as specific as possible. This especially applies to client-side (public clients) applications. Failing to do so could result in:

- Open redirects - this can allow attackers to create spoof links that looks like they are coming from your domain

- Unauthorized entry - when users are already authenticated with Red Hat build of Keycloak, an attacker can use a public client where redirect uris have not be configured correctly to gain access by redirecting the user without the users knowledge

In production for web applications always use **https** for all redirect URIs. Do not allow redirects to http.

A few special redirect URIs also exist:

http://127.0.0.1

This redirect URI is useful for native applications and allows the native application to create a web server on a random port that can be used to obtain the authorization code. This redirect uri allows any port. Note that per OAuth 2.0 for Native Apps, the use of **localhost** is **not** recommended and the IP literal **127.0.0.1** should be used instead.

urn:ietf:wg:oauth:2.0:oob

If you cannot start a web server in the client (or a browser is not available), you can use the special **urn:ietf:wg:oauth:2.0:oob** redirect uri. When this redirect uri is used, Red Hat build of Keycloak displays a page with the code in the title and in a box on the page. The application can either detect that the browser title has changed, or the user can copy and paste the code manually to the application. With this redirect uri, a user can use a different device to obtain a code to paste back to the application.

CHAPTER 3. USING SAML TO SECURE APPLICATIONS AND SERVICES

This section describes how you can secure applications and services with SAML using either Red Hat build of Keycloak client adapters or generic SAML provider libraries.

3.1. RED HAT BUILD OF KEYCLOAK JAVA ADAPTERS

Red Hat build of Keycloak comes with a range of different adapters for Java application. Selecting the correct adapter depends on the target platform.

3.1.1. Red Hat JBoss Enterprise Application Platform

3.1.1.1. 8.0 Beta

Red Hat build of Keycloak provides a SAML adapter for Red Hat Enterprise Application Platform 8.0 Beta. However, the documentation is not currently available, and will be added in the near future.

3.1.1.2. 6.4 and 7.x

Existing applications deployed to Red Hat JBoss Enterprise Application Platform 6.4 and 7.x can leverage adapters from Red Hat Single Sign-On 7.6 in combination with the Red Hat build of Keycloak server.

For more information, see the Red Hat Single Sign-On documentation .

CHAPTER 4. CONFIGURING A DOCKER REGISTRY TO USE RED HAT BUILD OF KEYCLOAK

NOTE

Docker authentication is disabled by default. To enable see the Enabling and disabling features chapter.

This section describes how you can configure a Docker registry to use Red Hat build of Keycloak as its authentication server.

For more information on how to set up and configure a Docker registry, see the Docker Registry Configuration Guide.

4.1. DOCKER REGISTRY CONFIGURATION FILE INSTALLATION

For users with more advanced Docker registry configurations, it is generally recommended to provide your own registry configuration file. The Red Hat build of Keycloak Docker provider supports this mechanism via the *Registry Config File* Format Option. Choosing this option will generate output similar to the following:

```
auth:
  token:
    realm: http://localhost:8080/realms/master/protocol/docker-v2/auth
    service: docker-test
    issuer: http://localhost:8080/realms/master
```

This output can then be copied into any existing registry config file. See the registry config file specification for more information on how the file should be set up, or start with a basic example.

WARNING

Don’t forget to configure the **rootcertbundle** field with the location of the Red Hat build of Keycloak realm’s public key. The auth configuration will not work without this argument.

4.2. DOCKER REGISTRY ENVIRONMENT VARIABLE OVERRIDE INSTALLATION

Often times it is appropriate to use a simple environment variable override for develop or POC Docker registries. While this approach is usually not recommended for production use, it can be helpful when one requires quick-and-dirty way to stand up a registry. Simply use the *Variable Override* Format Option from the client details, and an output should appear like the one below:

```
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
```

WARNING

Don’t forget to configure the **REGISTRY_AUTH_TOKEN_ROOTCERTBUNDLE** override with the location of the Red Hat build of Keycloak realm’s public key. The auth configuration will not work without this argument.

4.3. DOCKER COMPOSE YAML FILE

WARNING

This installation method is meant to be an easy way to get a docker registry authenticating against a Red Hat build of Keycloak server. It is intended for development purposes only and should never be used in a production or production-like environment.

