

Keycloak - Identity and Access Management for Modern Applications

Discover the power of Keycloak, OpenID Connect, and OAuth 2.0 for modern applications

Second Edition

Mike Shaver and
Paulo Freitas

(packt)

EXPERT INSIGHT

Keycloak - Identity and Access Management for Modern Applications

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

Second Edition



Stian Thorgersen
Pedro Igor Silva

packt

Keycloak - Identity and Access Management for Modern Applications

Second Edition

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

Stian Thorgersen

Pedro Igor Silva



BIRMINGHAM—MUMBAI

Keycloak - Identity and Access Management for Modern Applications

Second Edition

Copyright © 2023 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.

Senior Publishing Product Manager: Aaron Tanna

Acquisition Editor – Peer Reviews: Gaurav Gavas

Project Editor: Meenakshi Vijay

Content Development Editor: Liam Thomas Draper

Assistant Development Editor: Elliot Dallow

Copy Editor: Safis Editing

Technical Editor: Kushal Sharma

Proofreader: Safis Editing

Indexer: Manju Arasan

Presentation Designer: Rajesh Srisath

Developer Relations Marketing Executive: Meghal Patel

First published: May 2021

Second edition: July 2023

Production reference: 2140723

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-80461-644-4

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 started his career back in 2000 at an ISP, where he had his first experiences with open source projects such as FreeBSD and Linux. In this time he worked 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.

About the reviewers

Martin Besozzi is an experienced **Identity and Access Management (IAM)** architect with over 16 years of industry expertise. He specializes in designing and implementing robust IAM solutions using a variety of commercial and open-source IAM frameworks, aiming to achieve both security and business objectives based on best practices and industry standards. His job roles have spanned enterprise architect, team lead, and software development in the IAM space. Martin has published articles and conducted workshops on modern IAM topics, sharing his knowledge and insights with the community.

Thomas Darimont is a recognized Keycloak expert who contributed many significant features, bug fixes, documentation, and examples over many years to the project, for which he eventually got promoted as the first Keycloak maintainer outside of Red Hat. He currently works as principal consultant at codecentric AG, where he assists clients in enhancing their customer journey with cutting-edge digital identity solutions.

Thanks to Stian and Pedro for letting me participate by reviewing this book.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/SecNet>



Contents

Preface

Who this book is for

What this book covers

To get the most out of this book

1. Getting Started with Keycloak

Technical requirements

Introducing Keycloak

Installing and running Keycloak

Running Keycloak on Docker

Installing and running Keycloak with OpenJDK

Installing OpenJDK

Installing Keycloak

Starting Keycloak

Discovering the Keycloak admin and account consoles

Getting started with the Keycloak admin console

Creating and configuring a realm

Creating a user

Creating a group

Creating a global role

Getting started with the Keycloak account console

Summary

Questions

2. Securing Your First Application

Technical requirements

Understanding the sample application

Running the application

Understanding how to log in to the application

Securely invoking the backend REST API

Summary

Questions

3. Brief Introduction to Standards

Authorizing application access with OAuth 2.0

Authenticating users with OpenID Connect

Leveraging JWT for tokens

[Understanding why SAML 2.0 is still relevant](#)

[Summary](#)

[Questions](#)

[4. Authenticating Users with OpenID Connect](#)

[Technical requirements](#)

[Running the OpenID Connect playground](#)

[Understanding the Discovery endpoint](#)

[Authenticating a user](#)

[Understanding the ID token](#)

[Updating the user profile](#)

[Adding a custom property](#)

[Adding roles to the ID token](#)

[Invoking the UserInfo endpoint](#)

[Dealing with users logging out](#)

[Initiating the logout](#)

[Leveraging ID and access token expiration](#)

[Leveraging OIDC Session Management](#)

[Leveraging OIDC Back-Channel Logout](#)

[A note on OIDC Front-Channel Logout](#)

[How should you deal with logout?](#)

[Summary](#)

[Questions](#)

[Further reading](#)

[5. Authorizing Access with OAuth 2.0](#)

[Technical requirements](#)

[Running the OAuth 2.0 playground](#)

[Obtaining an access token](#)

[Requiring user consent](#)

[Limiting the access granted to access tokens](#)

[Using the audience to limit token access](#)

[Using roles to limit token access](#)

[Using the scope to limit token access](#)

[Validating access tokens](#)

[Summary](#)

[Questions](#)

[Further reading](#)

[6. Securing Different Application Types](#)

[Technical requirements](#)
[Understanding internal and external applications](#)
[Securing web applications](#)
 [Securing server-side web applications](#)
 [Securing a SPA with a dedicated REST API](#)
 [Securing a SPA with an intermediary REST API](#)
 [Securing a SPA with an external REST API](#)
[Securing native and mobile applications](#)
 [Authenticating with an input-constrained device](#)
[Securing REST APIs and services](#)
[Summary](#)
[Questions](#)
[Further reading](#)

7. [Integrating Applications with Keycloak](#)

[Technical requirements](#)
[Choosing an integration architecture](#)
[Choosing an integration option](#)
[Integrating with Golang applications](#)
[Integrating with Java applications](#)
 [Using Quarkus](#)
 [Creating a Quarkus client](#)
 [Creating a Quarkus resource server](#)
 [Using Spring Boot](#)
 [Creating a Spring Boot client](#)
 [Creating a Spring Boot resource server](#)
[Integrating with JavaScript applications](#)
[Integrating with Node.js applications](#)
 [Creating a Node.js client](#)
 [Creating a Node.js resource server](#)
[Using a reverse proxy](#)
[Try not to implement your own integration](#)

[Summary](#)

[Questions](#)

[Further reading](#)

8. [Authorization Strategies](#)

[Understanding authorization](#)
[Using RBAC](#)

Using GBAC

Mapping group membership into tokens

Using OAuth2 scopes

Using ABAC

Using Keycloak as a centralized authorization server

Summary

Questions

Further reading

9. Configuring Keycloak for Production

Technical requirements

Setting the hostname for Keycloak

Setting the frontend URL

Setting the backend URL

Setting the admin URL

Enabling TLS

Configuring a database

Enabling clustering

Configuring a reverse proxy

Distributing the load across nodes

Forwarding client information

Keeping session affinity

Testing your environment

Testing load balancing and failover

Testing the frontend and backchannel URLs

Summary

Questions

Further reading

10. Managing Users

Technical requirements

Managing local users

Creating a local user

Managing user credentials

Obtaining and validating user information

Enabling self-registration

Managing user attributes

Integrating with LDAP and Active Directory

Understanding LDAP mappers

[Synchronizing groups](#)

[Synchronizing roles](#)

[Integrating with third-party identity providers](#)

[Creating an OpenID Connect identity provider](#)

[Integrating with social identity providers](#)

[Allowing users to manage their data](#)

[Summary](#)

[Questions](#)

[Further reading](#)

11. [Authenticating Users](#)

[Technical requirements](#)

[Understanding authentication flows](#)

[Configuring an authentication flow](#)

[Using passwords](#)

[Changing password policies](#)

[Resetting user passwords](#)

[Using OTPs](#)

[Changing OTP policies](#)

[Allowing users to choose whether they want to use OTPs](#)

[Forcing users to authenticate using OTPs](#)

[Using Web Authentication \(WebAuthn\)](#)

[Enabling WebAuthn for an authentication flow](#)

[Registering a security device and authenticating](#)

[Using strong authentication](#)

[Summary](#)

[Questions](#)

[Further reading](#)

12. [Managing Tokens and Sessions](#)

[Technical requirements](#)

[Managing sessions](#)

[Managing session lifetimes](#)

[Managing active sessions](#)

[Expiring user sessions prematurely](#)

[Understanding cookies and their relation to sessions](#)

[Managing tokens](#)

[Managing ID tokens' and access tokens' lifetimes](#)

[Managing refresh tokens' lifetimes](#)

[Enabling refreshing token rotation](#)
[Revoking tokens](#)

[Summary](#)

[Questions](#)

[Further reading](#)

13. [Extending Keycloak](#)

[Technical requirements](#)

[Understanding service provider interfaces](#)

[Packaging and deploying a custom provider](#)

[Understanding the KeycloakSessionFactory and KeycloakSession components](#)

[Understanding the life cycle of a provider](#)

[Changing the look and feel](#)

[Understanding themes](#)

[Creating and deploying a new theme](#)

[Extending templates](#)

[Extending theme-related SPIs](#)

[Customizing authentication flows](#)

[Looking at other customization points](#)

[Summary](#)

[Questions](#)

[Further reading](#)

14. [Securing Keycloak and Applications](#)

[Securing Keycloak](#)

[Encrypting communication to Keycloak](#)

[Configuring the Keycloak hostname](#)

[Rotating the signing keys used by Keycloak](#)

[Regularly updating Keycloak](#)

[Loading secrets into Keycloak from an external vault](#)

[Protecting Keycloak with a firewall and an intrusion prevention system](#)

[Securing the database](#)

[Protecting the database with a firewall](#)

[Enabling authentication and access control for the database](#)

[Encrypting the database](#)

[Securing cluster communication](#)

[Enabling cluster authentication](#)

[Encrypting cluster communication](#)

[Securing user accounts](#)

[Securing applications](#)

[Web application security](#)

[OAuth 2.0 and OpenID Connect best practice](#)

[Keycloak client configurations](#)

[Summary](#)

[Questions](#)

[Further reading](#)

[Assessments](#)

[Chapter 1](#)

[Chapter 2](#)

[Chapter 3](#)

[Chapter 4](#)

[Chapter 5](#)

[Chapter 6](#)

[Chapter 7](#)

[Chapter 8](#)

[Chapter 9](#)

[Chapter 10](#)

[Chapter 11](#)

[Chapter 12](#)

[Chapter 13](#)

[Chapter 14](#)

[Other Books You May Enjoy](#)

[Index](#)

Preface

Keycloak is an open source **Identity and Access Management (IAM)** tool with a focus on modern applications such as single-page applications, mobile applications, and REST APIs. Since the first edition of this book was published there have been some big changes to Keycloak.

The Keycloak administration console has received a full make-over with a bigger focus on usability and accessibility requirements.

This distribution of Keycloak is now based on Quarkus rather than the WildFly application server. This brings a new, and much improved, way to configure and deploy Keycloak to different computing environments – from on-premises infrastructure to public and hybrid clouds.

Some of the Keycloak Adapters have been deprecated, and instead Keycloak is now focusing on selecting quality libraries from existing communities; like leveraging built-in support for OpenID Connect and OAuth 2.0 from whatever language or framework your application is using.

The project was started in 2014 with a strong focus on making it easier for developers to secure their applications. It has since grown into a well-established open source project with a strong community and user base. It is used in production for scenarios

ranging from small websites with only a handful of users, up to large enterprises with millions of users.

This book introduces you to Keycloak, covering how to install Keycloak as well as how to configure it ready for production use cases. Furthermore, this book covers how to secure your own applications, as well as providing a good foundation for understanding OAuth 2.0 and OpenID Connect.

In this edition, there are updated chapters based on the latest release of Keycloak. If you are familiar with the content from the previous edition, this edition will give you relevant updates throughout to bring you up to speed with the latest release. For the newcomers, this edition will serve as an excellent first step towards understanding Keycloak and how it can help you to enable a rich IAM solution within your organization.

Who this book is for

This book is for developers, system administrators, and security engineers, or anyone who wants to leverage Keycloak and its capabilities to secure applications.

If you are new to Keycloak, this book will provide you with a strong foundation to leverage Keycloak in your projects.

If you have been using Keycloak for a while, but have not mastered everything yet, you should still find a lot of useful information in this book.

What this book covers

Chapter 1, Getting Started with Keycloak, gives you a brief introduction to Keycloak and steps on how to get quickly up to speed by installing and running Keycloak yourself. It also provides an introduction to the Keycloak admin and account consoles.

Chapter 2, Securing Your First Application, explains how to secure your first application with Keycloak through a sample application consisting of a single-page application and a REST API.

Chapter 3, Brief Introduction to Standards, provides a brief introduction and comparison of the standards Keycloak supports to enable you to integrate your applications securely and easily with Keycloak.

Chapter 4, Authenticating Users with OpenID Connect, teaches how to authenticate users by leveraging the OpenID Connect standard. This chapter leverages a sample application that allows you to see and understand how an application authenticates to Keycloak through Open ID Connect.

Chapter 5, Authorizing Access with OAuth 2.0, teaches how to authorize access to REST APIs and other services by leveraging the OAuth 2.0 standard. Through a sample application, you will see firsthand how an application obtains an access token through OAuth 2.0, which the application uses to invoke a protected REST API.

Chapter 6, Securing Different Application Types, covers best practices on how to secure different types of applications, including web,

mobile, and native applications, as well as REST APIs and other backend services.

Chapter 7, Integrating Applications with Keycloak, provides steps on how to integrate your applications with Keycloak, covering a range of different programming languages, including Go, Java, client-side JavaScript, Node.js, and Python. It also covers how you can utilize a reverse proxy to secure an application implemented in any programming language or framework.

Chapter 8, Authorization Strategies, covers how your application can use information about the user from Keycloak for access management, covering roles and groups, as well as custom information about users.

Chapter 9, Configuring Keycloak for Production, teaches how to configure Keycloak for production, including how to enable TLS, configuring a relational database, and enabling clustering for additional scale and availability.

Chapter 10, Managing Users, takes a closer look at the capabilities provided by Keycloak related to user management. It also explains how to federate users from external sources such as LDAP, social networks, and external identity providers.

Chapter 11, Authenticating Users, covers the various authentication capabilities provided by Keycloak, including how to enable second-factor authentication, as well as security keys.

Chapter 12, Managing Tokens and Sessions, helps understand how Keycloak leverages server-side sessions to keep track of

authenticated users, as well as best practices for managing tokens issued to your applications.

Chapter 13, Extending Keycloak, explains how you can extend Keycloak, covering how you can modify the look and feel of user-facing pages such as the login pages and account console. It also provides a brief introduction to one of the more powerful capabilities of Keycloak that allows you to provide custom extensions for a large number of extension points.

Chapter 14, Securing Keycloak and Applications, provides best practices on how to secure Keycloak for production. It also provides a brief introduction to some best practices to follow when securing your own applications.

Assessments, check your answers to the questions at the end of each chapter here.

To get the most out of this book

To be able to run the examples provided in this book, you need to have OpenJDK and Node.js installed on your computer. All code examples have been tested using OpenJDK 17 and Node.js 18 on Linux (Fedora). However, the examples should also work on newer versions of OpenJDK and Node.js, as well as with Windows and mac OS.

	OS requirements
--	------------------------

Software/hardware covered in the book	
Keycloak 22	Linux (any), macOS, Windows
OpenJDK 17+	Linux (any), macOS, Windows
Node.js 18+	Linux (any), macOS, Windows

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository

(link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

The code bundle for the book is hosted on GitHub at <https://github.com/PacktPublishing/Keycloak---Identity-and-Access-Management-for-Modern-Applications-2nd-Edition/>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: <https://packt.link/6BLPp>.

Code in Action

Code in Action videos for this book can be viewed at <https://packt.link/ZZQat>.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: “Keycloak supports

the `authorization_code` grant type and the `code` and `token` response types.”

A block of code is set as follows:

```
<Header>.<Payload>.<Signature>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

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

Any command-line input or output is written as follows:

```
$ npm install
$ npm start
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: “Now click on **Load OpenID Provider Configuration.**”

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit <http://www.packtpub.com/submit-errata>, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packtpub.com>.

Share your thoughts

Once you've read *Keycloak - Identity and Access Management for Modern Applications, Second Edition*, we'd love to hear your thoughts! Please [click here to go straight to the Amazon review page](#) for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781804616444>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

1

Getting Started with Keycloak

If you are new to **Keycloak**, this chapter will quickly get you up to speed. We'll start with a brief introduction to Keycloak. Then, you will find out how easy it is to install Keycloak and get it up and running. After we have started Keycloak, you will learn about the **Keycloak admin console**, which provides a great interface for managing and configuring Keycloak. Finally, we'll take a quick look at the Keycloak account console as well, which lets users of your applications manage their own accounts.

By the end of this chapter, you will know how to get started with the Keycloak server, and understand how you can use the Keycloak admin console to manage Keycloak. You will learn how to prepare Keycloak with an example user in order to get started with securing your first application in the next chapter.

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

- Introducing Keycloak
- Installing and running Keycloak
- Discovering the Keycloak admin and account consoles

Technical requirements

For this chapter, in order to run Keycloak, you will need to have Docker (<https://www.docker.com/>) or JDK 17+ (<https://openjdk.java.net/>) installed on your workstation.

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

<https://packt.link/ZuIUs>

Introducing Keycloak

Keycloak is an open source **Identity and Access Management** tool with a focus on modern applications such as single-page applications, mobile applications, and REST APIs.

The project was started in 2014 with a strong focus on making it easier for developers to secure their applications. It has since grown into a well-established open source project with a strong community and user base. It is used in production for scenarios ranging from small websites with only a handful of users up to large enterprises with millions of users.

Keycloak provides fully customizable login pages, including support for strong authentication, and built-in capabilities such as the recovery of passwords, requiring users to regularly update their passwords, accepting terms and conditions, and a lot more. All of this without any need to add anything to your applications, or any coding at all. All pages visible to your users support custom themes, making it very easy to modify the look and feel of the pages to integrate with your corporate branding and existing applications.

By delegating authentication to Keycloak, your applications do not need to worry about different authentication mechanisms, or how

to safely store passwords. This approach also provides a higher level of security as applications do not have direct access to user credentials; they are instead provided with security tokens that give them only access to what they need.

When it comes to authentication, Keycloak supports a wide range of authentication factors, allowing you to easily enable **Multi-Factor Authentication (MFA)** and **Strong Authentication (SA)** for your applications. As you will see in *Chapter 11, Authenticating Users*, with only a few steps, you are able to choose from authenticating your users using **OTPs**, security devices and WebAuthn, passwords, or any combination of these. Keycloak provides single sign-on as well as session management capabilities, allowing users to access multiple applications, while only having to authenticate once. Both users themselves and administrators have full visibility in to where users are authenticated, and can remotely terminate sessions when required.

Keycloak builds on industry-standard protocols supporting OAuth 2.0, OpenID Connect, and SAML 2.0. Using industry-standard protocols is important from both a security perspective and in terms of making it easier to integrate with existing and new applications.

Keycloak comes with its own user database, which makes it very easy to get started. You can also easily integrate with existing identity infrastructure. Through its identity brokering capabilities, you can plug in existing user bases from social networks, or other enterprise identity providers. It can also integrate with existing user directories, such as **Active Directory** and **LDAP servers**.

Keycloak is a lightweight and easy-to-install solution. It is highly scalable and provides high availability through clustering capabilities.

A lot of effort has gone into making Keycloak usable out of the box, supporting common use cases, but, at the same time, it is highly customizable and extendable when needed. Keycloak has a large number of extension points where you can implement and deploy custom code to Keycloak to modify existing behavior or add completely new capabilities. Examples of extensions that can be written to Keycloak include custom authentication mechanisms, integrations with custom user stores, and the custom manipulation of tokens. You can even implement your own custom login protocols.

This section was a very brief introduction to the features and capabilities of Keycloak. As this book aims to give you a practical guide to Keycloak, we will come back to many of these features in later chapters, where you will learn firsthand how you can put these to use.

Installing and running Keycloak

In this section, you will quickly learn how to install and run Keycloak. Once you have Keycloak up and running, we will take a look at the Keycloak admin console and the Keycloak account console.

Keycloak provides a few options on how it can be installed, including the following:

- Running as a container on Docker
- Installing and running Keycloak locally (which will require a Java virtual machine, such as OpenJDK)
- Running Keycloak on Kubernetes
- Using the Keycloak Kubernetes Operator

If you already have Docker installed on your workstation, this is the recommended approach as it is simpler to get up and running this way.

If you don't have Docker installed, it is easier to get started by installing and running it locally. The only dependency required is a Java virtual machine.

