

Keycloak - Identity and Access Management for Modern Applications

Harness the power of Keycloak, OpenID Connect, and OAuth 2.0 protocols to secure applications



Keycloak - Identity and Access Management for Modern Applications

Harness the power of Keycloak, OpenID Connect, and OAuth 2.0 protocols to secure applications

Stian Thorgersen

Pedro Igor Silva



BIRMINGHAM—MUMBAI

Keycloak - Identity and Access Management for Modern Applications

Copyright © 2021 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Group Product Manager: Wilson D'souza

Publishing Product Manager: Yogesh Deokar

Senior Editor: Shazeen Iqbal

Content Development Editor: Romy Dias

Technical Editor: Sarvesh Jayant

Copy Editor: Safis Editing

Project Coordinator: Shagun Saini

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Production Designer: Aparna Bhagat

First published: May 2021

Production reference: 1120521

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80056-249-3

www.packt.com

Contributors

About the authors

Stian Thorgersen started his career at Arjuna Technologies building a cloud federation platform, years before most companies were even ready for a single-vendor public cloud. He later joined Red Hat, looking for ways to make developers' lives easier, which is where the idea of Keycloak started. In 2013, Stian co-founded the Keycloak project with another developer at Red Hat.

Today, Stian is the Keycloak project lead and is also the top contributor to the project. He is still employed by Red Hat as a senior principal software engineer focusing on identity and access management, both for Red Hat and for Red Hat's customers.

In his spare time, there is nothing Stian likes more than throwing his bike down the mountains of Norway.

Pedro Igor Silva is a proud dad of amazing girls. He started his career back in 2000 at an ISP, where he had his first experiences with open source projects such as FreeBSD and Linux, as well as a Java and J2EE software engineer. Since then, he has worked in different IT companies as a system engineer, system architect, and consultant.

Today, Pedro Igor is a principal software engineer at Red Hat and one of the core developers of Keycloak. His main area of interest and study is now IT security, specifically in the application security and identity and access management spaces.

In his non-working hours, he takes care of his planted aquariums.

I want to thank my wonderful family for giving me the space and support I've needed to write this book. The whole Packt editing team has helped this first-time author immensely, but I'd like to give a special thanks to Romy Dias, who edited most of my work.

6

Securing Different Application Types

In this chapter, we will first begin by understanding whether the application we want to secure is an internal or external application. Then, we will look at how to secure a range of different application types, including web, native, and mobile applications. We will also look at how to secure REST APIs and other types of services with bearer tokens.

By the end of this chapter, you will have learned the principles and best practices behind securing different types of applications. You will understand how to secure web, mobile, and native applications, as well as how bearer tokens can be used to protect any type of service, including REST APIs, gRPC, WebSocket, and other types of services.

In this chapter, we're going to cover the following main topics:

- Understanding internal and external applications
- Securing web applications
- Securing native and mobile applications
- Securing REST APIs and services

Technical requirements

To run the sample application included in this chapter, you need to have Node.js (<https://nodejs.org/>) installed on your workstation.

You also need to have a local copy of the GitHub repository associated with the book. If you have Git installed, you can clone the repository by running this command in a terminal:

```
$ git clone https://github.com/PacktPublishing/Keycloak-Identity-and-Access-Management-for-Modern-Applications.git
```

Alternatively, you can download a ZIP of the repository from <https://github.com/PacktPublishing/Keycloak-Identity-and-Access-Management-for-Modern-Applications/archive/master.zip>.

Check out the following link to see the Code in Action video:

<https://bit.ly/3b5R0F2>

Understanding internal and external applications

When securing an application, the first thing to consider is whether the application is an internal application or an external application.

Internal applications, sometimes referred to as first-party applications, are applications owned by the enterprise. It does not matter who developed the application, nor does it matter how it is hosted. The application could be an off-the-shelf application, and it can also be a **Software as a Service (SaaS)**-hosted application, while still being considered an internal application.

For an internal application, there is no need to ask the user to grant access to the application when authenticating to the user, as this application is trusted and the administrator that registered the application with Keycloak can pre-approve the access on behalf of the user. In Keycloak, this is done by turning off the **Consent Required** option for the client, as shown in the following screenshot:

The screenshot shows the Myrealm console interface. On the left is a dark sidebar with a 'Myrealm' header and a 'Configure' section containing links to 'Realm Settings', 'Clients', 'Client Scopes', 'Roles', 'Identity Providers', 'User Federation', and 'Authentication'. The 'Clients' link is highlighted. The main area is titled 'Clients > internal-application' and 'Internal-application' with a trash icon. Below this is a tabbed interface with 'Settings' selected. The settings include: 'Client ID' (internal-application), 'Name' (Internal Application), 'Description' (An application owned by the enterprise), 'Enabled' (ON), and 'Consent Required' (OFF).