The zip file installation mechanism provides a quickstart for developers who want to understand how the Red Hat build of Keycloak server can interact with the Docker registry. In order to configure:

Procedure

1. From the desired realm, create a client configuration. At this point you will not have a Docker registry - the quickstart will take care of that part.
2. Choose the "Docker Compose YAML" option from the from *Action* menu and select the **Download adapter config** option to download the ZIP file.
3. Unzip the archive to the desired location, and open the directory.
4. Start the Docker registry with **docker-compose up**

NOTE

it is recommended that you configure the Docker registry client in a realm other than 'master', since the HTTP Basic auth flow will not present forms.

Once the above configuration has taken place, and the keycloak server and Docker registry are running, docker authentication should be successful:

```
[user ~]# docker login localhost:5000 -u $username
Password: *******
Login Succeeded
```

CHAPTER 5. USING THE CLIENT REGISTRATION SERVICE

In order for an application or service to utilize Red Hat build of Keycloak it has to register a client in Red Hat build of Keycloak. An admin can do this through the admin console (or admin REST endpoints), but clients can also register themselves through the Red Hat build of Keycloak client registration service.

The Client Registration Service provides built-in support for Red Hat build of Keycloak Client Representations, OpenID Connect Client Meta Data and SAML Entity Descriptors. The Client Registration Service endpoint is **/realms/<realm>/clients-registrations/<provider>**.

The built-in supported **providers** are:

- 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)

The following sections will describe how to use the different providers.

5.1. AUTHENTICATION

To invoke the Client Registration Services you usually need a token. The token can be a bearer token, an initial access token or a registration access token. There is an alternative to register new client without any token as well, but then you need to configure Client Registration Policies (see below).

5.1.1. Bearer token

The bearer token can be issued on behalf of a user or a Service Account. The following permissions are required to invoke the endpoints (see Server Administration Guide for more details):

- create-client or manage-client - To create clients
- view-client or manage-client - To view clients
- manage-client - To update or delete client

If you are using a bearer token to create clients it’s recommend to use a token from a Service Account with only the **create-client** role (see Server Administration Guide for more details).

5.1.2. Initial Access Token

The recommended approach to registering new clients is by using initial access tokens. An initial access token can only be used to create clients and has a configurable expiration as well as a configurable limit on how many clients can be created.

An initial access token can be created through the admin console. To create a new initial access token first select the realm in the admin console, then click on **Client** in the menu on the left, followed by **Initial access token** in the tabs displayed in the page.

You will now be able to see any existing initial access tokens. If you have access you can delete tokens that are no longer required. You can only retrieve the value of the token when you are creating it. To create a new token click on **Create**. You can now optionally add how long the token should be valid, also how many clients can be created using the token. After you click on **Save** the token value is displayed.

It is important that you copy/paste this token now as you won't be able to retrieve it later. If you forget to copy/paste it, then delete the token and create another one.

The token value is used as a standard bearer token when invoking the Client Registration Services, by adding it to the Authorization header in the request. For example:

```
Authorization: bearer eyJhbGciOiJSUz...
```

5.1.3. Registration Access Token

When you create a client through the Client Registration Service the response will include a registration access token. The registration access token provides access to retrieve the client configuration later, but also to update or delete the client. The registration access token is included with the request in the same way as a bearer token or initial access token.

By default, registration access token rotation is enabled. This means a registration access token is only valid once. When the token is used, the response will include a new token. Note that registration access token rotation can be disabled by using Client Policies.

If a client was created outside of the Client Registration Service it won't have a registration access token associated with it. You can create one through the admin console. This can also be useful if you lose the token for a particular client. To create a new token find the client in the admin console and click on **Credentials**. Then click on **Generate registration access token**.

5.2. RED HAT BUILD OF KEYCLOAK REPRESENTATIONS

The **default** client registration provider can be used to create, retrieve, update and delete a client. It uses Red Hat build of Keycloak Client Representation format which provides support for configuring clients exactly as they can be configured through the admin console, including for example configuring protocol mappers.

To create a client create a Client Representation (JSON) then perform an HTTP POST request to **/realms/<realm>/clients-registrations/default**.

It will return a Client Representation that also includes the registration access token. You should save the registration access token somewhere if you want to retrieve the config, update or delete the client later.

To retrieve the Client Representation perform an HTTP GET request to **/realms/<realm>/clients-registrations/default/<client id>**.

It will also return a new registration access token.

To update the Client Representation perform an HTTP PUT request with the updated Client Representation to: **/realms/<realm>/clients-registrations/default/<client id>**.