Keycloak can also be easily deployed to Kubernetes, where you have the option of using the Keycloak Kubernetes Operator, which makes installation, configuration, and management even simpler. We are not going to provide instructions for Kubernetes in this book, as we instead want to focus on Keycloak and its features.

If you are interested in knowing how to run Keycloak on Kubernetes, then the Keycloak website provides great *Getting started* guides at <https://www.keycloak.org/guides#getting-started>

In the next section, we will look at how you can run Keycloak as a container on Docker. If you prefer to run it locally, you can skip to the section titled *Installing and running Keycloak with OpenJDK*.

Running Keycloak on Docker

With Docker, it is very easy to run Keycloak as you don't need to install a Java virtual machine yourself, nor do you have to download and extract the Keycloak distribution.

To run Keycloak on Docker, simply execute the following command:

```
$ docker run -e KEYCLOAK_ADMIN=admin -e KEYCLOAK_ADM
```

As Keycloak does not ship with a default admin account, passing the environment variables, `KEYCLOAK_ADMIN` and `KEYCLOAK_ADMIN_PASSWORD`, makes it easy to create an initial admin account. We are also using `-p 8080:8080` to publish the port used by Keycloak to the host, so as to make it easy to access Keycloak.

After a few seconds, you can verify that Keycloak is running by opening `http://localhost:8080/admin` and log in with the username `admin` and password `admin`.

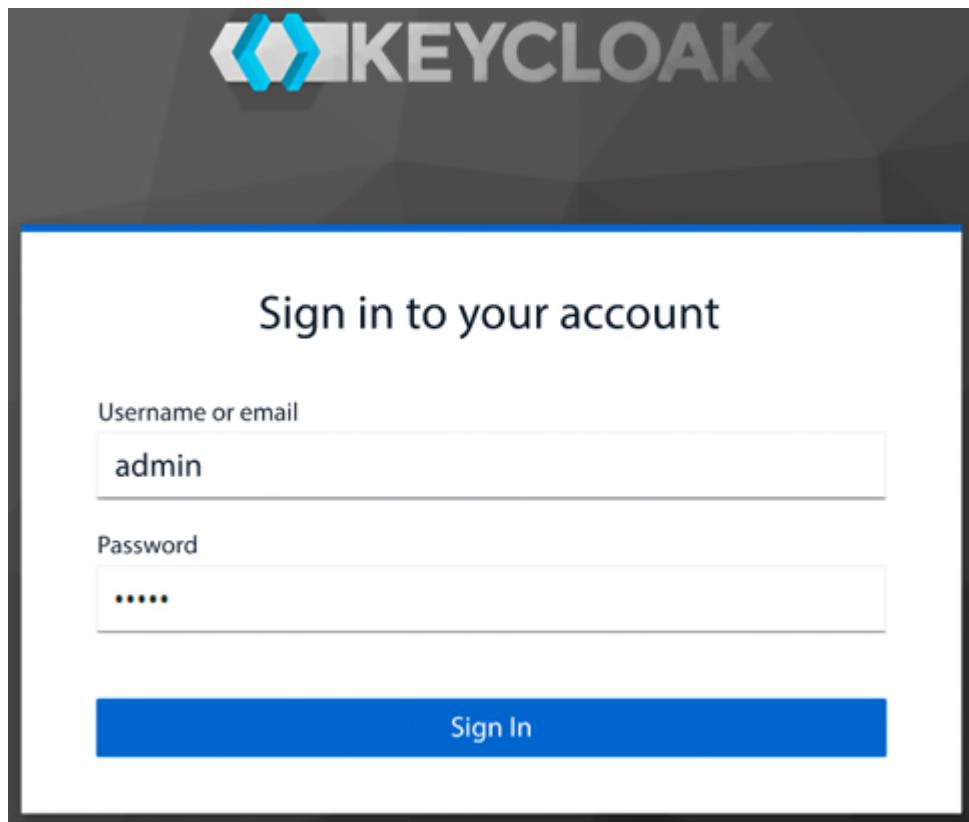


Figure 1.1: Log in to the administration console as the admin user

Congratulations! You now have Keycloak running as a Docker container and can get started with trying Keycloak out by first discovering the Keycloak admin and account consoles.

Installing and running Keycloak with OpenJDK

As Keycloak is implemented in Java, it is easy to run Keycloak on any operating system without the need to install additional dependencies. The only thing that you need to have installed is a Java virtual machine, such as OpenJDK.

In the next section, we will install OpenJDK, which is required before running Keycloak. If you already have a Java virtual machine installed, you can skip the next section and go directly to the section entitled *Installing Keycloak*.

Installing OpenJDK

To install Keycloak, you need to install version 17 of the **Java Runtime Environment (JRE)**. To install OpenJDK, download one of the ready builds at adoptium.net/temurin/archive. Open this page in your browser and then **download OpenJDK version 17**. Download the build for your operating system, and then extract it to a suitable location. Once extracted, set the `JAVA_HOME` environment variable to point to the extracted directory.

The following commands show an example of installing OpenJDK in Linux: shows an example of installing a ready build of OpenJDK on Linux:

```
$ mkdir ~/kc-book
$ cd ~/kc-book
$ tar xf vz ~/Downloads/OpenJDK17U-jdk_x64_linux_hot-
$ export JAVA_HOME=~/kc-book/jdk-17.0.6+10
$ $JAVA_HOME/bin/java -version
```

The last command (`java -version`) verifies that Java is working properly.

Now that you have OpenJDK installed, we will move on to installing Keycloak.

Installing Keycloak

Once you have the Java virtual machine installed on your workstation, the next step is to download the distribution of Keycloak from the Keycloak website. Open <https://www.keycloak.org/downloads>, and then download either the ZIP or the TAR.GZ archive of the server (standalone server distribution). Once downloaded, simply extract this archive to a suitable location.

The following screenshot shows an example of installing Keycloak on Linux:

```
$ unzip ~/Downloads/keycloak-22.0.0.zip  
$ export KC_HOME=~/kc-book/keycloak-22.0.0
```

You are now ready to start Keycloak, which we will cover next.

Starting Keycloak

Once you have installed Keycloak and created the initial admin account, it's easy to start Keycloak.

On Linux or macOS, start Keycloak with the following command:

```
$ export KEYCLOAK_ADMIN=admin  
$ export KEYCLOAK_ADMIN_PASSWORD=admin  
$ cd $KC_HOME  
$ bin/kc.sh start-dev
```

Or, on Windows, execute the following command:

```
> set KEYCLOAK_ADMIN=admin  
> set KEYCLOAK_ADMIN_PASSWORD=admin  
> cd %KC_HOME%  
> bin\kc.bat start-dev
```

As Keycloak does not ship with a default admin account, passing the environment variables, `KEYCLOAK_ADMIN` and `KEYCLOAK_ADMIN_PASSWORD`, makes it easy to create an initial admin account.

After a few seconds, you can verify that Keycloak is running by opening `http://localhost:8080/admin` and log in with the username `admin` and password `admin`.

Congratulations! You now have Keycloak running on your workstation and can get started with trying Keycloak out by first discovering the Keycloak admin and account consoles.

Discovering the Keycloak admin and account consoles

In this section, we will take a look at the Keycloak admin and account consoles. The admin console is an extensive console that allows you to configure and manage Keycloak. The account console, on the other hand, is there to allow your end users to manage their accounts.

Getting started with the Keycloak admin console

In this section, you will learn how to log in to the Keycloak admin console as well as learn how to set up the basic configuration needed to secure your first application.

The Keycloak admin console provides an extensive and friendly interface for administrators and developers to configure and manage Keycloak.

To access the admin console, open

`http://localhost:8080/admin` in a browser. You will be redirected to the Keycloak login page, where you can log in with the admin username and password you created in the previous section while installing Keycloak.

Once you have logged in, you will see the configuration for the master realm in Keycloak, as shown in the following screenshot:

The screenshot shows the Keycloak admin console interface. At the top, there is a dark header bar with the Keycloak logo, a search icon, the user name 'admin', and a profile icon. Below the header, the title 'master realm' is displayed. Underneath the title, there are two tabs: 'Server info' (which is selected) and 'Provider info'. The main content area is divided into two columns. The left column, titled 'Server Info', contains information about the version (22.0.0), product (Default), and memory usage (Total memory: 455 MB, Free memory: 400 MB, Used memory: 55 MB). The right column, titled 'Profile', lists various features: Enabled features (ACCOUNT3, ADMIN_FINE_GRAINED_AUTHZ, CLIENT_SECRET_ROTATION, DECLARATIVE_USER_PROFILE, DOCKER, DYNAMIC_SCOPES, FIPS, MAP_STORAGE, RECOVERY_CODES, SCRIPTS, TOKEN_EXCHANGE, UPDATE_EMAIL); Disabled features (none listed); and Experimental features (DYNAMIC_SCOPES, MAP_STORAGE).

Figure 1.2: The Keycloak admin console

You will learn a lot more about the admin console throughout the book, but let's go through a few steps to make your Keycloak application ready to start securing applications.

Creating and configuring a realm

The first thing you will want to do is create a realm for your applications and users. Think of a realm as a tenant. A realm is fully isolated from other realms; it has its own configuration, and its own set of applications and users. This allows a single installation of Keycloak to be used for multiple purposes. For example, you may want to have one realm for internal applications and employees, and another realm for external applications and customers.

To create a new realm, click on the menu icon in the top-left corner (on the left side of the Keycloak logo) to expand the menu. Now, click on the realm selector to see a list of realms, including a button to create a new realm. Click on the **Create Realm** button:

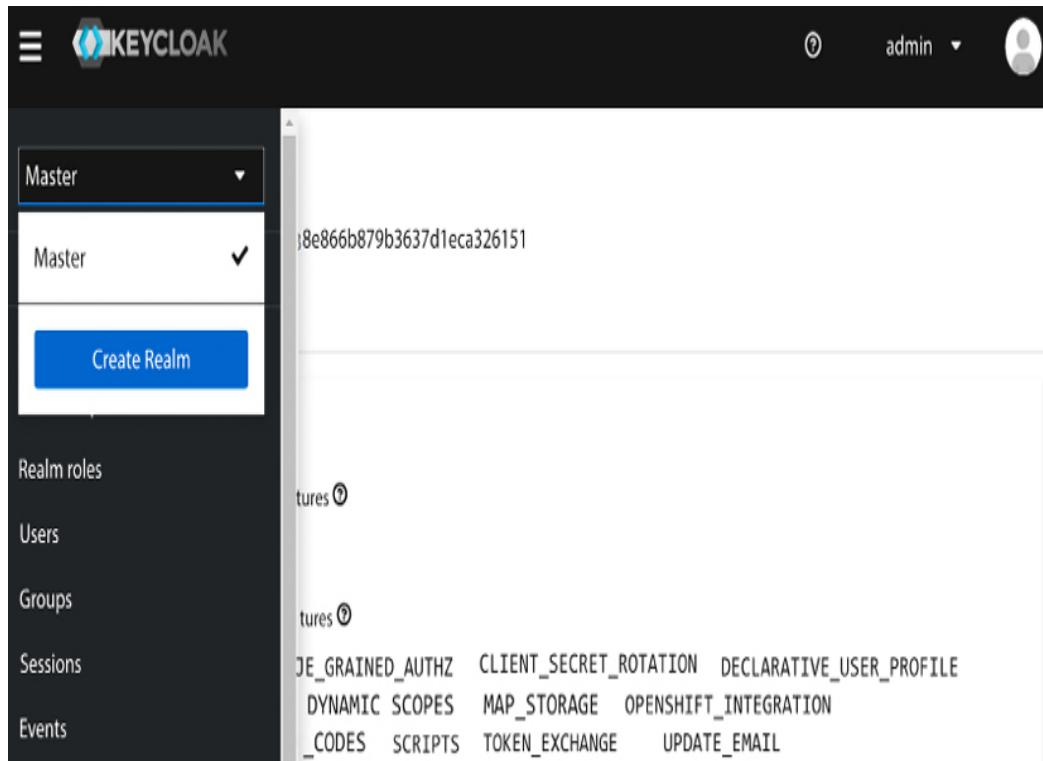


Figure 1.3: Realm selector

On the next page, enter a name for the realm. As the name is used in URLs, the name should ideally not use special characters that need escaping in URLs (such as spaces). Once created, you can set a human-friendly display name. For example, use `myrealm` for the name, and `My Realm` for the display name.

Creating a user

Once you have created the realm, let's create the first user in the realm:

1. From the left-hand menu, click on **Users**, and then click on **Create new user** button.
2. Enter a memorable username, and also enter a value of your choice for email, first name, and last name.
3. The **Email Verified** option can be selected by an administrator if they know this is the valid email address for the user.
4. **Required User Actions** allows an administrator to require a user to perform some initial actions on the next login; for example, to require the user to review their profile, or to verify their email address.
5. Remember to click on **Create** after you have completed the form:

Create user

Required user actions	<input type="button" value="Select action"/>
Username *	keycloak
Email	keycloak@keycloak.org
Email verified	<input checked="" type="radio"/> No
First name	Ola
Last name	Nordmann
Groups	<input type="button" value="Join Groups"/>
<input type="button" value="Create"/> <input type="button" value="Cancel"/>	

Figure 1.4: The Add user form

A user has a few standard built-in attributes, such as **First name**, but it is also possible to add any custom attributes through the **Attributes** tab.

Before the user can log in, you have to create an initial temporary password. To do this, click on the **Credentials** tab. In this tab, click on the **Set Password** button and follow the instructions to set a new password to the user.

If the **Temporary** option is enabled, the user will be required to change their password when logging in for the first time. In cases where an administrator creates the user, this makes a lot of sense.

Creating a group

Next, let's create a group and add the user we previously created to the group. From the menu on the left-hand side, click on **Groups**, and then click on the **Create group** button.

Enter a name for the group, for example, **mygroup**, and then click on **Create**.

Once you have created the group, you can see the **mygroup** group in the group list. By clicking on it, you can edit the group settings.

For instance, you can add attributes to the group. A user inherits all the attributes from a group it belongs to. This can be useful if, for example, you have a group for all employees in an office and want to add the office address to all employees in this group.

You can also grant roles to a group, which again are inherited by all members of the group.

To add the user to the group, go back to the **Users** page and select the user you created previously.

Next, click on the **Groups** tab. In this tab, click **Join Group**, select the group you created previously, and click on **Join** to add the user

to the group.

Creating a global role

To create a global role, click on **Realm roles** in the menu on the left-hand side, and then click on **Create role**. Enter a role name, for example, `myrole`. You can also add a description to the role, which can be especially useful if there are other administrators.

Any role in Keycloak can be turned into a composite role, allowing other roles to be added to the role. A user who is granted a composite role will dynamically be granted all roles within the composite role. Composite roles can even contain other composite roles. This feature can be very powerful, but, at the same time, should be used with some care. Composite roles can be a bit difficult to manage, and can also have a performance overhead if overused, especially if there are many layers of composite roles.

To add the user to the role, go back to the **Users** page and select the user you created previously.

Next, click on the **Role mapping** tab. In this tab, click **Assign role**, select the role you created previously, and click on **Assign** to add the user to the role.

You have now created all the required initial configuration to get started securing your first application, but first let's take a look at the Keycloak account console, which lets users manage their own accounts.

Getting started with the Keycloak account console

The Keycloak account console provides an interface where users can manage their own accounts, including the following:

- Updating their user profile
- Updating their password
- Enabling second-factor authentication
- Viewing applications, including what applications they have authenticated to
- Viewing open *sessions*, including remotely signing out of other sessions

To access the account console, open

`http://localhost:8080/realm/myrealm/account` in a browser

(if you used a different realm name in the previous section, replace `myrealm` with the name of your realm). You will be redirected to the Keycloak login pages, where you can log in with the username and password you created in the previous section while creating your first user:

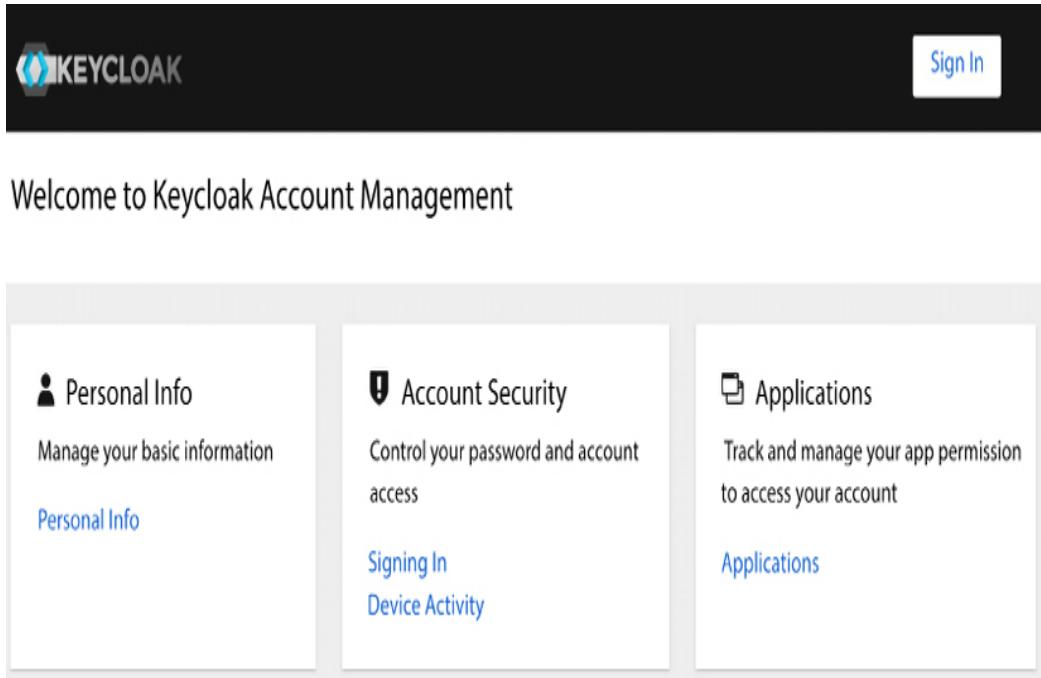


Figure 1.5: The Keycloak account console

You can also find the URL of the account console through the Keycloak admin console. In the admin console, click on **Clients**, and then you will find the URL of the account console next to the account client.

You have now learned how Keycloak not only provides an extensive admin console, but also a self-management console for users of your applications to manage their own accounts.

Summary

In this chapter, you learned how to install Keycloak and get it up and running. You also learned how to use the Keycloak admin console to create your first realm, including an example user with

an associated role. This provides you with the foundation on which to continue building throughout the book.

In the next chapter, we will use what you have learned in this chapter in order to secure your first application with Keycloak.

Questions

1. Can you run Keycloak on Docker and Kubernetes?
2. What is the Keycloak admin console?
3. What is the Keycloak account console?

Use the questions above to test your learning. When you're ready to check your answers go to the *Assessments* chapter to find out how you did. You'll find the answers to these, and all the other questions throughout the book there.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/SecNet>



2

Securing Your First Application

In this chapter, you will learn how to secure your first application with **Keycloak**. To make things a bit more interesting, the sample application you will be running consists of two parts, a **frontend web application** and a **backend REST API**. This will show you how a user can authenticate to the frontend, and also how it is able to securely invoke the backend.

By the end of this chapter, you will have a basic understanding of how applications can be secured by Keycloak by leveraging **OpenID Connect**.