Figure 6.1 – Internal application configured to not require consent

When a user authenticates or grants access to an internal application, the user is only required to enter the username and password. For external applications, on the other hand, a user must also grant access to the application.

External applications, sometimes referred to as third-party applications, are applications that are not owned and managed by the enterprise itself, but rather by a third party. All external applications should have the **Consent Required** option enabled, as shown in the following screenshot:

The screenshot shows the Myrealm console interface for an external application. The sidebar is identical to the previous screenshot. The main area is titled 'Clients > external-application' and 'External-application' with a trash icon. The 'Settings' tab is selected. The settings include: 'Client ID' (external-application), 'Name' (External Application), 'Description' (An application managed by a third-party), 'Enabled' (ON), and 'Consent Required' (ON).

Figure 6.2 – External application configured to require consent

When a user authenticates or grants access to an external application, the user is required to not only enter the username and password but also to grant access to the application, as shown in the following screenshot:

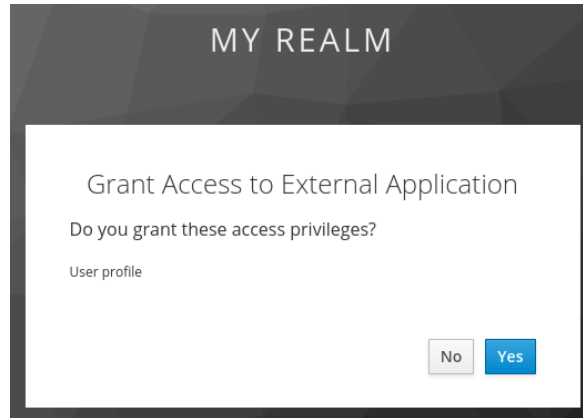


Figure 6.3 – User granting access to an external application

You should now understand the difference between an internal and an external application, including how to require users to grant access to external applications. In the next section, we will look at how to secure web applications with Keycloak.

Securing web applications

When securing a web application with Keycloak, the first thing you should consider is the architecture of the application as there are multiple approaches:

- First and foremost, is your web application a traditional web application running on the server side or a modern **single-page application (SPA)** running in the browser?
- The second thing to consider is whether the application is accessing any REST APIs, and if so, are the REST APIs a part of the application or external?

If it is a SPA-type application invoking external APIs, then there are two further options to consider. Does the application invoke the external REST API directly, or through a dedicated REST API hosted alongside the application?

Based on this, you should determine which of the following matches the architecture of the application you are securing:

- **Server side:** If the web application is running inside a web server or an application server.
- **SPA with dedicated REST API:** If the application is running in the browser and is only invoking a dedicated REST API under the same domain.
- **SPA with intermediary API:** If the application is running in the browser and invokes external REST APIs through an intermediary API, where the intermediary API is hosted under the same domain as the application
- **SPA with external API:** If the application is running in the browser and only invokes APIs hosted under different domains.

Before we take a look at details specific to these different web application architectures, let's consider what is common among all architectures.

Firstly, and most importantly, you should secure your web application using the Authorization Code flow with the **Proof Key for Code Exchange (PKCE)** extension. If you are not sure what the Authorization Code flow is, you should read *Chapter 4, Authenticating Users with OpenID Connect*, before continuing with this chapter. The PKCE extension is an extension to OAuth 2.0 that binds the authorization code to the application that sent the authorization request. This prevents abuse of the authorization code if it is intercepted. We are not covering PKCE in detail in this book, as we recommend you use a library. If you do decide not to use a library, you should refer to the specifications on how to implement support for OAuth 2.0 and OpenID Connect yourself.

When porting existing applications to use Keycloak, it may be tempting to keep the login pages in the existing application, then exchanging the username and password for tokens, by using the Resource Owner Password Credential grant to obtain tokens. This would be similar to how you would integrate your application with an LDAP server.

However, this is simply something that you should not be tempted to do. Collecting user credentials in an application effectively means that if a single application is compromised, an attacker would likely have access to all applications that the user can access. This includes applications not secured by Keycloak, as users often reuse passwords. You also do not have the ability to introduce stronger authentication, such as two-factor authentication. Finally, you do not get all the benefits of using Keycloak with this approach, such as **single sign-on (SSO)** and social login.

As an alternative to keeping the login pages within your existing applications, you may be tempted to embed the Keycloak login page as an iframe within the application. This is also something that you should avoid doing. With the login page embedded into the application, it can be affected by vulnerabilities in an application, potentially allowing an attacker access to the username and password.

As the login page is rendered within an iframe, it is also not easy for users to see where the login pages are coming from, and users may not trust entering their passwords into the application directly. Finally, with third-party cookies being frequently used for tracking across multiple sites, browsers are becoming more and more aggressive against blocking third-party cookies, which may result in the Keycloak login pages not having access to the cookies it needs to function.