It will also return a new registration access token.

To delete the Client Representation perform an HTTP DELETE request to: **/realms/<realm>/clients-registrations/default/<client id>**

5.3. RED HAT BUILD OF KEYCLOAK ADAPTER CONFIGURATION

The **installation** client registration provider can be used to retrieve the adapter configuration for a client. In addition to token authentication you can also authenticate with client credentials using HTTP basic authentication. To do this include the following header in the request:

```
Authorization: basic BASE64(client-id + ':' + client-secret)
```

To retrieve the Adapter Configuration then perform an HTTP GET request to **/realms/<realm>/clients-registrations/install/<client id>**.

No authentication is required for public clients. This means that for the JavaScript adapter you can load the client configuration directly from Red Hat build of Keycloak using the above URL.

5.4. OPENID CONNECT DYNAMIC CLIENT REGISTRATION

Red Hat build of Keycloak implements OpenID Connect Dynamic Client Registration, which extends OAuth 2.0 Dynamic Client Registration Protocol and OAuth 2.0 Dynamic Client Registration Management Protocol.

The endpoint to use these specifications to register clients in Red Hat build of Keycloak is **/realms/<realm>/clients-registrations/openid-connect[/<client id>]**.

This endpoint can also be found in the OpenID Connect Discovery endpoint for the realm, **/realms/<realm>/.well-known/openid-configuration**.

5.5. SAML ENTITY DESCRIPTORS

The SAML Entity Descriptor endpoint only supports using SAML v2 Entity Descriptors to create clients. It doesn't support retrieving, updating or deleting clients. For those operations the Red Hat build of Keycloak representation endpoints should be used. When creating a client a Red Hat build of Keycloak Client Representation is returned with details about the created client, including a registration access token.

To create a client perform an HTTP POST request with the SAML Entity Descriptor to **/realms/<realm>/clients-registrations/saml2-entity-descriptor**.

5.6. EXAMPLE USING CURL

The following example creates a client with the clientId **myclient** using CURL. You need to replace **eyJhbGciOiJSUz…** with a proper initial access token or bearer token.

```
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. EXAMPLE USING JAVA CLIENT REGISTRATION API

The Client Registration Java API makes it easy to use the Client Registration Service using Java. To use include the dependency **org.keycloak:keycloak-client-registration-api:>VERSION<** from Maven.

For full instructions on using the Client Registration refer to the JavaDocs. Below is an example of creating a client. You need to replace **eyJhbGciOiJSUz...** with a proper initial access token or bearer token.

```
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. CLIENT REGISTRATION POLICIES

NOTE

The current plans are for the Client Registration Policies to be removed in favor of the Client Policies described in the Server Administration Guide. Client Policies are more flexible and support more use cases.

Red Hat build of Keycloak currently supports two ways how new clients can be registered through Client Registration Service.

- Authenticated requests - Request to register new client must contain either **Initial Access Token** or **Bearer Token** as mentioned above.
- Anonymous requests - Request to register new client doesn’t need to contain any token at all

Anonymous client registration requests are very interesting and powerful feature, however you usually don’t want that anyone is able to register new client without any limitations. Hence we have **Client Registration Policy SPI**, which provide a way to limit who can register new clients and under which conditions.

In Red Hat build of Keycloak admin console, you can click to **Client Registration** tab and then **Client Registration Policies** sub-tab. Here you will see what policies are configured by default for anonymous requests and what policies are configured for authenticated requests.

NOTE

The anonymous requests (requests without any token) are allowed just for creating (registration) of new clients. So when you register new client through anonymous request, the response will contain Registration Access Token, which must be used for Read, Update or Delete request of particular client. However using this Registration Access Token from anonymous registration will be then subject to Anonymous Policy too! This means that for example request for update client also needs to come from Trusted Host if you have **Trusted Hosts** policy. Also for example it won't be allowed to disable **Consent Required** when updating client and when **Consent Required** policy is present etc.

Currently we have these policy implementations:

- Trusted Hosts Policy - You can configure list of trusted hosts and trusted domains. Request to Client Registration Service can be sent just from those hosts or domains. Request sent from some untrusted IP will be rejected. URLs of newly registered client must also use just those trusted hosts or domains. For example it won't be allowed to set **Redirect URI** of client pointing to some untrusted host. By default, there is not any whitelisted host, so anonymous client registration is de-facto disabled.