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

- Understanding the sample application
- Running the application
- Understanding how to log in to the application
- Securely invoking the backend REST API

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-2nd-Edition/archive/main.zip
```

Alternatively, you can download a ZIP of the repository from <https://github.com/PacktPublishing/Keycloak---Identity-and-Access-Management-for-Modern-Applications-2nd-Edition/archive/main.zip>.

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

<https://packt.link/oxOr8>

Understanding the sample application

The sample application consists of two parts – a frontend web application and a backend REST API.

The frontend web application is a **single-page application** written in **JavaScript**. As we want to focus on what Keycloak can offer, the application is very simple. Furthermore, to make it as simple as possible to run the application, it uses *Node.js*. The application provides the following features:

- Login with Keycloak.
- It displays the user's name.

- It displays the user's profile picture, if available.
- It shows the **ID token**.
- It shows the **Access token**.
- It refreshes the tokens.
- It invokes the secured endpoint provided by the backend.

The backend REST API is also very simple and is implemented with Node.js. It provides a REST API with two endpoints:

- `/public`: A publicly available endpoint with no security
- `/secured`: A secured endpoint requiring an access token with the `myrealm` global role

Node.js is used for example applications as we want to make the code as easy to understand and as simple to run as possible, regardless of what programming language you are familiar with.

The following diagram shows the relationship between the frontend, the backend, and Keycloak. The frontend authenticates the users using Keycloak and then invokes the backend, which uses Keycloak to verify that the request should be permitted:

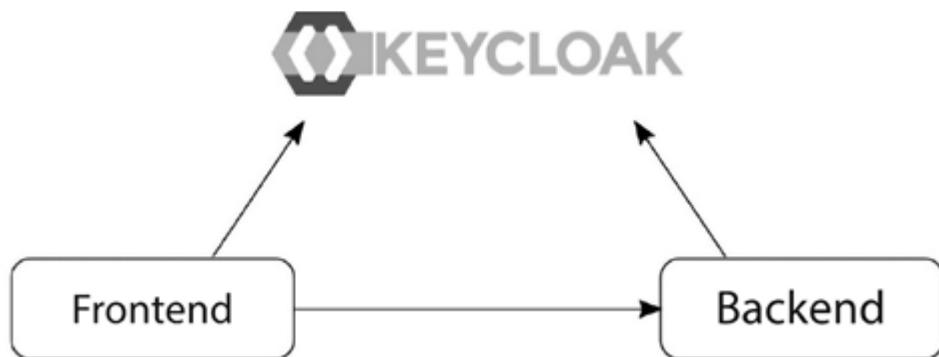


Figure 2.1: Application overview

Now that you have a basic understanding of the sample application, let's look at some more details on how it all comes together.

When the user clicks on the login button in the frontend application, the browser is redirected to the **Keycloak login page**. The user then authenticates with Keycloak, before the browser is redirected back to the application with a special code called an **authorization code**. The application then invokes Keycloak to exchange the authorization code for the following tokens:

- An ID token: This provides the application information pertaining to the authenticated user.
- An access token: The application includes this token when making a request to a service, which allows the service to verify whether the request should be permitted.
- A refresh token: Both the ID and the access token have short expirations – by default, 5 minutes. The refresh token is used by the application to obtain new tokens from Keycloak.

The flow described is what is known as the **authorization code flow** in OpenID Connect. If you are not already familiar with **OAuth 2.0** or OpenID Connect, they can be a bit daunting at first, but once you become familiar with them, they are actually quite simple and easy to understand.

To help visualize the login process, a simplified sequence diagram is provided as follows:

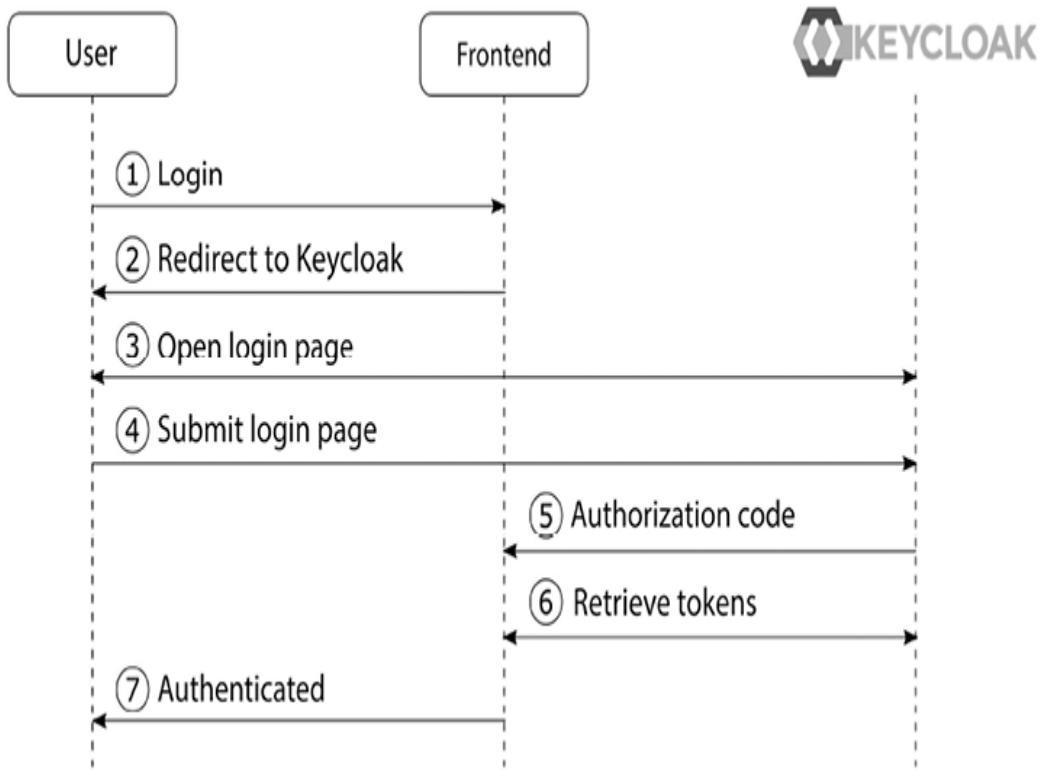


Figure 2.2: Authorization code flow in OpenID Connect simplified

The steps in this diagram are as follows:

1. The **User** clicks on the login button.
2. The application redirects to the Keycloak **Login** page.
3. The Keycloak login page is displayed to the **User**.
4. The **User** fills in the username and password and submits the results to Keycloak.
5. After verifying the username and password, Keycloak sends the **Authorization code** to the application.
6. The application exchanges the **Authorization code** for an ID token and an access token. The application can now verify the identity of the user by inspecting the ID token.

By delegating the authentication of the user to Keycloak, the application does not have to know how to authenticate the user. This is especially relevant when the authentication mechanisms change. For example, two-factor authentication can be enabled without having to make changes to the application. This also means the application does not have access to the user's credentials.

The next step related to Keycloak is when the frontend invokes the backend. The backend REST API has a protected endpoint that can only be invoked by a user with the global role, **myrole**.

To be completely accurate, the frontend is granted permissions to invoke the backend on behalf of the user. This is part of the beauty of OAuth 2.0. An application does not have access to do everything that the user is able to do, only what it should be able to do.

When the frontend makes a request to the backend, it includes the access token within the request. By default, Keycloak uses **JSON Web Signature (JWS)** as the token format. These types of tokens are often referred to as **non-opaque tokens**, meaning the contents of the token are directly visible to the application.

The token also includes a digital signature, making it possible to verify that the token was indeed issued by Keycloak. In essence, this means that the backend can both verify the token and read the contents without a request to Keycloak, resulting in less demand on the Keycloak server and lower latency when processing requests to the backend.

To help visualize what happens when the frontend sends a request to the backend, take a look at the following diagram:

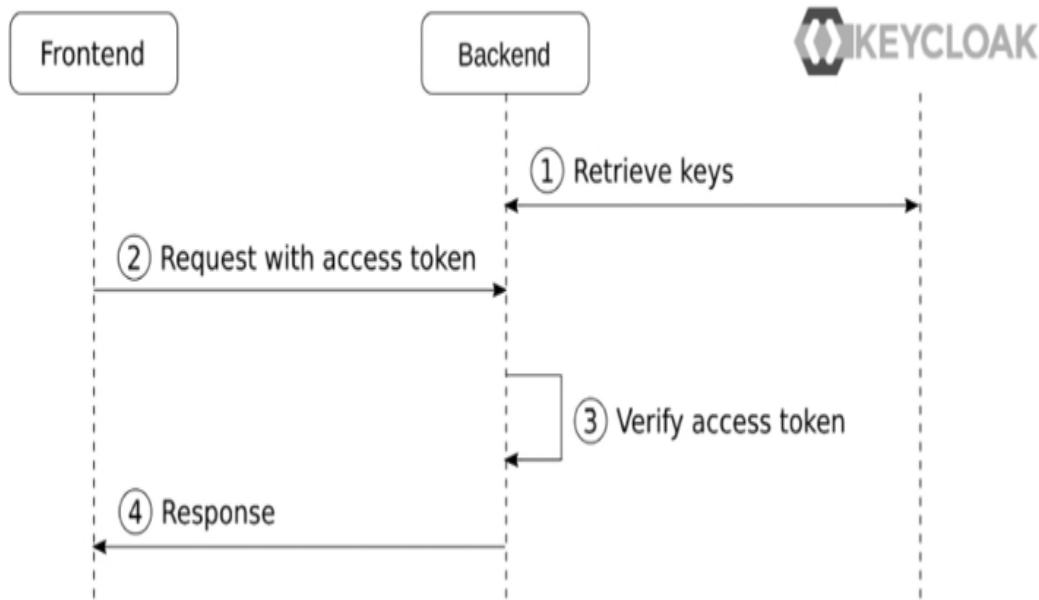


Figure 2.3: Secured request from the frontend to the backend simplified

The steps in the diagram are as follows:

1. The **Backend** retrieves Keycloak's public keys. The **Backend** does not need to do this for all requests to the **Backend**, but can instead cache the keys in memory.
2. The **Frontend** sends a request to the **Backend**, including the access token.
3. The **Backend** uses the public keys it retrieved earlier to verify that the **access token** was issued by a trusted Keycloak instance, and then verifies that the token is valid and that the token contains the role **myrole**.
4. The **Backend** returns the results to the **Frontend**.

You now have a basic understanding of how the sample applications are secured with Keycloak. In the next section, you will learn how to run the sample application.

Running the application

In this section, you will learn how to run the sample application.

If you don't already have Node.js installed on your workstation, go to <https://nodejs.org/> for instructions on how to install it.

To run the frontend on Node.js, open a terminal and run the following commands:

```
$ cd Keycloak---Identity-and-Access-Management-for-I  
$ npm install  
$ npm start
```

Next, open a new terminal to run the backend using the following commands:

```
$ cd Keycloak---Identity-and-Access-Management-for-I  
$ npm install  
$ npm start
```

Now that you have the sample application running with Node.js, you can register it with Keycloak, which we will cover in the next section.

Understanding how to log in to the application

In the previous chapter, covering how to get started with Keycloak, you learned how to run Keycloak, as well as how to create your first realm. Prior to continuing this section, you should have Keycloak running with the realm created, as covered in the previous chapter. In summary, what you require before continuing is the following:

- Keycloak up and running
- A realm named `myrealm`
- A global role named `myrole`
- A user with the preceding role

Before an application can log in with Keycloak, it has to be registered as a client with Keycloak.

Before registering the frontend, let's see what happens if an unregistered application tries to authenticate with Keycloak. Open `http://localhost:8000` and then click on the **Login** button.

You will see an error page from Keycloak with the message **Client not found**. This error is telling you that the application is not registered with Keycloak.

To register the frontend with Keycloak, open the Keycloak admin console. At the top of the menu on the left-hand side, there is an option to select what realm you are working with. Make sure you have selected the realm named `myrealm`. In the menu on the left-hand side, click on **Clients**, and then click on **Create client**.

Fill in the form with the following values:

- Client ID: `myclient`

After filling in the **Client ID** field, click on **Next**. On the following screen, it is possible to enable and disable various capabilities required by an application. For now, you can simply ignore this step and click on **Save**.

Before the client can be used by the frontend application to authenticate with Keycloak, you have to register the URL for the application. Under **Access settings**, fill in the following values:

- **Valid redirect URIs:** `http://localhost:8000/`
- **Valid post redirect URIs:** `http://localhost:8000/`
- **Web origins:** `http://localhost:8000`

Once you have filled in the form, click on **Save**. Before we continue to try to log in with the frontend application, let's look a bit more at what the last configuration values you entered mean:

- **Valid redirect URIs:** This value is very important in an OpenID Connect authorization code flow when a client-side application is used. A client-side application is not able to have any credentials as they would be visible to end users of the application. To prevent any malicious applications from being able to masquerade as the real application, the valid redirect URIs instruct Keycloak to only redirect the user to a URL that matches a valid redirect URI. In this case, since the value is set to `http://localhost:8000/`, an application hosted on <http://attacker.com> would not be able to authenticate.
- **Valid post redirect URIs:** This is the same as the previous value, but for logout requests rather than login requests, as it is

fairly common for an application to have different redirect URIs for login and logout. Keycloak supports adding a special post redirect URI with the value +, which results in permitting all valid redirect URIs as post redirect URIs.

- **Web origins:** This option registers the valid web origins of the application for **Cross-Origin Resource Sharing (CORS)** requests. To obtain tokens from Keycloak, the frontend application has to send an AJAX request to Keycloak, and browsers do not permit an AJAX request from one web origin to another, unless **CORS** is used. Keycloak supports adding a special web origin with the value +, which results in permitting all valid redirect URIs as web origins.

The following screenshot shows the created client in the Keycloak admin console.

General Settings

Client ID <small>?</small>	myclient
Name <small>?</small>	
Description <small>?</small>	
Always display in console <small>?</small>	<input checked="" type="radio"/> Off
Access settings	
Root URL <small>?</small>	
Home URL <small>?</small>	
Valid redirect URIs <small>?</small>	http://localhost:8000/ + Add valid redirect URIs
Valid post logout redirect URIs <small>?</small>	http://localhost:8000/ + Add valid post logout redirect URIs
Web origins <small>?</small>	http://localhost:8000/ + Add web origins

Figure 2.4: Client settings in the admin console

Now you can go back to the frontend by opening <http://localhost:8000>. This time, when you click on the **Login** button, you will see the Keycloak login page. Log in with the username and password you created during the previous chapter. Let's take a look at the ID token that Keycloak issued. Click on the **Show ID Token** button. The ID token that is displayed will look

something like the following:

```
{  
    "exp": 1664300152,  
    "iat": 1664299852,  
    "auth_time": 1664298915,  
    "jti": "21bb9f32-98ce-49aa-896d-796cb716be59",  
    "iss": "http://localhost:8080/realm/myrealm",  
    "aud": "myclient",  
    "sub": "eb14ea82-45e2-4413-8997-129fd0fc865b",  
    "typ": "ID",  
    "azp": "myclient",  
    "nonce": "ccf5f374-aa07-4280-b63a-efdba9c355c9",  
    "session_state": "22884115-55cb-4285-ba92-26c4bf74f74b",  
    "at_hash": "ngdMORpXQcEQJ6d9s3uHvw",  
    "acr": "0",  
    "sid": "22884115-55cb-4285-ba92-26c4bf74f74b",  
    "email_verified": true,  
    "name": "Stian Thorgersen",  
    "preferred_username": "st",  
    "given_name": "Stian",  
    "family_name": "Thorgersen",  
    "email": "st@localdomain.localhost"  
}
```

Here is a list of some of the more interesting values within the ID token:

- **exp**: This is the date and time the token expires in seconds since 01/01/1970 00:00:00 UTC (often referred to as Unix or

Epoch time).

- **iss**: This is the issuer of the token, which you may notice is the URL of the Keycloak realm.
- **sub**: This is the unique identifier of the authenticated user.
- **name**: This is the first name and last name of the authenticated user.
- **preferred_username**: This is the username of the authenticated user. You should avoid this as a key for the user as it may be changed, and may even refer to a different user in the future. Instead, always use the sub field for the user key.

The ID token is used by the application to establish the identity of the authenticated user.

Next, let's take a look at the access token. Click on the **Show Access Token** button. Let's also take a look at some fields in this token:

- **allowed-origins**: This is a list of permitted web origins for the application. The backend service can use this field when deciding whether web origins should be permitted for CORS requests.
- **realm_access**: This contains a list of global realm roles. It is the intersection between the roles granted to the user, and the roles the client has access to.
- **resource_access**: This contains a list of client roles.
- **scope**: Scopes can be used both to decide what fields (or claims) to include in the token and by backends to decide what APIs the token can access.

Currently, the information within the tokens is the default fields available in Keycloak. If you want to add additional information, Keycloak is very flexible in allowing you to customize the content within the tokens.

Let's give this a go by adding a picture for the user. Leave the tab with the frontend open, and then open a new tab with the Keycloak admin console. In the menu on the left-hand side, click on **Users**, and select the user you created previously. Now let's add a custom attribute to the user. Click on **Attributes**. In the table, there will be two empty input fields at the bottom. In the **Key** column, set the value to **picture**, and in the **Value** column, set the value to the URL of a profile picture (in the following screenshot, I've used my GitHub avatar). Then, click on **Save**.

The screenshot shows the Keycloak Admin Console interface for managing user attributes. At the top, a navigation bar includes tabs for Details, Attributes (which is selected and highlighted in blue), Credentials, Role mapping, Groups, Consents, Identity provider links, and Sessions. Below the navigation bar is a table for attributes. The first row contains a 'Key' column with 'picture' and a 'Value' column with the URL 'https://avatars.githubusercontent.com/u/2271511'. Below this row, there are two empty rows for adding new attributes, each with a 'Key' field containing 'Type a key' and a 'Value' field containing 'Type a value'. At the bottom of the table area, there is a link to 'Add an attribute' with a plus sign icon. At the very bottom of the screenshot, there are two buttons: a blue 'Save' button and a 'Revert' button.

Figure 2.5: Adding a custom attribute to a user

Now, go back to the tab where you have the frontend open. To display the profile picture, you can click on the **Refresh** button.

When you click on this button, the tokens will be refreshed, and the new ID token will now contain the picture attribute you just added, which allows the application to display a profile picture for the user.

Next, you will learn how to securely invoke the backend from the frontend.

Securely invoking the backend REST API

Now, open `http://localhost:3000/` and click on the **Public endpoint** link. You will see a message saying **Public message!**. The public endpoint is not secured by Keycloak and can be invoked without an access token.

Next, let's try the secured endpoint that is protected by Keycloak. Open `http://localhost:3000/` again. This time, click on the **Secured endpoint** link. Now you will see a message saying **Access denied**. This request is not permitted since it requires a valid access token to invoke the endpoint.

Let's now try to invoke the secured endpoint from the frontend. Open `http://localhost:8000/` and click on **Invoke Service**. You will now see a message displayed saying **Secret message!**. If instead you get the message **Access Denied**, this is most likely caused by the user not having the `myrole` role.

When you click **Invoke Service**, the frontend sends an AJAX request to the backend service, including the access token in the request, which allows the backend to verify that the invocation is

done on behalf of a user who has the required role to access the endpoint.

Summary

In this chapter, you learned how to secure your first application, consisting of a frontend web application and a backend REST API with Keycloak. You also gained a basic understanding of how Keycloak leverages OpenID Connect to make this all happen in a standard and secure way. Together with what you learned in the first chapter of the book, you now have a solid foundation to start learning more about Keycloak.

In the next chapter, we will dive deeper into securing applications with Keycloak, giving you a better understanding of how it all works.

Questions

1. How does an application authenticate with Keycloak?
2. What do you need to configure in the Keycloak admin console in order to allow an application to authenticate with Keycloak?
3. How does an application securely invoke a protected backend service?

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/SecNet>



3

Brief Introduction to Standards

In this chapter, you will get a brief introduction to the **standards** that enable you to integrate your applications securely and easily with Keycloak. We very briefly cover **OAuth 2.0**, **OpenID Connect**, **JSON Web Tokens (JWT)**, and **Security Assertion Markup Language 2.0 (SAML 2.0)**. If you are new to these standards, this chapter will give you a gentle introduction without going too much into detail. Even if you are fairly familiar with these standards, you may still want to skim through this chapter.

By the end of this chapter, you will have a basic understanding of OAuth 2.0, OpenID Connect, JWT, and SAML 2.0, along with a decent understanding of what these standards can offer you.

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

- Authorizing application access with OAuth 2.0
- Authenticating users with OpenID Connect
- Leveraging JWT for tokens
- Understanding why SAML 2.0 is still relevant

Authorizing application access with OAuth 2.0

OAuth 2.0 is by now a massively popular industry-standard protocol for authorization.

At the heart of OAuth 2.0 sits the OAuth 2.0 framework, which has enabled a whole ecosystem of websites to integrate with each other. Prior to OAuth 2.0, there was OAuth 1.0, as well as more bespoke solutions to allow third-party applications to access data on behalf of the user, but these approaches were complex or not easily interoperable.

With OAuth 2.0, sharing user data to third-party applications is easy, doesn't require sharing user credentials, and allows control over what data is shared.

OAuth 2.0 is not only useful when dealing with *third-party applications*. It is also incredibly useful for *limiting access* to your own *applications*. Just as it wasn't uncommon for third-party applications to ask for your username and password to other sites, this was a common pattern within an enterprise as well.

Applications would, for example, ask for your *LDAP username* and *password*, which would then be used to access other services within the enterprise. This effectively meant that if one application was compromised, all services within the enterprise could also be compromised.

There are four roles defined in OAuth 2.0:

- **Resource owner:** This is typically the end user that owns the resources an application wants to access.
- **Resource server:** This is the service hosting the protected resources.

- **Client:** This is the application that would like to access the resource.
- **Authorization server:** This is the server issuing access to the client, which is the role of Keycloak.

In essence, in an OAuth 2.0 protocol flow, the client requests access to a resource on behalf of a resource owner from the authorization server. The authorization server issues limited access to the resource in the form of an access token. After receiving the access token, the client can access the resource at the resource server by including the access token in the request.

Depending on the application type and use case, there are a number of different flows that can be used. To help you decide what flow type you should use for your application, you can use the following simple formula:

- If the application accesses the resource on behalf of itself (the application is the resource owner), use the **Client Credentials flow**.
- If the application runs on a device without a browser or is input-constrained, use the **Device flow**. This could, for example, be a **smart TV** where it would be difficult for the user to enter the username and password.
- If none of the preceding conditions are applicable, use the **authorization code flow**.

In addition, there are two more flow types that are now deprecated by the OAuth 2.0 Security Best Current Practice specification and should not be used:

- **Implicit flow:** This was a simplified flow for native applications and client-side applications, which is now considered insecure and should not be used.
- **Resource Owner Password Credentials flow:** In this flow, the application collects the user's credentials directly and exchanges them for an access token. It may be tempting to use this grant type for native applications, when a browser is not available, or simply because you want the login form to be directly integrated with your application. You should not be tempted, though. It is inherently insecure as you expose the user's credentials directly to the application, and you will also run into other problems in the long run, when you want your users to use stronger authentication than only a password, for example.

If you are not already familiar with the authorization code flow in OAuth 2.0, the following diagram will help you understand how it works:

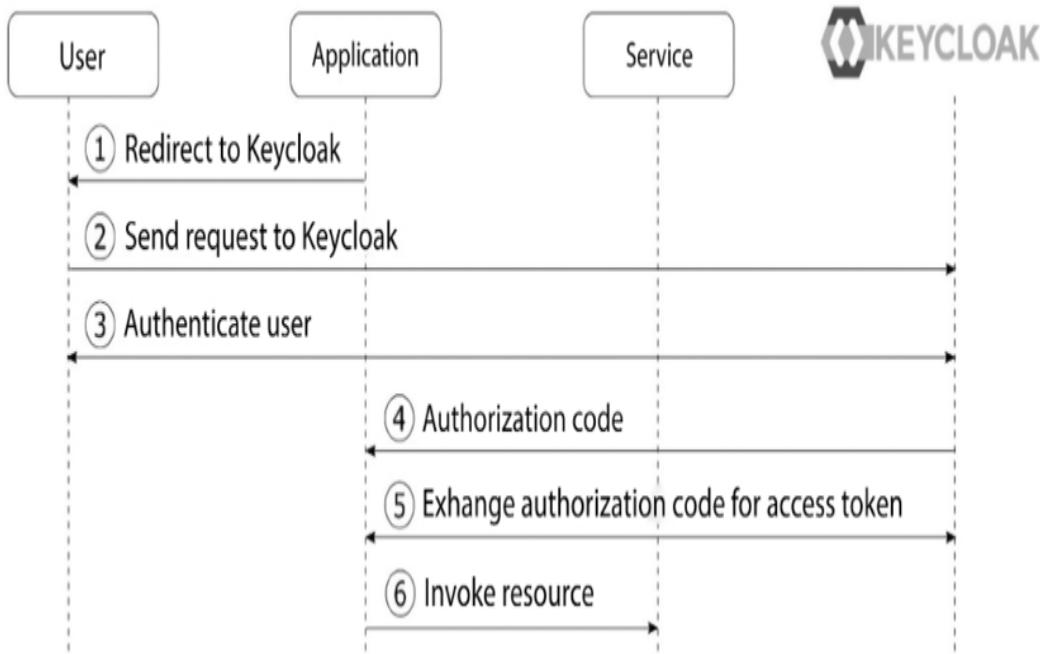


Figure 3.1: OAuth 2.0 Authorization Code grant type simplified

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

1. The **application** prepares what is called an **authorization request**, and requests the user's browser to be redirected to **Keycloak**.
2. The **user's** browser redirects the user to **Keycloak** at an endpoint called the **authorization endpoint**.
3. If the **user** is not already authenticated with **Keycloak**, **Keycloak** authenticates the user. Once authenticated, **Keycloak** asks the **user** to provide their consent to allow the application to access the service on their behalf.
4. The **application** receives an **authorization code** from **Keycloak** in the form of an **authorization response**.
5. The **application** exchanges the authorization code for an access token through an **access token** request to the **token endpoint** at **Keycloak**.

6. The **application** can now use the access token to invoke the protected resource.

Within an OAuth 2.0 flow, there are two client types, which are **confidential** and **public** clients. **Confidential clients** are applications such as server-side web applications that are able to safely store credentials, which they can use to authenticate with the authorization server. **Public clients**, on the other hand, are client-side applications that are not able to safely store credentials. As public clients are not able to authenticate with the authorization server, there are two safeguards in place:

- The authorization server will only send the authorization code to an application hosted on a pre-configured URL, in the form of a previously registered redirect URI.
- **Proof Key for Code Exchange (PKCE, RFC 7636)**, which is an extension to OAuth 2.0, prevents anyone that intercepts an authorization code from exchanging it for an access token.

As access tokens are passed around from the application to services, they typically have a short lifetime. To allow applications to obtain new access tokens without going through the complete flow, a refresh token is used. A refresh token should be kept securely by the application and can be used by the application to obtain new access tokens.

In addition to the core OAuth 2.0 framework, there are a few additional specifications you should be aware of:

- **Bearer Tokens (RFC 6750)**: OAuth 2.0 does not describe the type of access token, or how it should be used. Bearer tokens

are by far the most commonly used type of access tokens, and they are typically sent to resource servers through the HTTP Authorization header. They can also be sent in the form-encoded body, or as a query parameter. An important thing to note here is that sending bearer tokens as a query parameter has inherent security weaknesses and should be avoided.

- **Token Introspection (RFC 7662)**: In OAuth 2.0, the contents of access tokens are opaque to applications, which means the content of the access token is not readable by the application. The token introspection endpoint allows the client to obtain information about the access token without understanding its format.
- **Token Revocation (RFC 7009)**: OAuth 2.0 considers how access tokens are issued to applications, but not how they are revoked. This is covered by the token revocation endpoint.

There are also a number of best practices on how you should use OAuth 2.0. There are recommendations for **native applications** and **browser-based applications**, as well as security considerations and best practices. We will cover these in later chapters.

By now, you should have a basic understanding of what OAuth 2.0 is and how you can leverage it in your applications. Don't worry if you don't fully understand all the details as we will come back to this subject later. In most cases, you will not be required to have a deep understanding of OAuth 2.0 in order to use it, as you should use a library that hides its complexity from you and that helps you apply it in the correct way for your application.

One thing you may have noticed is that although OAuth 2.0 can grant access to resources, it does not cover the authentication of users. This is covered by an extension to OAuth 2.0 called **OpenID Connect**, which we will look at next.

Authenticating users with OpenID Connect

While OAuth 2.0 is a protocol for authorization, it does not cover authentication. OpenID Connect builds on top of OAuth 2.0 to add an authentication layer.

At the heart of OpenID Connect sits the **OpenID Connect Core specification**, which has enabled a whole ecosystem of websites to no longer need to deal with user management and authenticating users. In addition, it has significantly reduced the number of times a user has to authenticate, as well as the number of different passwords a user has to juggle, that is, if they care about using unique passwords for all websites they access. Just think about the endless number of websites that allow you to sign in using **Google**, or other social networks.

OpenID Connect has not only enabled social login but is also, of course, very useful within an **enterprise** in order to have a centralized solution for authentication, supporting **single sign-on**. This also significantly increases security as applications don't have access to the **user credentials** directly. It also enables the use of **stronger authentication**, such as **OTP** or **WebAuthn**, without the need to support it directly within applications.

Not only does OpenID Connect enable easy authentication within the enterprise but it also enables you to allow third parties such as employees at partner companies to access applications within your enterprise without having to create individual accounts within your enterprise.

Like OAuth 2.0, OpenID Connect defines a number of roles involved in the protocol:

- **End User:** This is the equivalent of the resource owner in OAuth 2.0. It is, of course, the human being that is authenticating.
- **Relying Party (RP):** A somewhat confusing term for the application that would like to authenticate the end user. It is called the RP as it is a party that relies on the **OpenID Provider (OP)** to verify the identity of the user.
- **OpenID Provider (OP):** The identity provider that is authenticating the user, which is the role of Keycloak.

In essence, in an OpenID Connect protocol flow, the RP requests the identity of the end user from the OP. As it builds on top of OAuth 2.0 at the same time as the identity of the user is requested, it can also obtain an access token.

OpenID Connect utilizes the *Authorization Code* grant type from OAuth 2.0. The main difference is that the client includes *scope=openid* in the initial request, which makes it an **authentication request**, rather than an **authorization request**.

While OAuth 2.0 calls the different flows grant types, OpenID Connect refers to them as flows. There are two flows in OpenID

Connect that you should care about:

- **Authorization code flow:** This uses the same flow as the OAuth 2.0 Authorization Code grant type and returns an authorization code like OAuth 2.0, which can be exchanged for an ID token, an access token, and a refresh token.
- **Hybrid flow:** In the Hybrid flow, the ID token is returned from the initial request alongside an authorization code.

Just like OAuth 2.0, OpenID Connect also defines the Implicit flow. However, as mentioned earlier, the Implicit flow is considered legacy and no longer recommended.

OpenID Connect does not define equivalents to the Client Credential flow and the Device flow. This makes sense as neither of these flows requires authenticating users, instead just granting access to a service.

If you are not already familiar with the authorization code flow in OpenID Connect, the following diagram will help you understand how it works:

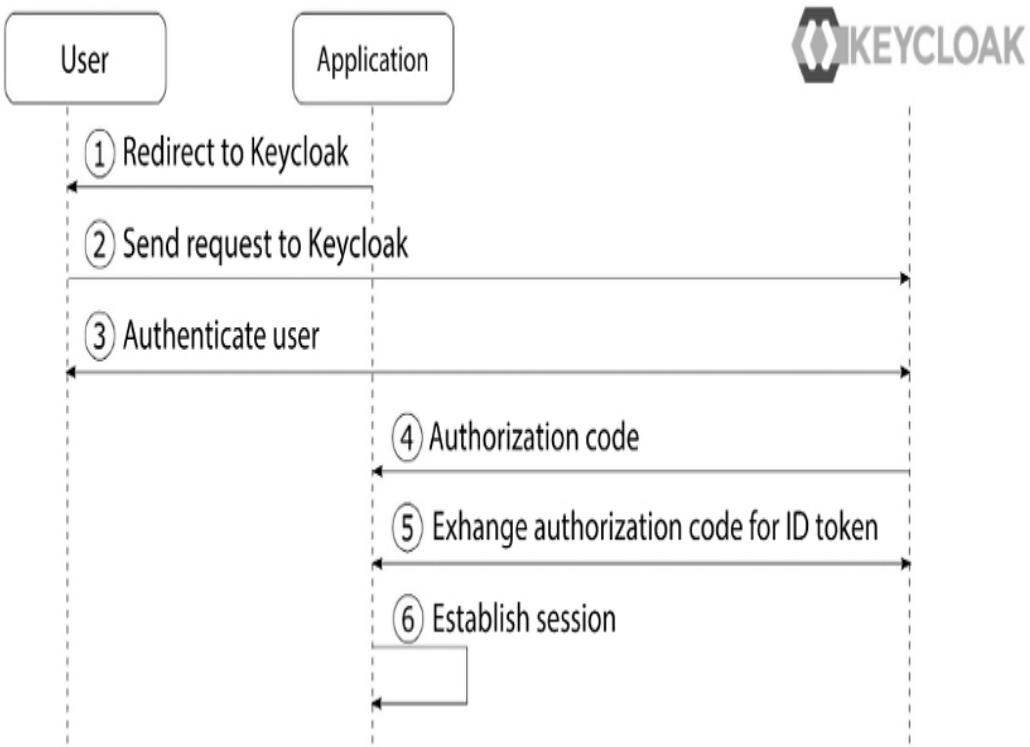


Figure 3.2: OpenID Connect authorization code flow simplified

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

1. The **application** prepares what is called an authentication request, and requests the **user's** browser to be redirected to **Keycloak**.
2. The **user's** browser redirects the user to **Keycloak** at an endpoint called the authorization endpoint.
3. If the **user** is not already authenticated with **Keycloak**, **Keycloak** authenticates the **user**.
4. The **application** receives an authorization code from **Keycloak** in the form of an authentication response.
5. The **application** exchanges the authorization code for an ID token and an access token through a token request to the token endpoint at **Keycloak**.

6. The **application** now has the ID token, which it can use to discover the user's identity, and can establish an authenticated session for the user.

In addition to the OpenID Connect Core specification, there are a few additional specifications you should be aware of:

- **Discovery**: Allows clients to dynamically discover information about the OP.
- **Dynamic Registration**: Allows clients to dynamically register themselves with the OP.
- **Session Management**: Defines how to monitor the end user's authentication session with the OP, and how the client can initiate a logout.
- **Front-Channel Logout**: Defines a mechanism for single sign-out of multiple applications using embedded iframes.
- **Back-Channel Logout**: Defines a mechanism for single sign-out for multiple applications using a back-channel request mechanism, which we will cover in the next chapter.

OpenID Connect has two additional concepts on top of OAuth 2.0. It clearly specifies the format of the ID token by leveraging the JWT specification, which, unlike the access token in OAuth 2.0, is not opaque. It has a well-specified format, and the values (called claims) within the token can be directly read by the client. This allows the clients to discover information about the authenticated user in a standard way. In addition, it defines a **userinfo endpoint**, which can be invoked with an access token and returns the same standard claims as found in the ID token. In the next chapter, we will cover

the `userinfo endpoint` in more detail, including how you can control what information is returned for a user.

For use cases where an increased level of security is required, there is a set of profiles from what is called the **Financial-Grade API (FAPI)** working group. These are profiles that describe best practices of how OpenID Connect and related specifications should be used in high-risk scenarios. You should not get too hung up on the name Financial-Grade API, as there is nothing specific to finance in these profiles.

By now, you should have a basic understanding of what OpenID Connect is and how you can leverage it in your applications. Don't worry if you don't fully understand all the details as we will come back to this later. In most cases, you will not be required to have a deep understanding of OpenID Connect in order to use it, as you should use a library that hides its complexity from you and that helps you to apply it in the correct way to your application.

While OpenID Connect defines a standard format for the ID token, it also does not define any standard for the access token. In the next section, you will find out why Keycloak leverages JWT as the format for the default access tokens it issues.

Leveraging JWT for tokens

Keycloak has leveraged JWT as the format for access tokens from the very beginning of the project. This was a very conscious decision for interoperability as well as performance reasons.

Using a standard format, which is relatively easily consumable, makes it easier to integrate with Keycloak. As JWT is based on JSON, it can also easily be parsed and understood in any programming language.

In addition, as the resource servers are now able to directly read the value of the access token, they do not always have to make a request to the OAuth 2.0 token introspection endpoint, or the OpenID Connect UserInfo endpoint. This potentially eliminates two additional requests to Keycloak for a request to the resource server, reducing latency as well as significantly reducing the number of requests to Keycloak.

JWT comes from a family of specifications known as **JOSE**, which stands for **JavaScript Object Signing and Encryption**. The related specifications are as follows:

- **JSON Web Token (JWT, RFC 7519)**: Consists of two base64url-encoded JSON documents separated by a dot, a header, and a set of claims.
- **JSON Web Signature (JWS, RFC 7515)**: Adds a digital signature of the header and the claims.
- **JSON Web Encryption (JWE, RFC 7516)**: Encrypts the claims.
- **JSON Web Algorithms (JWA, RFC 7518)**: Defines the cryptographic algorithms that should be leveraged for JWS and JWE.
- **JSON Web Key (JWK, RFC 7517)**: Defines a format to represent cryptographic keys in JSON format.

In addition to the preceding specifications, the OpenID Connect Discovery endpoint advertises an endpoint where the **JSON Web**

Key Set (JWKS) can be retrieved, as well as what signing and encryption mechanisms from the JWA specification are supported.

When a resource server receives an access token, it is able to verify the token in the following ways:

- Retrieving the JWKS URL from the OpenID Connect Discovery endpoint.
- Downloading the public signing keys for the OP from the JWKS URL endpoint. These are typically cached/stored at the Resource Server.
- Verifying the signature of the token using the public signing keys from the OP.

There are some potential issues with the JWT specifications that can lead to unexpected vulnerabilities if care is not taken when validating a JWT. Let's take a look at two examples of vulnerabilities that can occur through the incorrect application of these specifications:

- **alg=none**: Interestingly enough, the JWS specification defines an algorithm value that is **none**. This basically means the JWS is unsigned. As this is a valid value, a JWT library could tell you a JWS is valid even though it has not actually been signed.
- **RSA to HMAC**: Another well-known issue is using the public RSA key, but setting the algorithm to HMAC. Some libraries blindly accept these types of tokens as they simply pick up the public RSA key and use it as the HMAC secret key.

These types of vulnerabilities can be avoided with a few relatively simple steps:

- Do not accept `alg=none`.
- Only use a key for the algorithm and the use (signing or encryption) it is intended for, and don't blindly trust the values in the JWT header.

In general, you would want to pick up a trusted JWT library and make sure you used it in the correct way. Or, even better, use an OpenID Connect/OAuth 2.0 library that supports JWT as the access token, which can do it properly for you. If neither option is available to you, it is very likely safer to use the token introspection endpoint than to try to validate the token yourself. We will cover this in more detail in *Chapter 5, Authorizing Access with OAuth 2.0*.

By now, you should have a basic understanding of OAuth 2.0, OpenID Connect, and JWT. In the next section, we will take a look at a significantly more mature specification, SAML 2.0.

Understanding why SAML 2.0 is still relevant

SAML 2.0 is a mature and robust protocol for authentication and authorization. It is very widely used to enable *single sign-on* within enterprises and other domains, such as **education** and **government**. It was ratified as an *OASIS Standard* in March 2005, so has been around for a considerable amount of time.

SAML 2.0 is very widely available within enterprise applications, enabling you to easily allow your existing users to authenticate to new applications you wish to deploy. Not only is it available in self-hosted applications but it is also available as an option for a large

number of Software-as-a-Service solutions, such as Salesforce, Google Apps, and Office 365. For enterprises, this is a great option when choosing hosted solutions in the cloud as it quickly enables you to allow all your employees access to these solutions, without having to create accounts for each individual employee.

Even though SAML 2.0 is more mature and perhaps also more widely used, you may still want to favor OpenID Connect over SAML 2.0 for new applications. OpenID Connect has a stronger focus on modern architecture, such as single-page applications, mobile applications, REST APIs, and microservices, which means it is a better fit for the future. Developers will also often find that OpenID Connect is simpler to understand, due to OpenID Connect leveraging JSON and simple query parameters, while SAML 2.0 uses more complicated XML documents.

If you are unfamiliar with the details of OAuth 2.0, OpenID Connect, and SAML 2.0, we recommend starting by learning OAuth 2.0 and OpenID Connect. For this reason, we are not going to cover SAML 2.0 in this book.

That being said, SAML 2.0 is still important today. You will often find that SAML 2.0 is available to you as an option, while OpenID Connect is not. You may also find that SAML 2.0 is a better fit for your particular use case or, due to internal policies or compliance, you may be required to use SAML 2.0. The great thing about Keycloak is that both options are available to you. You can also seamlessly combine applications using OpenID Connect with applications using SAML 2.0 in the same single sign-on experience.

Summary

In this chapter, you learned how to use OAuth 2.0 to provide your applications, as well as third-party applications, with access to services without exposing credentials, as well as only giving applications exactly what access they need. You also learned how OpenID Connect can be leveraged for single sign-on to your applications, as well as allowing external users to access your applications. Finally, you learned how SAML 2.0 is still an important standard that you should be aware of, even though you may not want to choose it for your own applications.

In the next chapter, you will get a deeper understanding of OAuth 2.0 with a practical guide on how you can use Keycloak to leverage this standard in your applications.

Questions

1. How does OAuth 2.0 allow an application to access resources provided by a different application without asking for the user's username and password?
2. What does OpenID Connect add to OAuth 2.0?
3. What does JWT add to OAuth 2.0?

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/SecNet>



4

Authenticating Users with OpenID Connect

In this chapter, you will get a deeper understanding of how Keycloak enables you to authenticate users in your applications by leveraging the OpenID Connect standard. Through using a sample application that was written for this book, we will see the first-hand interaction between an application and Keycloak, including the contents of requests and responses.

By the end of this chapter, you will have a good understanding of OpenID Connect, including how to authenticate users, understand the ID token, and deal with users logging out.

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

- Running the OpenID Connect playground
- Understanding the Discovery endpoint
- Authenticating a user
- Understanding the ID token
- Invoking the UserInfo endpoint
- Dealing with users logging out

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 clone of the GitHub repository associated with the book. The GitHub repository is available at <https://github.com/PacktPublishing/Keycloak---Identity-and-Access-Management-for-Modern-Applications-2nd-Edition>.

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

<https://packt.link/xZ3iN>

Running the OpenID Connect playground

The **OpenID Connect (OIDC)** playground application was developed specifically for this book in order to make it as easy as possible for you to understand and experiment with OIDC in a practical way.

The playground application does not use any libraries for OIDC, but rather all OIDC requests are crafted by the application itself. One thing to note here is that this application is not implementing OIDC in a secure way, and is ignoring optional parameters in the requests that are important for a production application. There are two reasons for this. Firstly, it is so you can focus on understanding the general concepts of OIDC. Secondly, if you decide to implement

your own application libraries for OIDC, you should have a very good understanding of the specifications, and it is beyond the scope of this book to cover OIDC in that much detail.

Before continuing with reading this chapter, you should start the OIDC playground application, as it will be used throughout the rest of the chapter.

To run the OIDC playground application, open a terminal and run the following commands:

```
$ cd Keycloak---Identity-and-Access-Management-for-I  
$ npm install  
$ npm start
```

To verify the application is running, open <http://localhost:8000/> in your browser. The following screenshot shows the OIDC playground application page:

OpenID Connect Playground

1 - Discovery

2 - Authentication

3 - Token

4 - Refresh

5 - UserInfo

Reset

Discovery

Issuer

<http://localhost:8080/realm/myrealm>

[Load OpenID Provider Configuration](#)

OpenID Provider Configuration

Figure 4.1: The OpenID Connect playground application

In order to be able to use the playground application, you need Keycloak running, a realm with a user that you can log in with, and a client with the following configuration:

- **Client ID:** `oidc-playground`
- **Client authentication:** `Off`
- **Authentication flow:** `Standard flow`
- **Valid Redirect URIs:** `http://localhost:8000/`
- **Web Origins:** `http://localhost:8000`

If you are unsure about how to do this, you should refer to *Chapter 1, Getting Started with Keycloak*, and *Chapter 2, Securing Your First Application*.

In the next section, we will start taking a deeper look at OIDC by leveraging the playground application, starting with understanding how applications can discover information about an OpenID Provider.

Understanding the Discovery endpoint

The OIDC Discovery specification is an important aspect of both the interoperability and usability of OIDC Relying Party libraries. Without this specification, you would be required to do a lot of manual configuration in your applications to be able to authenticate with an OpenID Provider (more information on OpenID Providers can be found in *Chapter 3, Brief Introduction to Standards*).

It is an optional specification that an OpenID Provider can decide if it wants to implement or not. Luckily, most OpenID Providers, including Keycloak, implement this specification.

By simply knowing the base URL (often referred to as the issuer URL) for your OpenID Provider, a Relying Party can discover a lot of useful information about the provider. It does this by loading what is called the **OpenID Provider Metadata** from a standard endpoint, namely `<base URL>/.well-known/openid-configuration`.

To better understand the OpenID Provider Metadata, open the OIDC playground in your browser. You can see there is already a value filled in for the `issuer` input.

The value for the issuer URL that is already filled in is `http://localhost:8080/realm/myrealm`. Let's break this URL apart and take a look at the parts of the issuer URL:

- `http://localhost:8080`: This is the root URL for Keycloak. In a production system, this would obviously be a real domain name and would use HTTPS (for example, `https://auth.mycompany.com/`).
- `/realm/myrealm`: As Keycloak supports multi-tenancy, this is used to separate each realm in your Keycloak instance.

If you have Keycloak running on a different hostname or port, or have a different realm, you should change the `issuer` field. Otherwise, you can leave it as is.

Now click on **Load OpenID Provider Configuration**. When you click on this button, the playground application sends a request to `http://localhost:8080/realm/myrealm/.well-known/openid-configuration` (assuming you left the `issuer` URL untouched) and receives a response in the form of the OpenID Provider Metadata for this Keycloak instance. The returned metadata is displayed in the **OpenID Provider Configuration** section of the playground application.

The following screenshot from the playground application shows an example of the loaded OpenID Provider Metadata:

OpenID Provider Configuration

```
{
  "issuer": "http://localhost:8080/realm/myrealm",
  "authorization_endpoint": "http://localhost:8080/realm/myrealm/protocol/openid-connect/auth",
  "token_endpoint": "http://localhost:8080/realm/myrealm/protocol/openid-connect/token",
  "introspection_endpoint": "http://localhost:8080/realm/myrealm/protocol/openid-connect/token/introspect",
  "userinfo_endpoint": "http://localhost:8080/realm/myrealm/protocol/openid-connect/userinfo",
  "end_session_endpoint": "http://localhost:8080/realm/myrealm/protocol/openid-connect/logout",
  "frontchannel_logout_session_supported": true,
  "frontchannel_logout_supported": true,
  "jwks_uri": "http://localhost:8080/realm/myrealm/protocol/openid-connect/certs",
  "check_session_iframe": "http://localhost:8080/realm/myrealm/protocol/openid-connect/login-status-iframe.html",
  "grant_types_supported": [
    "authorization_code",
    "implicit",
    "refresh_token",
    "password",
    "client_credentials",
    "urn:ietf:params:oauth:grant-type:device_code",
    "urn:openid:params:grant-type:ciba"
  ],
  "acr_values_supported": [
    "110"
  ]
}
```

Figure 4.2: OpenID Provider Metadata for Keycloak

In the following list, we'll take a look at what some of these values mean:

- **authorization_endpoint**: The URL to use for authentication requests
 - **token_endpoint**: The URL to use for token requests
 - **introspection_endpoint**: The URL to use for introspection requests
 - **userinfo_endpoint**: The URL to use for UserInfo requests
 - **grant_types_supported**: The list of supported grant types
 - **response_types_supported**: The list of supported response types

With all of this metadata, the Relying Party can make intelligent decisions about how to use the OpenID Provider, including what endpoints to send requests to and what grant types and response types it can use.

If you took an extra good look at the metadata, you may have noticed that Keycloak supports the `authorization_code` grant type and the `code` and `token` response types. This is good news since we'll use this grant type and these response types to authenticate the user in our playground application in the next section.

Authenticating a user

The most common way to authenticate a user with Keycloak is through the OpenID Connect authorization code flow.

In summary, to authenticate a user with this flow, an application redirects to Keycloak, which displays a login page to authenticate the user. After the user has authenticated, the application receives an ID token, which contains information about the user.

In the following diagram, the authorization code flow is shown in more detail:

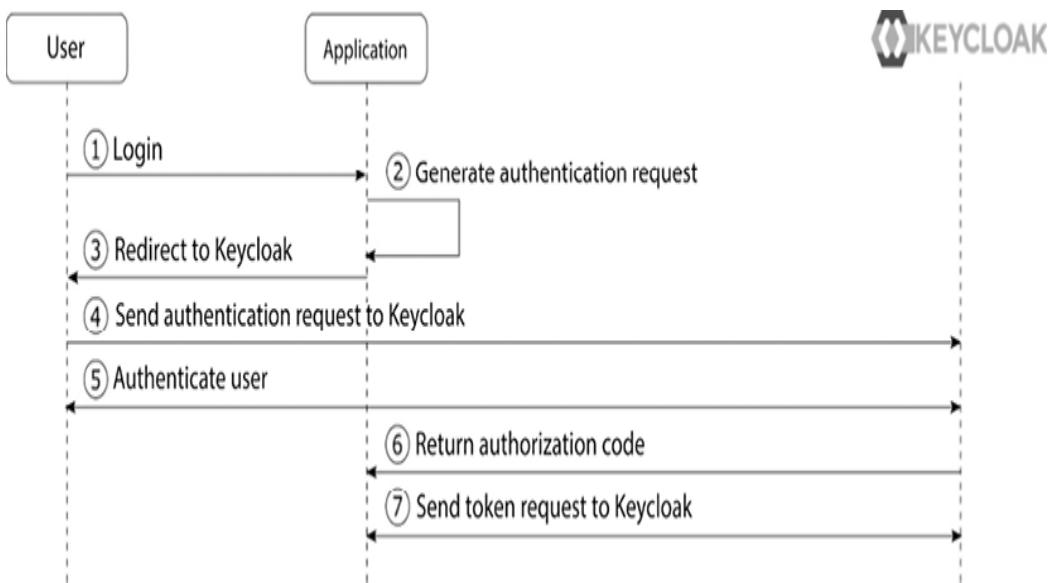


Figure 4.3: The authorization code flow

The steps from the diagram are explained in more detail as follows:

1. The **user** clicks on a login button in the application.
2. The **application generates an authentication request**.
3. The authentication request is sent to the user in form of a 302 redirect, instructing the user-agent to redirect to the authorization endpoint provided by Keycloak.
4. The user-agent opens the authorization endpoint with the query parameters specified by the **application** via the **authentication request**.
5. Keycloak displays a **login** page to the **user**. The **user** enters their username and credentials and submits the form.
6. After Keycloak has verified the user's credentials, it creates an **authorization code**, which is returned to the **application**.
7. The **application** can now exchange the **authorization code** for the **ID token**, as well as a refresh **token**.

8. Let's give this a go by going back to the OIDC playground application. As you already loaded the OpenID Provider Metadata in the previous section, the playground application already knows where to send the authentication request. To send an authentication request, click on the button labeled **2 - Authentication**.

The form that is displayed has the following values that you should fill in:

- **client_id**: This is the client ID for the application registered with Keycloak. If you used a different value than **oidc-playground** when creating the client, you should change this value.
- **scope**: The default value is **openid**, which means we will be doing an OpenID request. Leave this as is for now.
- **prompt**: This can be used for a few different purposes. For example, if you enter the value **none** in this field, Keycloak will not display a login screen to the user, but will instead only authenticate the user if the user is already logged in with Keycloak. You can also use the value **login** to require the user to log in again even if they are already logged in with Keycloak.
- **max_age**: This is the maximum number of seconds since the last time the user authenticated with Keycloak. If, for example, you set this field to **60**, it means that Keycloak will re-authenticate the user if it was more than 60 seconds since the user last authenticated.
- **login_hint**: If the application happens to know the username of the user that it wants to authenticate, it can use this

parameter to have the username filled in automatically on the login page.

Now let's take a look at what the authentication request will look like by clicking on the button labeled **Generate Authentication Request**. You will now see the actual request that the application will redirect the user-agent to in order to initiate the authentication.

The following screenshot from the playground application shows an example **Authentication Request**:

Authentication Request

```
http://localhost:8080/realm/myrealm/protocol/openid-connect/auth  
client_id=oidc-playground  
response_type=code  
redirect_uri=http://localhost:8000/  
scope=openid
```

Figure 4.4: Authentication request

This includes setting the `response_type` parameter to `code`, meaning that the application wants to receive an authorization code from Keycloak.

Next, click on the button labeled **Send Authentication Request**. You will now be redirected to the Keycloak login pages. Fill in the username and password for your user and click on **Log In**.

If you want to experiment a bit you can, for example, try the following steps:

- **Set prompt to login:** With this value, Keycloak should always ask you to re-authenticate.
- **Set max_age to 60:** With this value, Keycloak will re-authenticate you if you wait for at least 60 seconds since the last time you authenticated.
- **Set login_hint to your username:** This should prefill the username in the Keycloak login page.

If you try any of the preceding steps, don't forget to generate and send the authentication request again to see how Keycloak behaves.

After Keycloak has redirected back to the playground application, you will see the authentication response in the **Authentication Response** section. The code is what is called the **authorization code**, which the application uses to obtain the ID token and the refresh token.

Now that the application has the authorization code, you can go ahead and exchange it for some tokens.

Click on the button labeled **3 - Token**. You will see the authorization code has already been filled in on the form so you can go ahead and click on the button labeled **Send Token Request**.

Under **Token Request**, you can see the request the application sends to the token endpoint provided by Keycloak. It contains the authorization code and sets the `grant_type` to `authorization_code`, which means the application wants to exchange an authorization code for tokens.

An example **Token Request** is shown in the following screenshot from the playground application:

Token Request

```
http://localhost:8080/realm/myrealm/protocol/openid-connect/token  
  
grant_type=authorization_code  
code=14472ec2-69aa-4c79-8b8b-c6e2b3cc1877.3b403853-8afa-4b0a-9f29-84b689768d1a.73b9bd53-1831-485a-940b-986e1cf1add7  
client_id=oidc-playground  
redirect_uri=http://localhost:8000/
```

Figure 4.5: Token request

Under **Token Response**, you can see the response that Keycloak sent to the application. If you get the error with the value **invalid_grant**, it is most likely for one of the following two reasons:

- **You did the steps a bit too slowly:** The authorization code is only valid for one minute by default, so if more than one minute passes between receiving the authentication response from Keycloak and sending the token request, the request will fail.
- **You sent the token request more than once:** The authorization code is only valid once, so if it is included in more than one token request the request will fail.

The following screenshot shows an example successful **Token Response** from the playground application:

Token Response

```
{  
    "access_token": "eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJNU1jhVmwwSFkVFU4TmNKWFpzTW8tTmxzSWFGQj1KcvlabX  
    "expires_in": 300,  
    "refresh_expires_in": 1800,  
    "refresh_token": "eyJhbGciOiIuUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJkMTazMtcxYillMzA5LTQ1NWYtYWAZiliYjk2NTQxYTayN  
    "token_type": "Bearer",  
    "id_token": "eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJNU1jhVmwwSFkVFU4TmNKWFpzTW8tTmxz514F6Qj1KcvlabXkobV  
    "not-before-policy": 0,  
    "session_state": "3b403853-8afa-4b0a-9f29-84b689768d1a",  
    "scope": "openid profile email"  
}
```

Figure 4.6: Token response

Let's take a look at the values within this response:

- **access_token**: This is the access token, which in Keycloak is a signed JWT. We'll look more at this in the next chapter when we cover OAuth 2.0 in more detail.
- **expires_in**: As the access token is sometimes opaque, this will give the application a hint of when the token expires.
- **refresh_token**: This is the refresh token, which we will look more at in the next section.
- **refresh_token_expires_in**: The refresh token is also opaque, and this gives the application a hint of when the refresh token expires.
- **token_type**: This is the type of the access token, which in Keycloak is always **bearer**.
- **id_token**: This is the ID token, which we will look at in more detail in the next section.
- **session_state**: This is the ID of the session the user has with Keycloak.

- **scope**: The application requests a scope from Keycloak in the authentication request, but the actual returned scope of the tokens may not match the requested scope.

In the next section, we will take a deeper look at the ID token that the playground application just received from Keycloak.

Understanding the ID token

In the previous section, you received a token response, including an ID token from Keycloak, but we didn't take a good look at what's inside the ID token.

The ID token is by default a signed **JSON Web Token (JWT)**, which follows this format:

```
<Header>.<Payload>.<Signature>
```

The header and the payload are Base64 URL-encoded JSON documents.

If you take a look at the **Token Response** in the playground application, you can see the ID token in its encoded format. An example of the encoded ID token is also shown in the following screenshot from the playground application:

```
"token_type": "Bearer"  
"id_token": "eyJhbGciOiJSUzIlNiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJNU1jhVmwwSF8kVFU4TmNKWFpzTW8tTmxzSWFGQj1KcVlabXkObV  
"not-before-policy": 0,
```

Figure 4.7: Encoded ID token

Under the **ID Token** section, you will see the decoded token broken into three parts. The header tells you what algorithm is used, the type of the payload, and the key ID of the key that was used to sign the token.

An example of a decoded **ID Token** is shown in the following screenshot from the playground application:

ID Token

Header

```
{  
  "alg": "RS256",  
  "typ": "JWT",  
  "kid": "MRRaVl0HPdTU8NcJXzMo-NlsIaFB9JqYZmy4mUeXDM"  
}
```

Payload

```
{  
  "exp": 1665296255,  
  "iat": 1665295955,  
  "auth_time": 1665295938,  
  "jti": "daf6072b-d376-47eb-ba7a-cb203336c6c3",  
  "iss": "http://localhost:8080/realm/myrealm",  
  "aud": "oidc-playground",  
  "sub": "65588621-32e8-4655-8cf8-86fb8054822e",  
  "typ": "ID",  
  "azp": "oidc-playground",  
  "session_state": "3b403853-8afa-4b0a-9f29-84b689768d1a",  
  "at_hash": "pWygilI18hq3gIpdb8-1IwA",  
  "acr": "1",  
  "sid": "3b403853-8afa-4b0a-9f29-84b689768d1a",  
  "email_verified": false,  
  "name": "Stian Thorgersen",  
  "preferred_username": "st",  
  "given_name": "Stian",  
  "family_name": "Thorgersen",  
  "email": "st@localdomain.localhost"  
}
```

Signature

```
X2hfRpAyDb4tFRWbvPGc-i79IxBoy0OYKEK9cmKaDoAy0G-0DA7zVVirbEYhkPv66C6MAUQu4EmXzJ73-aM3xdngEPUA0rYSgg-PHkiYug3IiJgDyc-
```

Figure 4.8: Decoded ID token

Let's take a look at some of the claims (values) within the ID token:

- **exp**: When the token expires.
- **iat**: When the token was issued.
- **auth_time**: When the user last authenticated.

- `jti`: The unique identifier for this token.
- `aud`: The audience of the token, which must contain the Relying Party that is authenticating the user.
- `azp`: The party the token was issued to.
- `sub`: The unique identifier for the authenticated user. When referring to a user, it is recommended to use this instead of a username or email, as they may change over time.

All times in JWTs are represented in Unix epoch time (seconds since January 1, 1970). It's not all that readable to human beings, but great for computers, and takes very little space compared to other formats. You can find a handy tool to convert epoch times to human-readable dates at <https://www.epochconverter.com/>.

In addition to the claims listed previously, there is information about the user such as the given name, family name, and preferred username.

If you take a look at the `exp` value for the ID token with <https://www.epochconverter.com/>, you will notice that the token expires in only a few minutes.

Usually, ID tokens have a short duration in order to mitigate the risk of tokens being leaked. This doesn't mean that the application has to re-authenticate the user; rather there is a separate refresh token that can be used to obtain an updated ID token. The refresh token has a much longer expiration, and can only be used directly

with Keycloak, which means Keycloak can validate that the token is still valid.

Next, let's try to refresh the ID token. Click on the button labeled **4 – Refresh**, then click on the button labeled **Send Refresh Request**.

In the **Refresh Request** window, you will see the request sent by the playground to the Keycloak token endpoint. It uses the grant type `refresh_token`, and includes the refresh token and the client ID.

The following screenshot from the playground applications shows an example refresh request:



Refresh Request