In summary, you should get used to the fact that an application should redirect the user to a trusted identity provider for authentication, especially in SSO scenarios. This is also a pattern that most of your users will already be familiar with as it is widely in use nowadays. The following screenshot shows an example of the Google and Amazon login pages, where you can see that they are not embedded in the applications themselves:

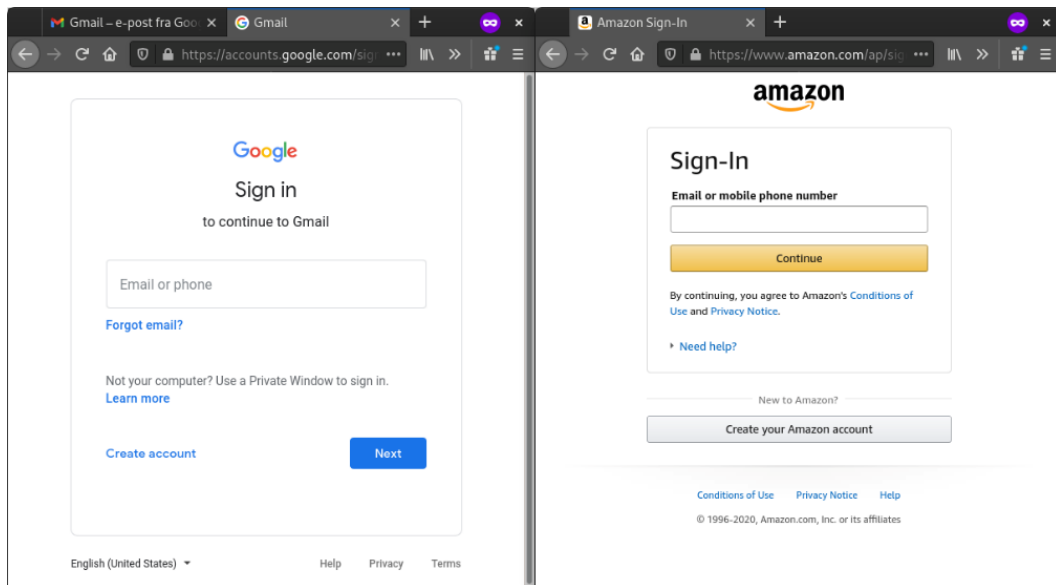


Figure 6.4 – Example from Google and Amazon showing external login pages

You should now have a good, basic understanding of how to go about securing a web application with Keycloak. In the next section, we will start looking at how to secure different types of web applications, starting with server-side web applications.

Securing server-side web applications

When securing a server-side web application with Keycloak, you should register a confidential client with Keycloak. As you are using a confidential client, a leaked authorization code can't be leveraged by an attacker. It is still good practice to leverage the PKCE extension as it provides protection against other types of attacks.

You must also configure applicable redirect URIs for the client as otherwise, you are creating what is called an open redirect. An open redirect can be used, for example, in a spamming attack to make a user believe they are clicking on a link to a trusted site. As an example, if a spammer sends the `https://trusted-site.com/...?redirect_uri=https://attacker.com` URL to a user in an email, the user may only notice the domain name is to a trusted site and click on the link. If you have not configured an exact redirect URI for your client, Keycloak would end up redirecting the user to the site provided by the attacker.

With a server-side web application, usually, only the ID token is leveraged to establish an HTTP session. The server-side web application can also leverage an access token if it wants to invoke external REST APIs under the context of the user.

The following diagram shows the flow for a server-side web application:

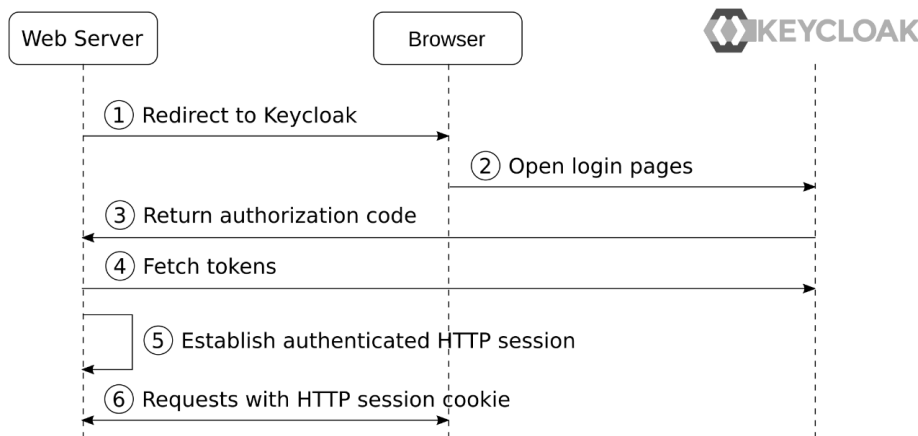


Figure 6.5 – Server-side web application

In more detail, the steps in the diagram are as follows:

1. The **web server** redirects the browser to the **Keycloak** login pages using the **Authorization Code** flow
2. The user authenticates with Keycloak.
3. The authorization code is returned to the server-side web application.
4. The application exchanges the authorization code for tokens, using the credentials registered with the client in Keycloak.
5. The application retrieves the ID token directly from Keycloak as it does not need to verify the token, and can directly parse the ID token to find out information about the authenticated user, and establish an authenticated HTTP session.
6. Requests from the browser now include the HTTP session cookie.