- Consent Required Policy - Newly registered clients will have **Consent Allowed** switch enabled. So after successful authentication, user will always see consent screen when he needs to approve permissions (client scopes). It means that client won't have access to any personal info or permission of user unless user approves it.

- Protocol Mappers Policy - Allows to configure list of whitelisted protocol mapper implementations. New client can't be registered or updated if it contains some non-whitelisted protocol mapper. Note that this policy is used for authenticated requests as well, so even for authenticated request there are some limitations which protocol mappers can be used.

- Client Scope Policy - Allow to whitelist **Client Scopes**, which can be used with newly registered or updated clients. There are no whitelisted scopes by default; only the client scopes, which are defined as **Realm Default Client Scopes** are whitelisted by default.

- Full Scope Policy - Newly registered clients will have **Full Scope Allowed** switch disabled. This means they won't have any scoped realm roles or client roles of other clients.

- Max Clients Policy - Rejects registration if current number of clients in the realm is same or bigger than specified limit. It's 200 by default for anonymous registrations.

- Client Disabled Policy - Newly registered client will be disabled. This means that admin needs to manually approve and enable all newly registered clients. This policy is not used by default even for anonymous registration.

CHAPTER 6. AUTOMATING CLIENT REGISTRATION WITH THE CLI

The Client Registration CLI is a command-line interface (CLI) tool for application developers to configure new clients in a self-service manner when integrating with Red Hat build of Keycloak. It is specifically designed to interact with Red Hat build of Keycloak Client Registration REST endpoints.

It is necessary to create or obtain a client configuration for any application to be able to use Red Hat build of Keycloak. You usually configure a new client for each new application hosted on a unique host name. When an application interacts with Red Hat build of Keycloak, the application identifies itself with a client ID so Red Hat build of Keycloak can provide a login page, single sign-on (SSO) session management, and other services.

You can configure application clients from a command line with the Client Registration CLI, and you can use it in shell scripts.

To allow a particular user to use **Client Registration CLI** the Red Hat build of Keycloak administrator typically uses the Admin Console to configure a new user with proper roles or to configure a new client and client secret to grant access to the Client Registration REST API.

6.1. CONFIGURING A NEW REGULAR USER FOR USE WITH CLIENT REGISTRATION CLI

Procedure

1. Log in to the Admin Console (for example, http://localhost:8080/admin) as **admin**.
2. Select a realm to administer.
3. If you want to use an existing user, select that user to edit; otherwise, create a new user.
4. Select **Role Mappings > Client Roles > realm-management**If you are in the master realm, select **NAME-realm**, where **NAME** is the name of the target realm. You can grant access to any other realm to users in the master realm.
5. Select **Available Roles > manage-client**to grant a full set of client management permissions. Another option is to choose **view-clients** for read-only or **create-client** to create new clients.

NOTE

These permissions grant the user the capability to perform operations without the use of Initial Access Token or Registration Access Token.

It is possible to not assign any **realm-management** roles to a user. In that case, a user can still log in with the Client Registration CLI but cannot use it without an Initial Access Token. Trying to perform any operations without a token results in a **403 Forbidden** error.

The Administrator can issue Initial Access Tokens from the Admin Console through the **Realm Settings > Client Registration > Initial Access Token** menu.

6.2. CONFIGURING A CLIENT FOR USE WITH THE CLIENT REGISTRATION CLI

By default, the server recognizes the Client Registration CLI as the **admin-cli** client, which is configured automatically for every new realm. No additional client configuration is necessary when logging in with a user name.

Procedure

1. Create a client (for example, **reg-cli**) if you want to use a separate client configuration for the Client Registration CLI.
2. Toggle the **Standard Flow Enabled** setting it to **Off**.
3. Strengthen the security by configuring the client **Access Type** as **Confidential** and selecting **Credentials > ClientId and Secret**

   NOTE

   You can configure either **Client Id and Secret** or **Signed JWT** under the **Credentials** tab .