In summary, the application leverages the Authorization Code flow to obtain an ID token from Keycloak, which it uses to establish an authenticated HTTP session.

For server-side web applications, you can also choose to use SAML 2.0, rather than using OpenID Connect. As OpenID Connect is generally easier to work with, it is recommended to use OpenID Connect rather than SAML 2.0, unless your application already supports SAML 2.0.

You should now have a good understanding of how to secure a server-side web application with Keycloak. In the next section, we will look at web applications running on the client side, starting with SPAs that have their own dedicated REST API backend.

Securing a SPA with a dedicated REST API

A SPA that has a dedicated REST API on the same domain should be secured with Keycloak in the same way as a server-side web application. As the application has a dedicated REST API, it should leverage the Authorization Code flow with a confidential client for the highest level of security, and use an HTTP session to secure the API requests from the client side to the dedicated REST API.

The following diagram shows the flow for a SPA with a dedicated REST API:

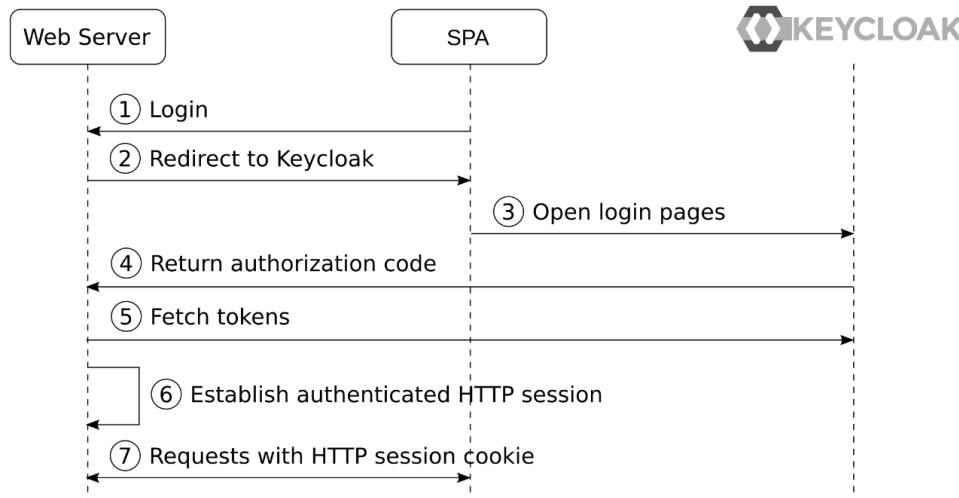


Figure 6.6 – A SPA with a dedicated REST API

In more detail, the steps in the diagram are as follows:

1. The user clicks on the login link in the application, which sends a request to the web server.
2. The web server redirects to the Keycloak login pages.
3. The user authenticates with Keycloak.
4. The authorization code is returned to the web server.
5. The web server exchanges the authorization code for tokens.
6. As the application retrieves the ID token directly from Keycloak, it does not need to verify the token, and can directly parse the ID token to find out information about the authenticated user, and establish an authenticated HTTP session.

Requests from the SPA to the dedicated REST API include the HTTP session cookie. In summary, the application leverages the Authorization Code flow to obtain an ID token from Keycloak, which it uses to establish an authenticated HTTP session, which enables the SPA to securely invoke the REST API provided by the web server.

You should now have a good understanding of how to go about securing a SPA when there is a dedicated REST API hosted on the same domain. In the next section, we will look at a SPA where an external REST API is invoked, but it is done through a backend REST API hosted on the same domain as the SPA.

Securing a SPA with an intermediary REST API

The most secure way to invoke external REST APIs from a SPA is through an intermediary API hosted on the same domain as the SPA. By doing this, you are able to leverage a confidential client and tokens are not directly accessible in the browser, which reduces the risk of tokens, especially the refresh token, being leaked.

This type of SPA is often referred to as the backend for frontends patterns. Not only does it have increased security, but it also makes your SPA more portable and may make it easier to develop. This is due to the application not having to directly deal with external APIs, but rather a dedicated REST API built specifically to service the frontend SPA.

Further, by default, browsers do not allow a SPA to invoke a REST API on a different domain unless **Cross-Origin Resource Sharing (CORS)** is enabled. CORS enables a REST API to return special HTTP headers that tell the browser a request from a different origin is permitted. As the SPA is making the requests through an intermediary REST API on the same domain, you don't need to deal with CORS in this case.

The following diagram shows the flow for a SPA with an intermediary REST API:

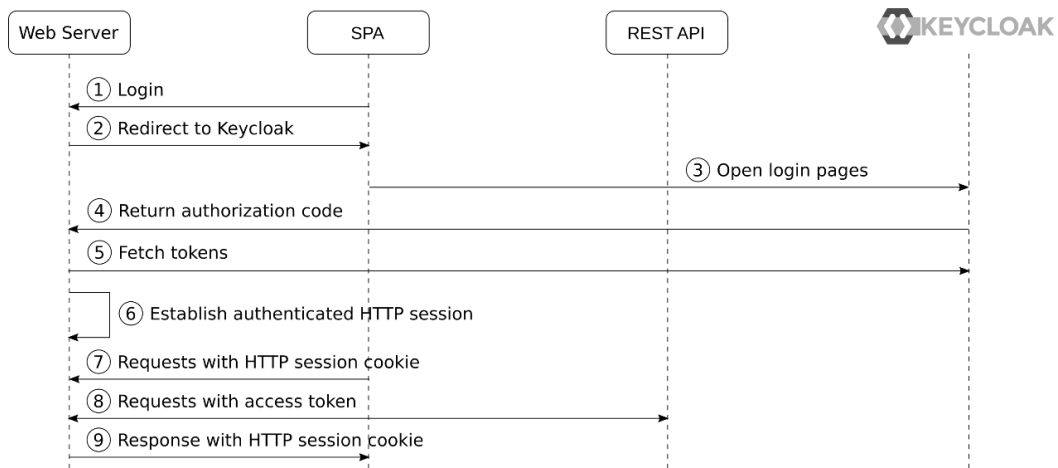


Figure 6.7 – SPA with an intermediary REST API

In more detail, the steps in the diagram are as follows:

1. The user clicks on the login link in the application, which sends a request to the web server.
2. The web server redirects to the Keycloak login pages.
3. The user authenticates with Keycloak.
4. The authorization code is returned to the web server.

5. The web server exchanges the authorization code for tokens.
6. As the web server retrieves the ID token directly from Keycloak, it does not need to verify the token, and can directly parse the ID token to find out information about the authenticated user, and establish an authenticated HTTP session. The refresh token and access token are stored within the HTTP session.
7. Requests from the SPA to the dedicated REST API includes the HTTP session cookie.
8. The web server retrieves the access token from the HTTP session and includes it in requests to the external REST API.
9. The web server returns the response to the SPA, including the HTTP session cookie.

In summary, the application leverages the Authorization Code flow to obtain an ID token from Keycloak, which it uses to establish an authenticated HTTP session, which enables the SPA to securely invoke the web server, which in turn proxies the request to the external REST API.

You should now have a good understanding of how to secure a SPA with an intermediary API hosted on the same domain, which is leveraged to invoke external REST APIs.

In the next section, we will look at a SPA where there is no REST API hosted on the same domain.

Securing a SPA with an external REST API

The simplest way to secure a SPA with Keycloak is by doing the Authorization Code flow directly from the SPA itself with a public client registered in Keycloak. This is a somewhat less secure approach as the tokens, including the refresh token, are exposed directly to the browser. For very critical applications, such as financial applications, this is not an approach you want to use. However, there are a number of techniques that can be leveraged to provide a good level of security for this approach, such as the following:

- Have a short expiration for the refresh token. In Keycloak, this is configured by setting the client session timeouts for the client. This makes it possible to configure a client to for example have refresh tokens that are valid for 30 minutes, while the SSO session can be valid for several days.
- Rotate refresh tokens. In Keycloak, this is configured by enabling **Revoke Refresh Token** for the realm, which results in previously used refresh tokens being invalidated. If an invalid refresh token is used, the session is invalidated. This would result in a leaked refresh token being quickly invalidated as both the SPA and the attacker would try to use the refresh token, resulting in it being invalidated.

- Use the PKCE extension. For a public client, using the PKCE extension is required; otherwise, there is a high chance that a leaked authorization code can be used by an attacker to obtain tokens.
- Store tokens in the window state or HTML5 storage session, and avoid using easily guessable keys such as `window.sessionStorage.accessToken`.
- Protect the SPA from **Cross-Site Scripting (XSS)** and other attacks by following best practices from the **Open Web Application Security Project (OWASP)**.
- Be careful when using third-party scripts in your application.

At the end of the day, this is a trade-off that you will have to decide for yourself. Are you comfortable with the risk of tokens being leaked, and have you made sure your SPA is secure? If so, then using this approach provides you with a simpler solution and also removes the need to have a dedicated backend for your SPA, which also reduces the cost of maintaining the application.