4. Enable service accounts if you want to use a service account associated with the client by selecting a client to edit in the **Clients** section of the **Admin Console**.
   a. Under **Settings**, change the **Access Type** to **Confidential**, toggle the **Service Accounts Enabled** setting to **On**, and click **Save**.
   b. Click **Service Account Roles** and select desired roles to configure the access for the service account. For the details on what roles to select, see Section 6.1, "Configuring a new regular user for use with Client Registration CLI".
5. Toggle the **Direct Access Grants Enabled** setting it to **On** if you want to use a regular user account instead of a service account.
6. If the client is configured as **Confidential**, provide the configured secret when running **kcreg config credentials** by using the **--secret** option.
7. Specify which **clientId** to use (for example, **--client reg-cli**) when running **kcreg config credentials**.
8. With the service account enabled, you can omit specifying the user when running **kcreg config credentials** and only provide the client secret or keystore information.

6.3. INSTALLING THE CLIENT REGISTRATION CLI

The Client Registration CLI is packaged inside the Red Hat build of Keycloak Server distribution. You can find execution scripts inside the **bin** directory. The Linux script is called **kcreg.sh**, and the Windows script is called **kcreg.bat**.

Add the Red Hat build of Keycloak server directory to your **PATH** when setting up the client for use from any location on the file system.

For example, on:

- Linux:

```
$ export PATH=$PATH:$KEYCLOAK_HOME/bin
$ kcreg.sh
```

- Windows:

```
c:\> set PATH=%PATH%;%KEYCLOAK_HOME%\bin
c:\> kcreg
```

KEYCLOAK_HOME refers to a directory where the Red Hat build of Keycloak Server distribution was unpacked.

6.4. USING THE CLIENT REGISTRATION CLI

Procedure

1. Start an authenticated session by logging in with your credentials.
2. Run commands on the **Client Registration REST** endpoint.
   For example, on:
   - 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
     ```

   NOTE

   In a production environment, Red Hat build of Keycloak has to be accessed with **https:** to avoid exposing tokens to network sniffers.

3. If a server's certificate is not issued by one of the trusted certificate authorities (CAs) that are included in Java's default certificate truststore, prepare a **truststore.jks** file and instruct the Client Registration CLI to use it.
   For example, on:
   - Linux:

     ```
     $ kcreg.sh config truststore --trustpass $PASSWORD ~/.keycloak/truststore.jks
     ```

   - Windows:

     ```
     c:\> kcreg config truststore --trustpass %PASSWORD%
     %HOMEPATH%\.keycloak\truststore.jks
     ```

6.4.1. Logging in

Procedure

1. Specify a server endpoint URL and a realm when you log in with the Client Registration CLI.
2. Specify a user name or a client id, which results in a special service account being used. When using a user name, you must use a password for the specified user. When using a client ID, you use a client secret or a **Signed JWT** instead of a password.

Regardless of the login method, the account that logs in needs proper permissions to be able to perform client registration operations. Keep in mind that any account in a non-master realm can only have permissions to manage clients within the same realm. If you need to manage different realms, you can either configure multiple users in different realms, or you can create a single user in the **master** realm and add roles for managing clients in different realms.

You cannot configure users with the Client Registration CLI. Use the Admin Console web interface or the Admin Client CLI to configure users. See Server Administration Guide for more details.

When **kcreg** successfully logs in, it receives authorization tokens and saves them in a private configuration file so the tokens can be used for subsequent invocations. See Section 6.4.2, "Working with alternative configurations" for more information on configuration files.

See the built-in help for more information on using the Client Registration CLI.

For example, on:

- Linux:

```
$ kcreg.sh help
```

- Windows:

```
c:\> kcreg help
```

See **kcreg config credentials --help** for more information about starting an authenticated session.

6.4.2. Working with alternative configurations

By default, the Client Registration CLI automatically maintains a configuration file at a default location, **./.keycloak/kcreg.config**, under the user's home directory. You can use the **--config** option to point to a different file or location to maintain multiple authenticated sessions in parallel. It is the safest way to perform operations tied to a single configuration file from a single thread.

IMPORTANT

Do not make the configuration file visible to other users on the system. The configuration file contains access tokens and secrets that should be kept private.

You might want to avoid storing secrets inside a configuration file by using the **--no-config** option with all of your commands, even though it is less convenient and requires more token requests to do so. Specify all authentication information with each **kcreg** invocation.

6.4.3. Initial Access and Registration Access Tokens

Developers who do not have an account configured at the Red Hat build of Keycloak server they want to use can use the Client Registration CLI. This is possible only when the realm administrator issues a developer an Initial Access Token. It is up to the realm administrator to decide how and when to issue and distribute these tokens. The realm administrator can limit the maximum age of the Initial Access Token and the total number of clients that can be created with it.