The following diagram shows the flow for a SPA with an external REST API:

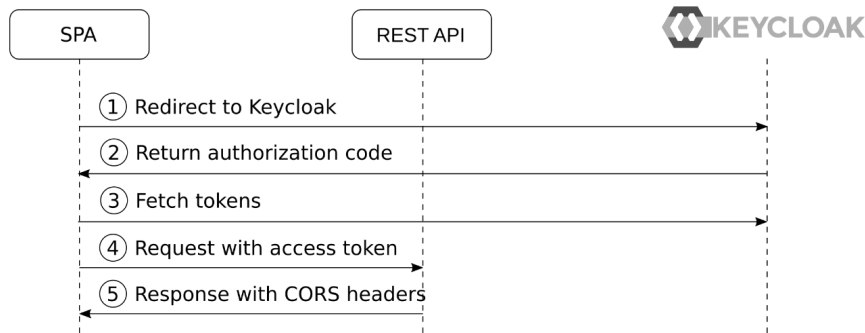


Figure 6.8 – A SPA with an external REST API

In more detail, the steps in the diagram are as follows:

1. The SPA redirects to the Keycloak login pages.
2. After the user has authenticated, the authorization code is returned to the SPA.
3. The SPA exchanges the authorization code for tokens. As the SPA is running in the browser, it does not have a way to secure credentials for the client and for this reason, it uses a public client registered in Keycloak.
4. The SPA has direct access to the access token and includes this in requests to the REST API.
5. The REST API is required to include CORS headers in the response. Otherwise, the browser would block the SPA from reading the response.

In summary, the SPA uses the Authorization Code flow directly to obtain tokens from Keycloak, which results in the tokens being available directly in the browser, which has a higher risk of tokens being leaked.

You should now have a good understanding of how to secure different types of web applications, such as traditional server-side web applications, and more modern client-side applications. You have learned that the best practice for securing any web application is redirecting to the Keycloak login pages through the Authorization Code flow, with the PKCE extension. Finally, you also learned that although a SPA can obtain tokens directly from Keycloak, it may not be secure enough for highly sensitive applications.

In the next section, we will look at how to secure mobile applications with Keycloak.

Securing native and mobile applications

Securing a web application with Keycloak is more straightforward than securing a native or mobile application. Keycloak login pages are essentially a web application and it is more natural to redirect a user to a different web application when they are already within the browser.

You may be tempted to implement login pages within the application itself to collect the username and password, then leverage the OAuth 2.0 Resource Owner Password Credential grant to obtain tokens. However, this is simply something that you should not be tempted to do. As mentioned in the previous section, applications should never have direct access to the user credentials, and this approach also means you miss out on a lot of features provided by Keycloak.

To secure a native or mobile application, you should use the Authorization Code flow with the PKCE extension instead. This is more secure, and at the same time gives you the full benefits of using Keycloak.

Effectively, this means that your native or mobile application must use a browser to authenticate with Keycloak. In this regard, there are three options available depending on the type of application:

- Use an embedded web view.
- Use an external user agent (the user's default browser).
- Use an in-app browser tab without the application, which is supported on some platforms, such as Android and iOS.

Using an embedded web view may be tempting as it provides a way to place the login pages within the application. However, this option is not recommended as it is open to vulnerabilities where the credentials may be intercepted. It also does not enable SSO as there are no shared cookies between multiple applications.

Using an in-app browser tab is a decent approach as it enables leveraging the system browser while displaying the login pages with the application. However, it is possible for a malicious application to render a login page within the application that looks like an in-app browser tab, allowing the malicious application to collect the credentials. For users that are concerned about this, they can open the page in the external browser instead.

The following screenshot shows the Keycloak login page in an in-app browser tab on Android:



Figure 6.9 – Keycloak login pages displayed in an in-app browser tab on Android

In all the options, the Keycloak login pages are opened in a browser to authenticate the user. After the user is authenticated, the authorization code is returned to the application, which can then obtain tokens from Keycloak. The following diagram shows the steps involved:

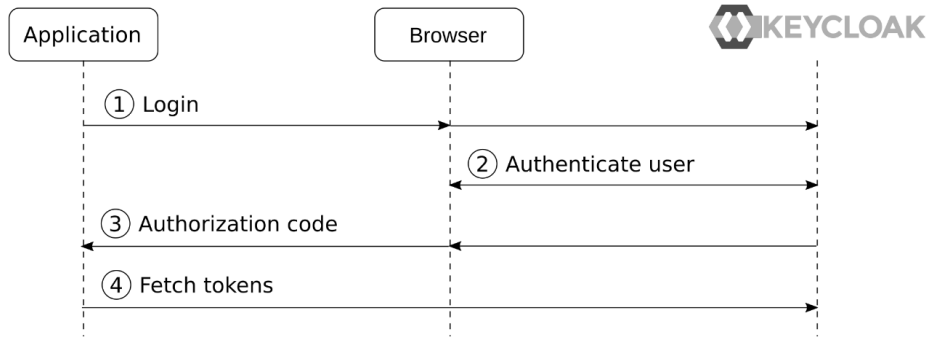


Figure 6.9 – Native application

In more detail, the steps in the diagram are as follows:

1. The application opens the login page in an external browser or using an in-app browser tab.
2. The user authenticates with Keycloak through the external browser.
3. The authorization code is returned to the application.
4. The application exchanges the authorization code for tokens.

To return the authorization code to the application, there are four different approaches using special redirect URIs defined by OAuth 2.0:

- **Claimed HTTPS scheme:** Some platforms allow an application to claim an HTTPS scheme (a URL starting with `https://`), which opens the URI in the application instead of the system browser. For example, the `https://app.acme.org/oauth2callback/provider-name` redirect URI could be claimed by an application called Acme App, resulting in the callback being opened in the Acme App rather than in the browser.
- **Custom URI scheme:** A custom URI scheme is registered with the application. When Keycloak redirects to this custom URI scheme, the request is sent to the application. The custom URI scheme should match the reverse of a domain that is owned by the application developer. For example, the `org.acme.app://oauth2/provider-name` redirect URI matches the domain name `app.acme.org`.
- **Loopback interface:** The application can open a temporary web server on a random port on the loopback interface, then register the `http://127.0.0.1/oauth2/provider-name` redirect URI, which will send the request to the web server started by the application.

- **A special redirect URI:** By using the special `urn:ietf:wg:oauth:2.0:oob` redirect URI, the authorization code is displayed by Keycloak, allowing the user to manually copy and paste the authorization code into the application.

When available, the claimed HTTPS scheme is the recommended approach, as it is more secure. In cases when neither a claimed HTTPS scheme nor a custom scheme can be used, for example, in a CLI, the loopback interface option is a good approach.

To give you a better understanding of how a native application is secured with Keycloak, there is an example application included with this chapter that you can try. The example is showing a CLI that uses the system browser to obtain the authorization code. Before running the example, you need to register a new client with Keycloak with the following settings:

- **Client ID:** `cli`
- **Access Type:** `public`
- **Standard Flow Enabled:** `ON`
- **Valid Redirect URIs:** `http://localhost/callback`

After you have registered the client, you can run the sample in a terminal by running the following commands:

```
$ cd Keycloak-Identity-and-Access-Management-for-Modern-Applications/ch6/  
$ npm install  
$ node app.js
```

When you run the example CLI, it starts a temporary web server on a random port, then it opens the authorization request in the system browser. After you have logged in to Keycloak, it redirects to the web server provided by the application, including the authorization code. The application now has access to the authorization code and can exchange it for an access token.

When running the example CLI, you should see the following output:

```
Listening on port: 40437  
Authorization Code: 32ab30d2...  
Access Token: eyJhbGciOiJSUzI1NiIsInR3GOMibcto...
```

There are also, of course, cases where a browser is not available – for example, running a terminal within a server that does not have a graphical interface. In these cases, the Device Code grant type is a good option.

In summary, the Device Code grant type works by the application showing a short code that a user enters into a special endpoint at the authorization server in a different device with a browser. After entering the code, the user will be asked to log in if they are not already logged in. After completion, the application is able to retrieve the authorization code from the authorization server.

You should now have a good understanding of how to secure native and mobile applications with Keycloak by using the Authorization Code flow through an external browser. In the next section, we will look at how to secure REST APIs with Keycloak.

Securing REST APIs and services

When an application wants to invoke a REST API protected by Keycloak, it first obtains an access token from Keycloak, then includes the access token in the authorization header in requests it sends to the REST API:

```
Authorization: bearer eyJhbGciOiJSUzI1NiIsInR5c...
```

The REST API can then verify the access token to decide whether access should be granted.

This approach makes it easy to provide a REST API that can be leveraged by many applications, even making the REST API available as a public API on the internet for third-party applications to consume.

In *Chapter 5, Authorizing Access with OAuth 2.0*, we covered how the application obtains an access token from Keycloak, then includes the access token in requests it makes to REST APIs so that the REST API can verify whether access should be granted. We also covered various strategies for limiting the access provided by a specific access token, as well as how an access token is verified by the REST API.