Once a developer has an Initial Access Token, the developer can use it to create new clients without authenticating with **kcreg config credentials**. The Initial Access Token can be stored in the configuration file or specified as part of the **kcreg create** command.

For example, on:

- Linux:

```
$ kcreg.sh config initial-token $TOKEN
$ kcreg.sh create -s clientId=myclient
```

or

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

- Windows:

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

or

```
c:\> kcreg create -s clientId=myclient -t %TOKEN%
```

When using an Initial Access Token, the server response includes a newly issued Registration Access Token. Any subsequent operation for that client needs to be performed by authenticating with that token, which is only valid for that client.

The Client Registration CLI automatically uses its private configuration file to save and use this token with its associated client. As long as the same configuration file is used for all client operations, the developer does not need to authenticate to read, update, or delete a client that was created this way.

See Client Registration for more information about Initial Access and Registration Access Tokens.

Run the **kcreg config initial-token --help** and **kcreg config registration-token --help** commands for more information on how to configure tokens with the Client Registration CLI.

6.4.4. Creating a client configuration

The first task after authenticating with credentials or configuring an Initial Access Token is usually to create a new client. Often you might want to use a prepared JSON file as a template and set or override some of the attributes.

The following example shows how to read a JSON file, override any client id it may contain, set any other attributes, and print the configuration to a standard output after successful creation.

- 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
```

Run the **kcreg create --help** for more information about the **kcreg create** command.

You can use **kcreg attrs** to list available attributes. Keep in mind that many configuration attributes are not checked for validity or consistency. It is up to you to specify proper values. Remember that you should not have any id fields in your template and should not specify them as arguments to the **kcreg create** command.

6.4.5. Retrieving a client configuration

You can retrieve an existing client by using the **kcreg get** command.

For example, on:

- Linux:

```
$ kcreg.sh get myclient
```

- Windows:

```
C:\> kcreg get myclient
```

You can also retrieve the client configuration as an adapter configuration file, which you can package with your web application.

For example, on:

- Linux:

```
$ kcreg.sh get myclient -e install > keycloak.json
```

- Windows:

```
C:\> kcreg get myclient -e install > keycloak.json
```

Run the **kcreg get --help** command for more information about the **kcreg get** command.

6.4.6. Modifying a client configuration

There are two methods for updating a client configuration.

One method is to submit a complete new state to the server after getting the current configuration, saving it to a file, editing it, and posting it back to the server.

For example, on:

- 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
```

The second method fetches the current client, sets or deletes fields on it, and posts it back in one step.

For example, on:

- Linux:

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

- Windows:

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

You can also use a file that contains only changes to be applied so you do not have to specify too many values as arguments. In this case, specify **--merge** to tell the Client Registration CLI that rather than treating the JSON file as a full, new configuration, it should treat it as a set of attributes to be applied over the existing configuration.

For example, on:

- Linux:

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

- Windows:

```
C:\> kcreg update myclient --merge -d redirectUris -f mychanges.json
```

Run the **kcreg update --help** command for more information about the **kcreg update** command.

6.4.7. Deleting a client configuration

Use the following example to delete a client.

- Linux:

```
$ kcreg.sh delete myclient
```

- Windows:

```
C:\> kcreg delete myclient
```

Run the **kcreg delete --help** command for more information about the **kcreg delete** command.

6.4.8. Refreshing invalid Registration Access Tokens

When performing a create, read, update, and delete (CRUD) operation using the **--no-config** mode, the Client Registration CLI cannot handle Registration Access Tokens for you. In that case, it is possible to lose track of the most recently issued Registration Access Token for a client, which makes it impossible to perform any further CRUD operations on that client without authenticating with an account that has **manage-clients** permissions.

If you have permissions, you can issue a new Registration Access Token for the client and have it printed to a standard output or saved to a configuration file of your choice. Otherwise, you have to ask the realm administrator to issue a new Registration Access Token for your client and send it to you. You can then pass it to any CRUD command via the **--token** option. You can also use the **kcreg config registration-token** command to save the new token in a configuration file and have the Client Registration CLI automatically handle it for you from that point on.

Run the **kcreg update-token --help** command for more information about the **kcreg update-token** command.

6.5. TROUBLESHOOTING

- Q: When logging in, I get an error: **Parameter client_assertion_type is missing [invalid_client]**.
  A: This error means your client is configured with **Signed JWT** token credentials, which means you have to use the **--keystore** parameter when logging in.