With microservices, using tokens to secure the services is especially useful as it enables propagating the authentication context when a service invokes another service, making it easy to provide full end-to-end authentication of the user, as shown in the following diagram:

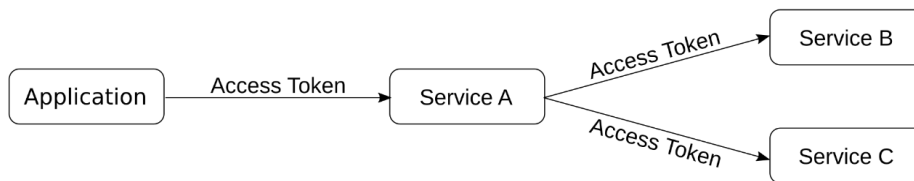



Figure 6.10 – End-to-end user authentication for microservices

In this example, the application includes an access token when it invokes Service A. Service A is then able to invoke both Service B and Service C with the same access token, resulting in all the services using the same authentication context.

Keycloak also has support for service accounts, which allows a service to obtain an access token on behalf of itself by using the Client Credential grant type. Let's give this a go by opening the Keycloak admin console and creating a new client. Use the following values when creating the client:

- **Client ID:** `service`
- **Client Protocol:** `openid-connect`
- **Access Type:** `confidential`
- **Standard Flow Enabled:** `OFF`
- **Implicit Flow Enabled:** `OFF`
- **Direct Access Grants Enabled:** `OFF`
- **Service Accounts Enabled:** `ON`

The following screenshot shows the client you should create:

Service 

[Settings](#) [Credentials](#) [Roles](#) [Client Scopes ?](#) [Mappers ?](#) [Scope ?](#) [Revocation](#) [Sessions ?](#) [Offline Access ?](#)

Client ID ?	<input type="text" value="service"/>
Name ?	<input type="text"/>
Description ?	<input type="text"/>
Enabled ?	<input checked="" type="checkbox"/> ON
Consent Required ?	<input type="checkbox"/> OFF
Login Theme ?	<input type="text" value=""/>
Client Protocol ?	<input type="text" value="openid-connect"/>
Access Type ?	<input type="text" value="confidential"/>
Standard Flow Enabled ?	<input type="checkbox"/> OFF
Implicit Flow Enabled ?	<input type="checkbox"/> OFF
Direct Access Grants Enabled ?	<input type="checkbox"/> OFF
Service Accounts Enabled ?	<input checked="" type="checkbox"/> ON

Figure 6.11 – Service account client in Keycloak

As you have turned off the **Standard Flow Enabled** option for this client, it is not able to obtain tokens using the Authorization Code flow, but as it has **Service Accounts Enabled** turned on, it can use the Client Credential flow instead. The Client Credential flow allows a client to obtain tokens on behalf of itself by using the credentials for the client.

To obtain an access token, the client makes a POST request to the Keycloak token endpoint with the following parameters:

- `client_id`
- `client_secret`
- `grant_type=client_credentials`

Let's try to use `curl` to get an access token. First, you need to go to the **Credentials** tab for the client you just created and copy the secret for the client. Then, you can open a terminal and run the following command:

```
$ export SECRET=<insert secret from Keycloak Admin Console>
$ curl --data "client_id=service&client_secret=$SECRET&grant_type=client_credentials" http://localhost:8080/auth/realms/myrealm/protocol/openid-connect/token
```

Keycloak will return an access token response, which is a JSON document that, among other things, includes the access token:

```
{
  "access_token": "eyJhbGciOiJSUzI1NiIsI...",
  "expires_in": 299,
  "token_type": "bearer",
  "scope": "profile email",
  ...
}
```

It is not only REST APIs that can leverage tokens. **Simple Authentication and Security Layer (SASL)**, which is a popular protocol for authentication for a range of protocols, also include support for bearer tokens. gRPC and WebSockets can also leverage bearer tokens for secure invocation.

In this section, you have learned how by including a bearer token in the request to a service, the service is able to verify whether the request should be accepted by verifying the token either directly or through the token introspection endpoint.

Summary

In this chapter, you learned the difference between an internal and an external application, where external applications require asking the user for consent to grant access, while internal applications do not. You then learned how different web application architectures are secured with Keycloak, and why it is more secure to have a backend for a SPA that obtains tokens from Keycloak, instead of directly obtaining tokens in the SPA itself. You then learned how Keycloak can be used to secure other types of applications, such as native and mobile applications. Finally, you learned that bearer tokens can be used to secure a range of different services, including REST APIs, microservices, gRPC, WebSockets, and a range of other protocols.

You should now have a good understanding of the principles and best practices for securing your application with Keycloak. In the next chapter, we will look at what options are available to integrate all your applications with Keycloak.

Questions

1. What is the best way to secure the invocations from a SPA to a REST API?
2. Can OAuth 2.0 and bearer tokens only be used to secure web applications and REST APIs?
3. How should you secure a native or mobile application with Keycloak?

Further reading

For more information on the topics covered in this chapter, refer to the following links:

- OAuth 2.0 for browser-based apps: <https://oauth.net/2/browser-based-apps/>
- OAuth 2.0 for mobile and native apps: <https://oauth.net/2/native-apps/>
- AppAuth: <https://appauth.io/>