```
http://localhost:8080/realms/myrealm/protocol/openid-connect/token

grant_type=refresh_token
refresh_token=eyJhbGciOiJIUzI1NiIsInR5cCig0iAiSldUIiwia21kIiA6ICJkMTAzMTcxYjllMzA5LTQ1NWYtYWJhZiliYjk2NTQxYTAyMjkifQ
client_id=oidc-playground
scope=openid
```

Figure 4.9: Refresh request

Under **Refresh Response**, you will see the response Keycloak sent to the playground. It is pretty much the same as the response for the original token request.

The following screenshot from the playground applications shows an example refresh response:

Refresh Response

```
{  
    "access_token": "eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUiwia21kIiA6ICJNU1jhVmwwSFbKVfU4TmNKFpzTw8tTmxzSVIFGQj1KcVlabX  
    "expires_in": 300,  
    "refresh_expires_in": 1800,  
    "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cCIgOiAiSldUiwia21kIiA6ICJkMTAzMTcxYillMzA5LTQ1NWYtYWJziliYjk2NTQxYTAYM  
    "token_type": "Bearer",  
    "id_token": "eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUiwia21kIiA6ICJNU1jhVmwiSFbKVfU4TmNKFpzTw8tTmxzSWFGQj1KcVlabXkObV  
    "not-before-policy": 0,  
    "session_state": "3b403853-8afa-4b0a-9f29-84b689768d1a",  
    "scope": "openid profile email"  
}
```

Figure 4.10: Refresh response

One thing to notice here is that the refresh response also includes a refresh token. It is important that the application uses this updated **refresh token** the next time it wants to refresh the ID Token. This is important for a few reasons, including the following:

- **Key rotation:** Keycloak may rotate its signing keys, and it relies on clients receiving new refresh tokens signed with the new keys.
- **Session idle:** A client (or a session) has a feature called session idle, which means a refresh token may have shorter expiration than the associated session.
- **Refresh token leak detection:** To discover leaked refresh tokens, Keycloak will not allow the re-use of refresh tokens. This feature is currently disabled by default in Keycloak.

Finally, under ID Token you may notice that the token now has more or less the same values, except the expiration time (`exp`), the issue time (`iat`), and the token now ID (`jti`) have changed.

Another benefit of refreshing the token now is that your application can update information about the user from Keycloak without having to re-authenticate. We'll now experiment a bit with this.

For the next few sections, you should keep the playground application open. In a new browser window, open the Keycloak Admin Console, click on **Users**, and locate the user you used when authenticating to the playground application.

First, let's try to update the user profile.

Updating the user profile

Change the email, first name, and last name of the user. Then go back to the playground application and click on the **Send Refresh Request** button. You will now notice that the user profile was updated.

Now that you have tried updating the user profile, let's try to add a custom property to the user.

Adding a custom property

Let's take a look at the steps to add a custom property:

1. Going back to the Keycloak Admin Console window, which should still have the user open, click on **Attributes**.
2. In the table that is displayed, set the key to `myattribute` and the value to `myvalue`, then click on **Save**. You have now added

a custom attribute to the user, but this is still not available to the application.

3. We will now create what is called a **client scope**. A client scope allows creating re-usable groups of claims that are added to tokens issued to a client. In the menu on the left-hand side, click on **Client Scopes**, then click on **Create client scope**. For the name in the form, enter `myscope`. Leave everything else as is and click **Save**.
4. Now we'll add the custom attribute to the client scope by creating a mapper. Click on **Mappers**, then click on **Configure a new mapper**. Then select **User Attribute**.

Fill in the form with the following values:

- **Name:** `myattribute`
- **User Attribute:** `myattribute`
- **Token Claim Name:** `myattribute`
- **Claim JSON Type:** `String`

5. Make sure **Add to ID Token** is turned on, then click on **Save**. Next, we will add your newly created client scope to the client.
6. In the menu on the left-hand side, click on **Clients** and locate the `oidc-playground` application. Select **Client Scopes**; then select **Add client scope window**, select `myscope` and click on **Add**, and select **Optional**.

As we added this scope to the optional client scopes for the client, it means that the client has to explicitly request this scope. If you had added it to the default client scopes, it would have always been added for the client.

We're doing this as we want to show how a client can request different information from Keycloak using the `scope` parameter. This allows the client to only request the information it needs at any given time, which is especially useful when asking users for additional consent to use their data as well as to avoid unnecessary claims in tokens.

You'll learn more about that in the next chapter.

7. Now go back to the playground application and again click on the **Send Refresh Request** button. You will notice that your custom attribute has not been added to the ID token.

If you get an error when refreshing the token, it is probably because your **single sign-on (SSO)** session with Keycloak has expired. By default, an SSO session expires if there is no activity for 10 minutes. Later in the book, we will look at how to change this.

Now let's send a new authentication request, but this time we'll include the `myscope` scope. In the playground application, click on **2 – Authentication**. In the `scope` field, set the value to `openid myscope`. Make sure you leave `openid` in there because, otherwise, Keycloak will not send you an ID token. Now go through these steps again to obtain a new token:

8. Click on **Generate Authentication Request**.
9. Click on **Send Authentication Request**.

10. Click on **3 – Token**.

11. Click on **Send Token Request**.

In the payload for the ID token, you will now notice the custom claim that you just added to the client.

Now that you have added a custom attribute, let's add roles to the ID token.

Adding roles to the ID token

By default, roles are not added to the ID token. You can change this behavior by going to **Client Scopes**, then selecting the **roles** client scope. Click on **Mappers**, then select **realm roles**. Turn on **Add to ID Token**, and click **Save**.

Assuming that the user you are using was the user you created during *Chapter 1, Getting Started with Keycloak*, the user should have a realm role associated with it. If it's a different user, make sure it does have a realm role associated with it.

Go back to the playground application and refresh the token again. You will now see `realm_access` within the ID token.

By default, all roles are added to all clients. This is not ideal as you want to limit what access each individual client has.

This has less impact on the ID token as it is only used to authenticate the user to a specific client, while it has a bigger impact on the access token, which is used to access other services.

By now you should have a reasonably good understanding of how an application uses the ID token in order to authenticate the user,

as well as discover information about the user. If you want to experiment some more with client scopes, now would be a good time since the playground application will allow you to play with scopes and see the result in the ID token.

In the next section, we will take a look at a different way an application can discover information about the authenticated user.

Invoking the UserInfo endpoint

In addition to being able to find information about the authenticated user from the ID token, it is also possible to invoke the UserInfo endpoint with an access token obtained through an OIDC flow.

Let's try this out by opening the playground application. You may at this point have to send new authentication and token requests, as it may be that your SSO session has expired.

If you're a quick reader (or you obtained new tokens), then click on **5 – UserInfo**. Under **UserInfo Request**, you will see that the playground application is sending a request to the Keycloak UserInfo endpoint, including the access token in the authorization header.

The following screenshot from the playground application shows an example **UserInfo Request**:

UserInfo Request

```
http://localhost:8080/realm/myrealm/protocol/openid-connect/userinfo  
Authorization: Bearer eyJhbGciOiJSUzIlNiIsInR5cCIgOiAiSldUIiwia21kIiA6ICJNU1JhVmwwSFBkVFU4TmNKW
```

Figure 4.11: UserInfo request

Under **UserInfo Response** you will see the response Keycloak sent. You may notice that this does not have all the additional fields in the ID token, but rather is just a simple JSON response including only the user attributes.

The following screenshot from the playground application shows an example **UserInfo Response**:

UserInfo Response

```
{  
  "sub": "65588621-32e8-4655-8cf8-86fb8054822e",  
  "email_verified": true,  
  "name": "Stian Thorgersen",  
  "preferred_username": "st",  
  "given_name": "Stian",  
  "family_name": "Thorgersen",  
  "email": "st@localdomain.localhost",  
  "myattribute": "myvalue"  
}
```

Figure 4.12: UserInfo response

Just as you can configure what information Keycloak returns in the ID token through client scopes and protocol mappers, you can also configure what information is returned in the UserInfo endpoint. Further, you can control what information is returned to the client that is invoking the UserInfo endpoint, and not the client that obtained the access token. This means that if a single access token is sent to two separate resource servers, they may see different information in the UserInfo endpoint for the same access token.

Let's try to add some custom information to the UserInfo endpoint. This time, instead of using a client scope, we'll add a protocol mapper directly to the client. Open the Keycloak Admin Console, then under **Clients**, locate the `oidc-playground` client. Click on **Client scopes**, then select the `oidc-playground-dedicated` client scope. Click on **Configure a new mapper**, and select **Hardcoded claim**. Finally, fill in the form with the following values:

- **Name:** `myotherclaim`
- **Token Claim Name:** `myotherclaim`
- **Claim value:** `My Other Claim`
- **Claim JSON Type:** `String`

Make sure **Add to userinfo** is turned on then click on **Save**. Go back to the playground application and send a new UserInfo request using the **Send UserInfo Request** button. You will now see the additional claim `myotherclaim` in the response.

One thing to remember about the UserInfo endpoint is that it can only be invoked with an access token obtained through an OIDC flow. We can try this out by going to the playground application, then clicking on the **2 – Authentication** button.

In the `scope` field, remove `openid`. Then click on **Generate Authentication Request** and **Send Authentication Request**.

Now click on **3 – Token**, then on **Send Token Request**. You will notice now that in the **Token Response** there is no `id_token` value, which is why there is no ID token displayed in the **ID Token** section.

Now, if you go to **5 – UserInfo** and click on the **Send UserInfo Request** button, you will also notice that the **Userinfo Request** fails.

In the next section, we will take a look at how you can deal with users logging out.

Dealing with users logging out

Dealing with logout in an SSO experience can actually be a quite difficult task, especially if you want an instant logout of all applications a user is using.

Initiating the logout

A logout can, for example, be initiated by the user by clicking on a logout button in the application. When the logout button is clicked, the application would send a request to the OpenID Connect RP-Initiated logout.

The application redirects the user to the Keycloak End Session endpoint, which is registered in the OpenID Provider Metadata as `end_session_endpoint`. The endpoint takes the following parameters:

- `id_token_hint`: A previously issued ID token. This token is used by Keycloak to identify the client that is logging out, the user, as well as the session that the client wants to log out of.
- `post_logout_redirect_uri`: If the client wants Keycloak to redirect back to it after the logout, it can pass the URL to Keycloak. The client has to previously have registered the logout URL with Keycloak.
- `state`: This allows the client to maintain state between the logout request and the redirect. Keycloak simply passes this parameter when redirecting to the client.
- `ui_locales`: The client can use this parameter to hint to Keycloak what locale should be used for the login screen.

When Keycloak receives the logout request, it will notify other clients in the same session about the logout. Then it will invalidate the session, which effectively makes all tokens invalid.

Leveraging ID and access token expiration

The simplest and perhaps most robust mechanism for an application to discover if a logout has taken place is simply to leverage the fact that ID and access token usually have a short expiration. As Keycloak invalidates the session on logout, a refresh token can no longer be used to obtain new tokens.

This strategy has a downside that it may be a few minutes from the user having logged out until all applications are effectively logged out, but in many cases, this is more than sufficient.

This is also a good strategy for public clients. As they don't usually provide a service directly themselves, but rather leverage the access token to invoke other services, they will quickly realize the session is no longer valid.

In cases where tokens have a long validity, it is still good practice to invoke the Token Introspection endpoint to check token validity periodically, which we will look at in the next chapter.

Leveraging OIDC Session Management

Through OIDC Session Management, an application can discover if a session has been logged out without the need for any requests to Keycloak, or for Keycloak to send any requests to it.

This works by monitoring the state of a special session cookie that Keycloak manages. As the application is usually hosted on a different domain than Keycloak, it is not able to read this cookie directly. Instead, a hidden HTML `iframe` tag loads a special page with Keycloak that monitors the cookie value and sends an event to the application when it observes the session state has changed.

This is an effective strategy, especially if the application is currently open. If the application is not open, it does mean that the application would not observe the logout until it is next opened. If, for example, a workstation was compromised, there is also a chance that a malicious party could prevent the session `iframe` from doing its job, leaving the application session still open. However, this can be relatively easily mitigated. One option is to only keep the application session open while the application is open. The Keycloak JavaScript adapter does exactly this by only storing tokens

in the window state. It is also, of course, mitigated by having a short expiration time for tokens.

The OIDC Session Management approach is sadly becoming less relevant today, as many browsers have started blocking access to third-party content. This effectively means the hidden session `iframe` is no longer able to access the session cookie in some browsers.

As such it is not a good idea to leverage this approach in new applications, and you will most likely want to migrate away from this approach in applications that are already using this approach.

Leveraging OIDC Back-Channel Logout

Through OIDC Back-Channel Logout, an application can register an endpoint to receive logout events.

When a logout is initiated with Keycloak, it will send a logout token to all applications in the session that have a back-channel logout endpoint registered.

The logout token is similar to an ID token, so it is a signed JWT. On receiving the logout token, the application verifies the signature and can now log out of the application session associated with the Keycloak session ID.

For server-side applications, using the back-channel logout is fairly effective. It does, however, become a bit complex to deal with for clustered applications with session stickiness. A common approach to scaling a stateful application is to distribute the application session among the instances of the application, and there is no

guarantee that the logout request from Keycloak is sent to the same application instance that is actually holding the application session. It is not trivial to configure a load balancer to route the logout request to the correct session, so this is usually something that has to be dealt with at the application level.

For stateless server-side applications, a logout request is also hard to handle, as usually the session is stored in a cookie in this case. In this case, the application would have to remember the logout request until the next time a request is made to the application for the given session, or the application session expires.

A note on OIDC Front-Channel Logout

The OpenID Connect Front-Channel Logout renders a hidden `iframe` for each application that has registered a front-channel logout endpoint on a logout page at the OpenID Provider. This, in theory, would be a nice way to log out of stateless server-side applications, as well as client-side applications. However, in practice, it can be unreliable. There is no effective way for the OpenID Provider to discover that the application was successfully logged out, so using this approach is a bit hit-and-miss.

In addition, the OIDC Front-Channel logout approach also suffers from browsers blocking third-party content, which means that when the OpenID Provider opens the logout endpoint in an `iframe`, there is no access to any application-level cookies, leaving the application unable to access the current authentication session.

How should you deal with logout?

In summary, the simplest approach is simply to rely on relatively short application sessions and token expiration. As Keycloak will keep the user logged in, it is possible to effectively use short application sessions without requiring users to frequently re-authenticate.

In other cases, or where logout has to be instant, you should leverage OIDC Back-Channel Logout.

Summary

In this chapter, you experienced first-hand the interactions in an OIDC authentication flow. You learned how the application prepares an authentication request and then redirects the user-agent to the Keycloak authorization endpoint for authentication. Then you learned how the application obtains an authorization code, which it exchanges for an ID token. By inspecting the ID token, you then learned how an application can find out information about the authenticated users. You also learned how to leverage client scopes and protocol mappers in Keycloak to add additional information about users. Finally, you learned how to deal with not only single sign-on, but also single sign-out.

You should now have a basic understanding of OpenID Connect and how it can be used to secure your own applications. We will build on this knowledge later in the book to get you ready to start securing all your applications with Keycloak.

In the next chapter, you will get a deeper understanding of OAuth 2.0, with a practical guide on how you can use Keycloak to use this standard in your applications.

Questions

1. How does the OpenID Connect Discovery specification make it easier for you to switch between different OpenID Providers?
2. How does an application discover information about the authenticated user?
3. How do you add additional information about the authenticated user?

Further reading

Refer to the following links for more information on topics covered in this chapter:

- OpenID Connect Core specification:
https://openid.net/specs/openid-connect-core-1_0.html
- OpenID Connect Discovery specification:
https://openid.net/specs/openid-connect-discovery-1_0.html
- OpenID Connect Session Management specification:
https://openid.net/specs/openid-connect-session-1_0.html
- OpenID Connect Back-Channel Logout specification:
<https://openid.net/specs/openid-connect->

[backchannel-1_0.html](#)

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/SecNet>



5

Authorizing Access with OAuth 2.0

In this chapter, you will get a deeper understanding of how Keycloak enables you to authorize access to REST APIs and other services by leveraging the OAuth 2.0 standard. Through using a sample application that was written for this book, you will see first-hand the interaction between an application and Keycloak to retrieve an access token that can be used to securely invoke a service.

We will start by getting the playground application up and running, before using the playground application to obtain a token from Keycloak that can be used to securely invoke a REST API. Then, we'll build on this knowledge to look at obtaining consent from a user before granting access to the application, as well as how to limit the access provided to the application. Finally, we'll look at how a REST API validates a token to verify whether access should be granted.

By the end of this chapter, you will have a good understanding of OAuth 2.0, including how to obtain an access token, understanding the access token, and how to use the access token to securely invoke a service.

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

- Running the OAuth 2.0 playground
- Obtaining an access token
- Requiring user consent
- Limiting the access granted to access tokens
- Validating access tokens

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 the GitHub repository associated with the book checked out locally. The GitHub repository is available at <https://github.com/PacktPublishing/Keycloak---Identity-and-Access-Management-for-Modern-Applications-2nd-Edition>.

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

<https://packt.link/fgdzB>

Running the OAuth 2.0 playground

The **OAuth 2.0** playground was developed specifically for this book in order to make it as easy as possible for you to understand and experiment with OAuth 2.0 in a practical way.

It does not use any libraries for OAuth 2.0, but rather all OAuth 2.0 requests are crafted by the application itself. One thing to note here is that the example application provided for this chapter does not implement OAuth 2.0 in a secure way, and ignores optional parameters in the requests that are important for a production application. There are two reasons for this. Firstly, this is done so that you can focus on understanding the general concepts of OAuth 2.0. Secondly, if you decide to implement your own libraries for OAuth 2.0, you should have a very good understanding of the specifications, and it is beyond the scope of this book to cover OAuth 2.0 in that much detail.

Before continuing with reading this chapter, you should start the OAuth 2.0 playground application, as it will be used throughout the rest of the chapter.

There are two parts to the playground application: a frontend application and a backend application.

To run the playground application, open a terminal and run the following commands to start the frontend part:

```
$ cd Keycloak---Identityand-Access-Management-for-Me
$ npm install
$ npm start
```

Then, in a new terminal window, run the following commands to start the backend part:

```
$ cd Keycloak---Identity-and-Access-Management-for-Me
$ npm install
$ npm start
```

To verify that the application is running, open <http://localhost:8000/> in your browser. The following screenshot shows the OpenID Connect Playground application:

OAuth 2.0 Playground

[1 - Discovery](#) [2 - Authorization](#) [3 - Invoke Service](#) [Reset](#)

Discovery

Issuer

<http://localhost:8080/realm/myrealm>

[Load OAuth 2.0 Provider Configuration](#)

OAuth 2.0 Provider Configuration

Figure 5.1: The OAuth 2.0 playground application

In order to be able to use the playground application, you need Keycloak to be running, as well as to have a realm with a user with the `myrole` global role and a client with the following configuration:

- **Client type:** OpenID Connect
- **Client ID:** oauth-playground
- **Client authentication:** Off
- **Valid Redirect URIs:** http://localhost:8000/
- **Web Origins:** http://localhost:8000

If you are unsure about how to do this, you should refer to *Chapter 1, Getting Started with Keycloak*, and *Chapter 2, Securing Your First Application*.

In the next section, we will start to take a deeper look into OAuth 2.0 by leveraging the playground application.

Obtaining an access token

The most common way to obtain an **access token** that can be used to, for example, invoke a secure REST API is through the OAuth 2.0 Authorization Code grant type.

In summary, to obtain an access token, an application redirects to Keycloak, which authenticates the user and optionally prompts the user to grant the application access or not, before returning an access token to the application. The application can then include the access token in the requests it sends to the REST API, allowing the REST API to verify whether access should be provided.

In the following diagram, the Authorization Code grant type is shown in more detail:

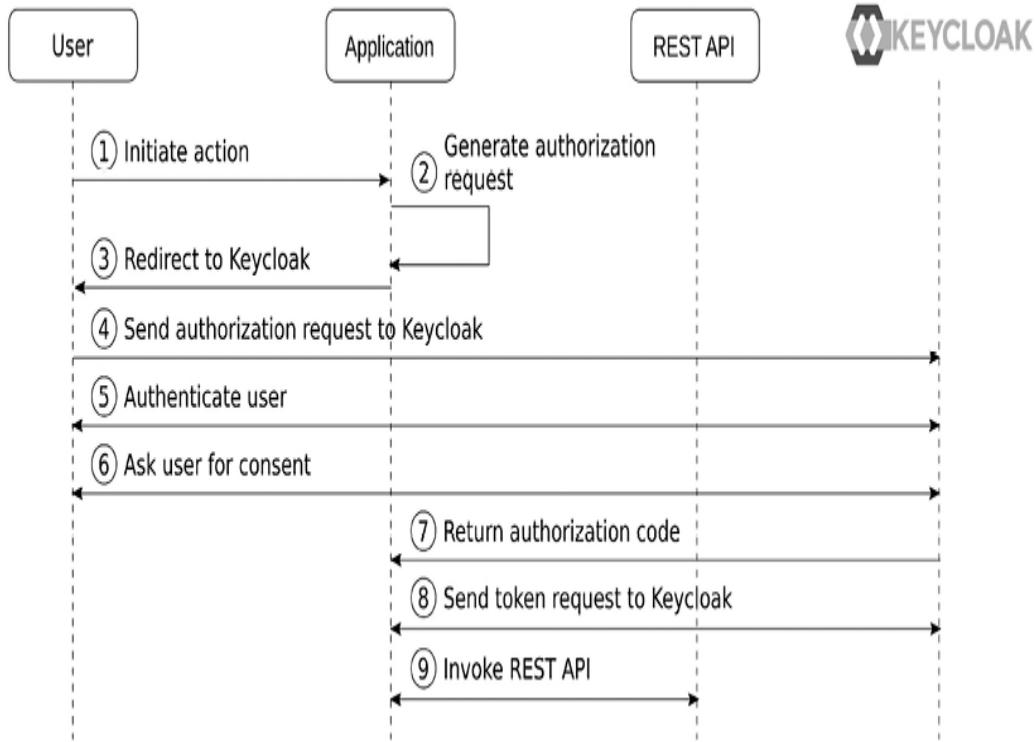


Figure 5.2: The Authorization Code grant type

The steps in the diagram in more detail are as follows:

1. The **user** performs an action that requires sending a request to an external REST API.
2. The **application** generates an **authorization request**.
3. The authorization request is sent to the **user** agent in the form of a 302 redirect, instructing the user agent to redirect to the authorization endpoint provided by Keycloak.
4. The user agent opens the **authorization endpoint** with the query parameters specified by the **application** in the **authorization request**.
5. If the **user** is not already **authenticated** with Keycloak, a login page is displayed to the user.

6. If the **application** requires consent to access the REST API, a consent page is displayed to the user asking whether the user wants to provide access to the application.
7. Keycloak returns an **authorization code** to the **application**.
8. The **application** exchanges the authorization code for an access token, as well as a refresh token.
9. The **application** can now use the access token to invoke the REST API.

Let's give this a go with the OAuth 2.0 Playground application. Open the playground application at <http://localhost:8000>. First, you need to load the OAuth 2.0 provider configuration by clicking on the button labeled **Load OAuth 2.0 Provider Configuration**. After you've done this, click on the button labeled **2 - Authorization**. You can leave the **client_id** and **scope** values as they are, then click on the button labeled **Send Authorization Request**.

You will be redirected to the Keycloak login page. Log in with the user you created in the first chapter. After you have logged in and have been redirected back to the playground application, the access token is displayed in the **Access Token** section. As Keycloak uses **JSON Web Token (JWT)** for its default token format, the playground application is able to directly parse and view the contents of the access token.

In the OpenID Connect Playground application that you experimented with in the previous chapter, you generated an authentication request, received an authorization code, then manually exchanged the authorization code for an ID token. As you've already experimented with this part, and it is completely the

same flow for an OAuth 2.0 Authorization Code grant type, this flow has been simplified to a single step in the OAuth 2.0 Playground application.

The following screenshot shows an example access token from the playground application:

Access Token

Header

```
{  
  "alg": "RS256",  
  "typ": "JWT",  
  "kid": "JI0_xxUiSi7oUkx0xLQe8xsdRTnK0irURnBuik6EyyQ"  
}
```

Payload

```
{  
  "exp": 1667145073,  
  "iat": 1667144773,  
  "auth_time": 1667144773,  
  "jti": "dc6caecb-7f4e-49c6-a9ed-bf2fd5af0db0",  
  "iss": "http://localhost:8080/realms/myrealm",  
  "aud": "account",  
  "sub": "f89a8358-de60-4238-883b-9143af28c56b",  
  "typ": "Bearer",  
  "azp": "oauth-playground",  
  "session_state": "567f4493-18dc-4f85-853c-4d3490f67f8f",  
  "acr": "1",  
  "allowed-origins": [  
    "http://localhost:8000"  
,  
  "realm_access": {  
    "roles": [  
      "default-roles-myrealm",  
      "offline_access",  
      "uma_authorization",  
      "myrole"  
    ]  
  },  
  "resource_access": {  
    "account": {  
      "roles": [  
        "manage-account",  
        "manage-account-links",  
        "view-profile"  
      ]  
    }  
  },  
  "scope": "profile email",  
  "sid": "567f4493-18dc-4f85-853c-4d3490f67f8f",  
  "email_verified": true,  
  "name": "Stian Thorgersen",  
  "preferred_username": "st",  
  "given_name": "Stian",  
  "family_name": "Thorgersen",  
  "email": "st@localhost.localdom"  
}
```

Figure 5.3: Example access token displayed in the playground application

Let's take a look at some of the values within the access token:

- `aud`: This is a list of services that this token is intended to be sent to.
- `realm_access`: This is a list of global roles the token provides access to. It is a union of the roles a user has been granted and the roles an application is allowed to access.
- `resource_access`: This is a list of client roles the token provides access to.
- `scope`: This is the scope included in the access token.

Now that the playground application has obtained an access token, let's try to invoke the REST API. Click on the button labeled **3 - Invoke Service**, then click on **Invoke**. You should now see a response that says **Secret message!**, as shown in the following screenshot:

OAuth 2.0 Playground

1 - Discovery

2 - Authorization

3 - Invoke Service

Reset

Invoke Service

Invoke

Response

Secret message!

Figure 5.4: Successful response from the playground application backend

You should now have a good understanding of how OAuth 2.0 can be leveraged to issue an access token to an application that allows the application to access resources on behalf of users.

In the next section, we will take a look at how the user can consent to grant the application access.

Requiring user consent

When an application wants access to a third-party service on behalf of a user, the user will usually be asked whether they want to grant access to the application. Without this step, a user would not know what kind of access the application is getting, and if the user is

already authenticated with the authorization server, the user may not even observe the application getting access.

In Keycloak, applications can be configured to either require consent or not require consent. For an external application, you should always require consent, but for an application you know and trust, you may choose to not require consent, which in essence means that you as an admin are trusting the application and are granting it access on behalf of users.

To try this out yourself, open the Keycloak admin console and navigate to the **oauth-playground** client. Then, under the **Login settings** for the client, turn on the **Consent required** option, as shown in the following screenshot:

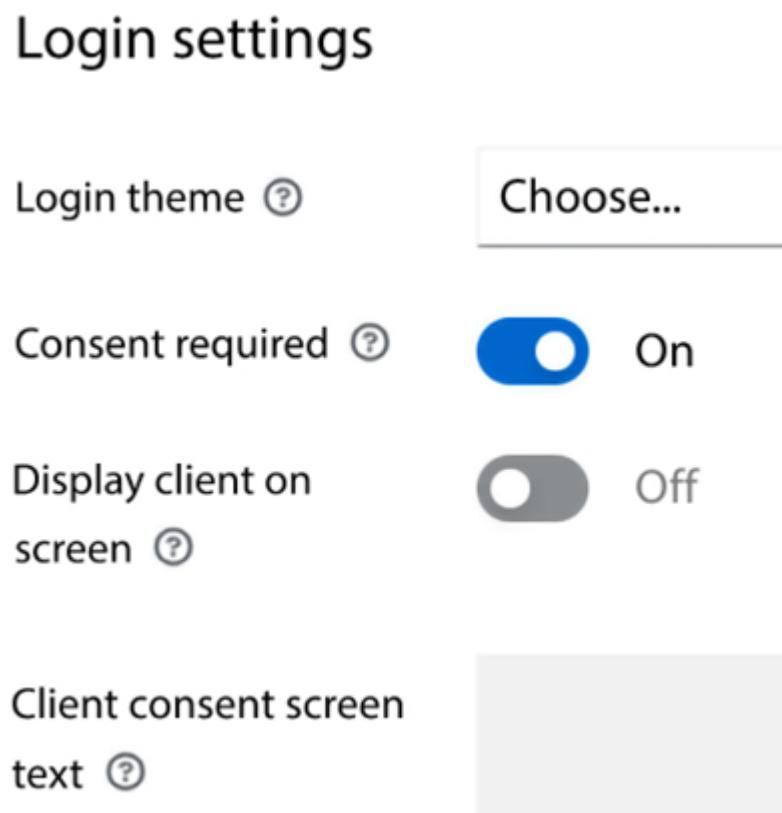


Figure 5.5: Requiring consent for a client

Once you have this enabled, go back to the playground application and obtain a new token by clicking on the button labeled **2 - Authorization**, followed by the button labeled **Send Authorization Request**. You should now see a similar screen to what is displayed in the following screenshot:

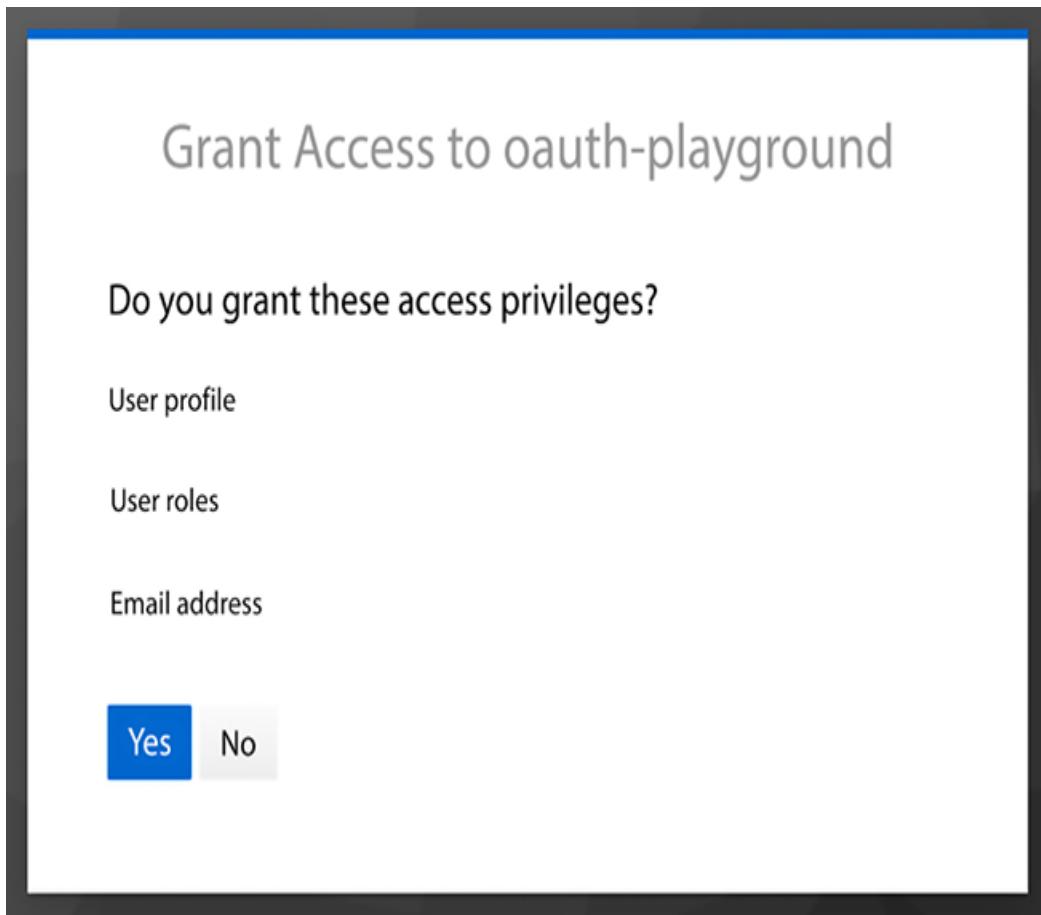


Figure 5.6: Granting access

This screen lists all scopes the application is requested to access on behalf of the user, which is a core concept of OAuth 2.0, making it possible for an application to be granted limited access to a user's account.

One interesting aspect of scopes is that an application can initially ask for limited access. As the user is starting to use more features within the application, the application can ask for additional access as needed. Doing this may be less intimidating to the user as it is clearer why your application is asking for that access.

Let's give this a go by creating a new client scope and requesting this additional scope in the playground application.

Go back to the Keycloak admin console and click on **Client scopes** in the menu on the left-hand side. Then, click on **Create client scope**. Fill in the form with the following values:

- **Name:** albums
- **Display On Consent Screen:** ON
- **Consent Screen Text:** View your photo albums

The following screenshot shows the client scope that you should create:

Create client scope

Name ? albums

Description

Type ? None

Protocol ? OpenID Connect

Display on consent screen ? On

Consent screen text ? View your photo albums

Include in token scope ? On

Display Order

[Save](#) [Cancel](#)

Figure 5.7: Creating a client scope

You can leave the other values as is, then click on **Save**. Now, navigate to the **oauth-playground** client again, then click on **Client scopes**. Click on **Add client scope**, select **albums**, and click on **Add** followed by **Optional**.

Now, return to the playground application again, then click on the button labeled **2 - Authorization**. In the **scope** field, enter **albums**, then click on **Send Authorization Request**. This time, you should be prompted to grant access to view photo albums, as shown in the following screenshot:

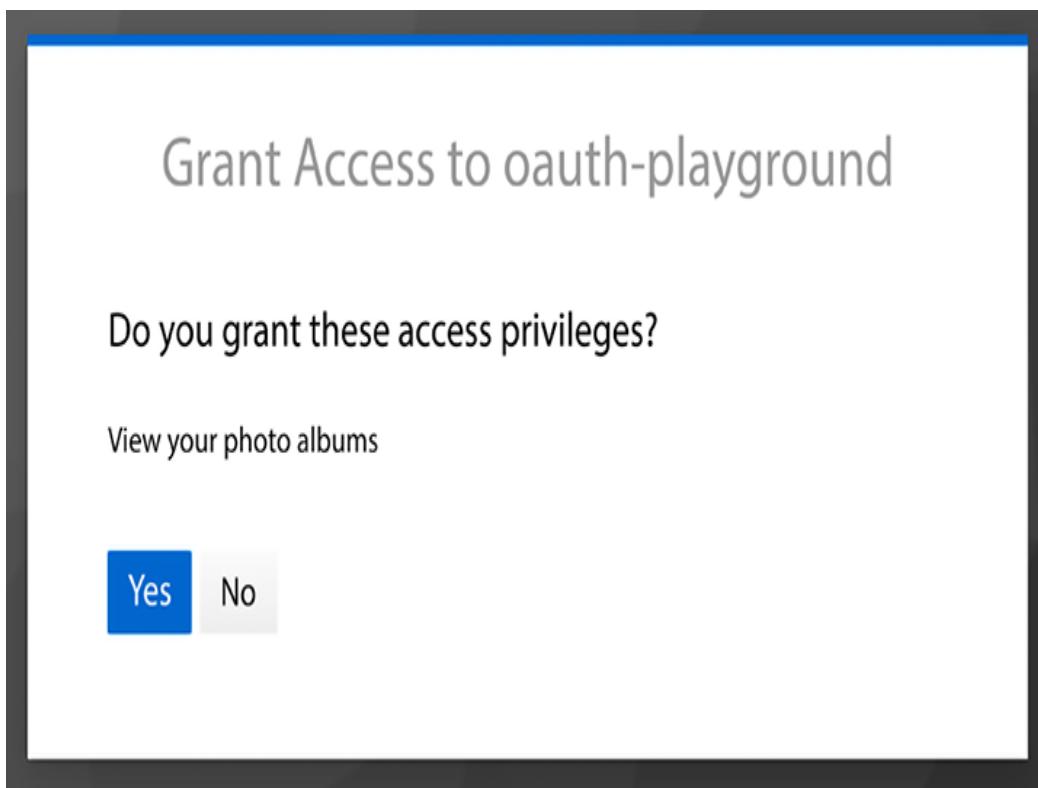


Figure 5.8: Granting access to photo albums

Keycloak remembers what consent a user has given to a particular application, which means the next time the application asks for the same scope, the user will not be prompted again.

Through the account console, a user can remove access to an application if they wish. You can try this out by going to the account console, navigating to **Applications**, then revoking the access for the **oauth-playground** application. The next time you try to obtain an access token again through the playground application, Keycloak will again ask you to provide access to **oauth-playground**.

You should now have a good understanding of how an admin can grant access to trusted applications on behalf of users, as well as how third-party applications can be required to ask the user for consent prior to getting access.

In the next section, we will look at strategies for scoping access tokens, which in essence means controlling what access a token provides to the application.

Limiting the access granted to access tokens

As access tokens get passed around from the application to services, it is important to limit the access granted. Otherwise, any access token could potentially be used to access any resource the user has access to.

There are a few different strategies that can be used to limit access for a specific access token. These include the following:

- **Audience:** Allows listing the resource providers that should accept an access token.
- **Roles:** By controlling what roles a client has access to, it is possible to control what roles an application can access on behalf of the user.
- **Scope:** In Keycloak, scopes are created through client scopes, and an application can only have access to a specific list of scopes. Furthermore, when applications require consent, the user must also grant access to the scope.

Let's go through these one at a time and see exactly how this can be done with Keycloak, starting with the audience.

Using the audience to limit token access

At the moment, access tokens issued to the frontend part of the playground application do not actually include the backend in the audience. The reason this works is that the backend part has not been configured to check the audience in the token.

Let's start with configuring the backend to check the audience. Stop the backend part, then open the `Keycloak---Identityand-Access-Management-for-Modern-Applications-2nd-Edition/ch5/backend/keycloak.json` file in a text editor.

Change the value of the `verify-token-audience` field to `true`, as shown in the following screenshot:

```
1{
2  "realm": "myrealm",
3  "bearer-only": true,
4  "auth-server-url": "${env.KC_URL:http://localhost:8080/auth}",
5  "resource": "oauth-backend",
6  "verify-token-audience": true
7}
```

Figure 5.9: Enabling verifying the token audience for the backend

One thing to notice in this file is the `resource` field, which is the value the backend will look for in the `aud` field to know whether it should accept the token.

Start the backend part again. Once started, go back to the playground application and obtain a new access token. If you look at the values for the access token, you will see the `aud` field, and you will also notice that `oauth-backend` is not included.

If you now try to invoke the service through the playground application, you will get a response telling you that access was denied. The backend is now rejecting the access token.

In Keycloak, there are two different ways to include a client in the audience. It can be done manually by adding a specific client to the audience with a protocol mapper (added directly to the client, or through a client scope), or it can be done automatically if a client has a scope on another client role from another client.

Let's try to add the audience manually with a protocol mapper. Open the Keycloak admin console and navigate to **Clients**. Create a new client with the **Client ID** value set as `oauth-backend`.

As this client will be used by the backend, it won't be used to obtain tokens, which means you can disable **Standard flow** and **Direct access grant** when creating the client.

Now go back to the client list and open the **oauth-playground** client. Click on **Client scopes**, then select the **oauth-playground-dedicate** client scope from the list. Click on **Configure a new mapper**, then select **Audience**. Fill in the form with the following values:

- **Name:** **backend audience**
- **Included Client Audience:** **oauth-backend**

Then, You then need to select **Add to access token**. Finally, click **Save**. Go back to the playground application and obtain a new access token. Now **oauth-backend** is included in the **aud** field, and if you again try to invoke the service through the playground application, you will get a successful response.

When looking at the **aud** field of the access token, you may have noticed that **account** was included. The reason this is included is that, by default, a client has a scope on all roles, and by default, a user has a few client roles for the account client that provide the user access to the Keycloak account console. In the next section, we will take a closer look at how roles work.

Using roles to limit token access

Keycloak has built-in support for roles, which can be used to grant users permissions. Roles are also a very useful tool to limit the permissions for an application as you can configure what roles are included in the access token for a given application.

A user has role mappings on a number of roles granting the user the permissions that the role provides. A client, on the other hand, does not have roles assigned directly to it but instead has a scope on a set of roles, which controls what roles can be included in the tokens sent to the client.

This means that the roles included in tokens are the intersection between the roles a user has and the roles a client is allowed to use, as shown in the following diagram:

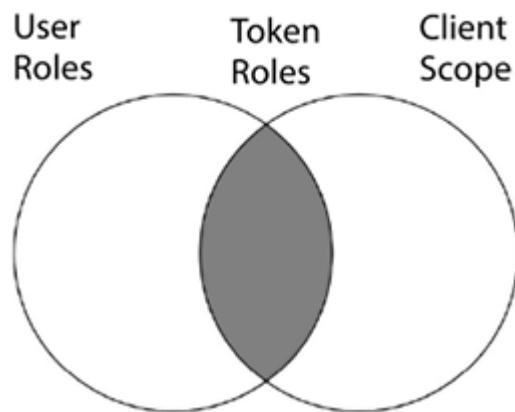


Figure 5.10: Roles included in tokens

Let's try this out in the playground application. Before making any changes, obtain a new access token and take a look at the `aud`, `realm_access`, and `resource_access` claims. The following shows an example access token with all non-relevant claims removed:

```
{  
  "aud": [  
    "oauth-backend",  
    "account"  
,  
  "realm_access": {  
    "roles": [  
      "default-roles-myrealm",  
      "offline_access",  
      "uma_authorization",  
      "myrole"  
    ]  
  },  
  "resource_access": {  
    "account": {  
      "roles": [  
        "manage-account",  
        "manage-account-links",  
        "view-profile"  
      ]  
    }  
  }  
}
```

Within the `aud` claim, you can see two clients. The `oauth-backend` client is included as we explicitly included this client in the audience in the previous section. The `account` client, on the other hand, is included as the token includes roles for the account client, which by default results in the client automatically being added to the audience of the token, as we can assume that if the token

includes roles specifically for a client, the token is intended to be used to access this client.

You can also see that the token includes all roles granted to the user. By default, all roles for a given user are included in the token. This is for convenience when getting started with Keycloak and you should not include all roles in a production scenario.

Now, let's try to limit the role scope for the `oauth-playground` client to limit what is included in the token. Open the Keycloak admin console and navigate to the `oauth-playground` client. Then, click on the tab labeled **Client scopes**, select the `oauth-playground-dedicated` client scope, and click on the tab labeled **Scope**. You will notice that it has the **Full Scope Allowed** option turned on. This is the feature that, by default, includes all roles for a user in the tokens sent to this client.

Turn off **Full Scope Allowed**, then return to the playground application and obtain a new access token. In the new access token, you will notice that there are no longer any roles in the token and the `aud` claim now only includes the `oauth-backend` client. If you now try to invoke the service with this token, you will get an access denied message. This is because the service only permits requests that include the `myrole` role.

Go back to the Keycloak admin console and again open the **scope** tab for the `oauth-playground-dedicated` client scope for the `oauth-backend` client. Select **Assign role**, select the `myrole` role, and click on **Assign**. Return to the playground application and obtain a new access token, and you will now see that the `myrole`

role is included in the `realm_access` claim, as shown in the following access token snippet:

```
{  
  "aud": "oauth-backend",  
  "realm_access": {  
    "roles": [  
      "myrole"  
    ]  
  }  
}
```

It is also possible to add scope through a client scope that is attached to a client. This may be a bit confusing as the term **scope** is somewhat overused within Keycloak. The following list tries to clarify this potential confusion:

- **A client can access one or more client scopes:** This is configured through the **client scopes** tab for a client.
- **A client scope can have a scope on roles:** When a client has access to a client scope that in turn has a scope on roles, the client has a scope on the roles that the client scope has.

As this may still be a bit confusing, let's experiment a bit with this in practice by leveraging the playground application.

Before continuing, you should first remove the scope that the `oauth-playground` client has on the `myrole` role. To do this, go back to the Keycloak admin console and again open the **scope** tab for the `oauth-playground-dedicated` client scope for the `oauth-`

`playground` client. Then, select the `myrole` role from the **Assigned Roles** section and click on **Unassign**.

Now the tokens sent to the `oauth-playground` client no longer include the `myrole` role, which is exactly what we want as we will now add this through a client scope instead of directly to the client.

Open the Keycloak admin console and go to **Client scopes**. Click on **Create client scope** to create a new client scope. For the name, enter `myrole` and leave everything else as is, then click on **Save**. Now, select the tab labeled **Scope**. This is where you control what roles are included in the token when this client scope is included. Client on **Assign role**, select the `myrole` role, and click on **Assign**.

You have now created a client scope that has a scope on the `myrole` role. Next, let's add this client scope as an optional client scope to the `oauth-playground` client. Navigate to the `oauth-playground` client and click on **Client Scopes**. Click on **Add client scope**, select `myrole`, and click on **Add** followed by **Optional**.

As you added the `myrole` client scope as an optional client scope, it means the `myrole` role is only included in the token if the `oauth-playground` client explicitly requests the `myrole` scope.

Return to the playground application and obtain a new access token. You will see that the `myrole` role is not yet included in the `realm_access` claim. In fact, the `realm_access` claim should not be included at all since the client does not at this point have a scope on any global roles. In the **Scope** field, set the value to `myrole`, and click on **Send Authorization Request** to obtain a new access token that includes this scope. This results in the playground application

requesting the `myrole` scope, which in turn will add the `myrole` role to the token.

In the next section, we will take a look at how scopes on their own can be leveraged to limit the access granted by a token.

Using the scope to limit token access

The default mechanism in OAuth 2.0 to limit the permissions for an access token is through scopes directly. Using scopes is especially useful with third-party applications where users should consent to giving applications access to resources on their behalf.

Within Keycloak, a scope in OAuth 2.0 is mapped to a client scope. If you only want to have a scope that the application can request that is then used by a resource provider to provide limited access to resources, you can simply define an empty client scope that has no protocol mappers and doesn't have access to any roles.

While defining scopes, it is important to not go overboard and to limit the number of scopes you define, and consider how scope is represented to an end user, who should understand what giving permissions to that scope implies and not be confused by an application requesting a large number of scopes.

Scopes should usually also be unique within all applications in your organization, so you may want to prefix the scope with the name of the service, or even consider using the URL of the service as a prefix.

Here are some example scopes to give you an idea:

- `albums:view`

- `albums:create`
- `albums:delete`
- `https://api.acme.org/bombs/bombs.purchase`
- `https://api.acme.org/bombs/bombs.detonate`

There is no standard for defining scopes, so you need to define your own. What can be useful is looking at scopes defined by Google, GitHub, Twitter, and so on for inspiration. One thing to bear in mind in these cases is that GitHub and Twitter, in a way, have a dedicated authorization server for a single service, which means they do not have to worry as much about prefixing scopes with the service. On the other hand, Google uses the same authorization server for multiple services.

Here are some example scopes defined by Google:

- `https://www.googleapis.com/auth/gmail.compose`
- `https://www.googleapis.com/auth/photoslibrary.readonly`
- `https://www.googleapis.com/auth/calendar.events`

Here are some example scopes defined by GitHub:

- `repo`
- `write:org`
- `notifications`

Let's give this a go with the playground application by imagining that we have a photo album service that provides access to view albums, create albums, and delete albums. We'll also pretend that

the playground application offers functionality to view and manage photo albums.

Start by creating the following three client scopes through the Keycloak admin console:

- `albums:view`
- `albums:create`
- `albums:delete`

We have already covered how to create client scopes previously, but in summary, the steps you need to create a client scope are as follows:

1. Open **Client Scopes** in the Keycloak admin console.
2. Click on the button labeled **Create client scope**.
3. Enter the name from the preceding list, and enter some value for the **Consent Screen Text** field that describes to a user what permissions are given (for example, `View photo albums`).
4. Click on the button labeled **Save**.

After you have created the three client scopes, navigate to the `oauth-playground` client and click on **Client scopes**, followed by **Add client scope**. Select `albums:view` and click on **Add** followed by **Default**. Click on **Add client scope** again; this time, select `albums:create` and `albums:delete`, then click on **Add** followed by **Optional**.

We added the scope to view permissions as a default scope as we're assuming that the playground application always requires viewing albums. On the other hand, we set the ability to create and delete albums as optional. This is sometimes referred to as incremental

authorization, where the application requests additional permissions only when the user starts using a part of an application that requires the additional permissions. This approach makes it a lot more intuitive to the user why the application is requesting the permissions.

Before continuing, make sure the `oauth-playground` client requires consent by selecting the **Settings** tab and then checking that **Consent Required** is turned on.

Now, return to the playground application and remove any value in the **Scope** field before clicking on the **Send Authorization Request** button. Keycloak should now ask you to grant the `oauth-playground` application access to view photo albums, as shown in the following screenshot:

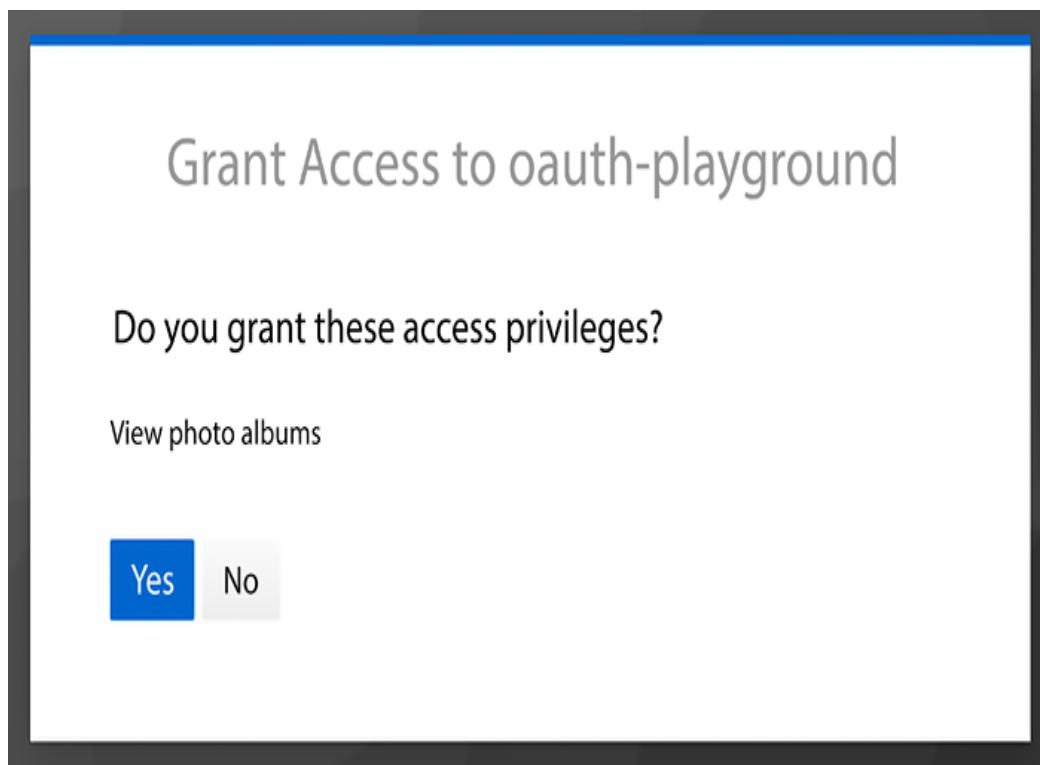


Figure 5.11: Granting `oauth-playground` access to view photo albums

After clicking on **Yes**, the access token displayed within the playground application will include **albums:view** in the scope claim. Let's now imagine that the user would like to create a new photo album through the playground application, so it must have access to also create albums. Set the value of the **scope** field to **albums:create** and click on **Send Authorization Request** again. This time, you will be prompted to grant access to create photo albums. After clicking **Yes**, this time, you will see that the scope claim in the access token now includes both **albums:view** and **albums:create**.

By now, you should have a good understanding of the different techniques for limiting the access provided by a given access token. In the next section, we will take a look at how an application can validate an access token.

Validating access tokens

You have two choices to validate an access token, either by invoking the token introspection endpoint provided by Keycloak or by directly verifying the token.

Using the token introspection endpoint is the simplest approach, and it also makes your applications less tied to Keycloak being the authorization server. OAuth 2.0 does not define a standard format for access tokens and these should be considered opaque to the application. Instead, it defines a standard token introspection endpoint that can be used to query the authorization server for the state of a token as well as claims associated with the token. This also enables tokens to not be self-contained, meaning that not all

relevant information about the token is encoded into the token, but rather the token only serves as a reference to the information.

One downside of using the token introspection endpoint is that it introduces extra latency in processing the request as well as additional load on the authorization server. A common technique here is to have a cache that remembers previously verified tokens preventing the service from re-validating already verified tokens within a configurable amount of time. The time between re-validating the token should be fairly short, usually in terms of a few minutes only.

You can try to invoke the token introspection endpoint by using `curl`, or any other tool that lets you send an HTTP request.

First, we need two things: the credentials for the `oauth-backend` client and an encoded access token.

To get the credentials for the `oauth-backend` client, go to the Keycloak admin console and navigate to the `oauth-backend` client. In the **Capability config** section, enable **Client authentication** and click on **Save**. Now, at the top of the page, click on the **Credentials** tab and copy the value of the **Client secret** field.

Open a terminal and set the secret to an environment variable, as shown in the following example:

```
$ export SECRET=b1e0073d-3f2b-4ea4-bec0-a35d1983d5b6
```

Keep this terminal open, then open the playground application and obtain a new access token. At the bottom of the field, in the

Encoded section, you will see the encoded access token. Copy this value, then set an environment variable in the terminal you opened previously, as shown in the following example:

```
$ export TOKEN=eyJhbGciOiJSUzI1NiIsInR5c...
```

Now, you can invoke the token introspection endpoint by running the following command in the same terminal:

```
$ curl --data "client_id=oauth-backend&client_secre
```

The endpoint will return a JSON document with the state of the token and associated information for the token, as shown in the following screenshot:

```
{  
    "exp": 1667147808,  
    "iat": 1667147508,  
    "auth_time": 1667144773,  
    "jti": "681209f3-0b87-4ef2-b68d-21d46f615fc5",  
    "iss": "http://localhost:8080/realm/myrealm",  
    "aud": "oauth-backend",  
    "sub": "f89a8358-de60-4238-883b-9143af28c56b",  
    "typ": "Bearer",  
    "azp": "oauth-playground",  
    "session_state": "567f4493-18dc-4f85-853c-4d3490f67f8f",  
    "name": "Stian Thorgersen",  
    "given_name": "Stian",  
    "family_name": "Thorgersen",  
    "preferred_username": "st",  
    "email": "st@localhost.localdom",  
    "email_verified": true,  
    "acr": "0",  
    "allowed-origins": [  
        "http://localhost:8000"  
    ],  
    "realm_access": {  
        "roles": [  
            "myrole"  
        ]  
    },  
    "scope": "myrole albums:create profile email albums:view",  
    "sid": "567f4493-18dc-4f85-853c-4d3490f67f8f",  
    "client_id": "oauth-playground",  
    "username": "st",  
    "active": true  
}
```

Figure 5.12: Token introspection endpoint response

The output in the example above has been formatted with `jq`, which is a nice utility for formatting and processing JSON.

The other approach to verifying access tokens issued by Keycloak is validating them directly in the application.

As Keycloak uses JWT as its access token format, this means you can parse and read the contents directly from your application, as well as verify that the token was issued by Keycloak, as it is signed by Keycloak using its private signing keys. One thing to note about this approach is the session is not validated, which means a user may have signed out, but the application will still consider the token valid. We're not going to go into detail on how to do this though, as there are quite a lot of mistakes you can make when verifying a token yourself, and you should have a very good understanding of JWT as well as the information within the token before you consider implementing this approach yourself.

All Keycloak client libraries, which are referred to as application adapters by Keycloak, verify tokens directly without the token introspection endpoint. There are also a number of good libraries available for different programming languages you can use. To give you an idea of how to verify an access token, you would need to do the following:

- Retrieve the public signing keys from the JWKS endpoint provided by Keycloak.
- Verify the signature of the access token.
- Verify that the access token has not expired.

- Verify the issuer, audience, and type of the token.
- Verify any other claims that your application cares about.

You should now have a good idea of how a service can verify a token, either by using the token introspection endpoint or by directly verifying the token if it is a JWT.

Summary

In this chapter, you experienced first-hand how to obtain access tokens using the OAuth 2.0 Authorization Code grant type. You learned how an admin can grant access to internal applications on behalf of a user, and how users themselves can grant access to third-party applications. You learned about various techniques for how you can limit the access provided by a specific access token. Finally, you learned how a service can directly read and understand the contents of an access token issued by Keycloak, as it is using JWT-based tokens. You also learned how the token introspection endpoint can be leveraged to validate and discover information about an access token in a more standard and portable way.

You should now have a basic understanding of OAuth 2.0 and how it can be used to secure your own applications. We will build on this knowledge later in the book to get you ready to start securing all your applications with Keycloak.

In the next chapter, you will learn about how to use Keycloak to secure a range of different application types.

Questions

1. How does an application invoke a protected REST API by leveraging OAuth 2.0?
2. What are the three different techniques you can use to limit the access provided by an access token?
3. How can a service validate an access token to decide whether permission should be granted?

Further reading

Refer to the following links for more information on the topics covered in this chapter:

- OAuth 2.0 Authorization Code grant specification:
<https://oauth.net/2/grant-types/authorization-code/>
- OAuth 2.0 token introspection specification:
<https://oauth.net/2/token-introspection/>
- OAuth scopes: <https://oauth.net/2/scope/>

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/SecNet>



6

Securing Different Application Types

In this chapter, we will begin by evaluating whether the application we want to secure is internal or external. 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/archive/master.zip
```

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://packt.link/teMUX>

Understanding internal and external applications

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

Internal applications, sometimes referred to as first-party applications, are applications owned by the company. 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 or 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 Keycloak administration interface. On the left, a sidebar menu has 'Clients' selected. The main content area is titled 'Login settings'. It contains two configuration items: 'Login theme' (set to 'Choose...') and 'Consent required' (set to 'Off', indicated by a blue circle around the radio button). To the right, a vertical sidebar lists navigation options: 'Jump to section', 'General Settings', 'Access settings' (which is currently selected and highlighted in blue), and 'Capability config'.

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 authenticate. 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 'Login settings' configuration page. On the left, there's a sidebar with a 'Master' dropdown and a 'Manage' section containing links for 'Clients', 'Client scopes', 'Realm roles', and 'Users'. The 'Clients' link is currently selected, indicated by a blue underline. The main content area is titled 'Login settings'. It contains three configuration items: 'Login theme' with a dropdown menu labeled 'Choose...', 'Consent required' with a radio button set to 'Off', and 'Display client on screen' with a radio button also set to 'Off'. To the right of the main content, there's a vertical sidebar with navigation links: 'Jump to section', 'General Settings', 'Access settings', and 'Capability config'. The 'Access settings' link is underlined in blue, indicating it's the active section.

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:

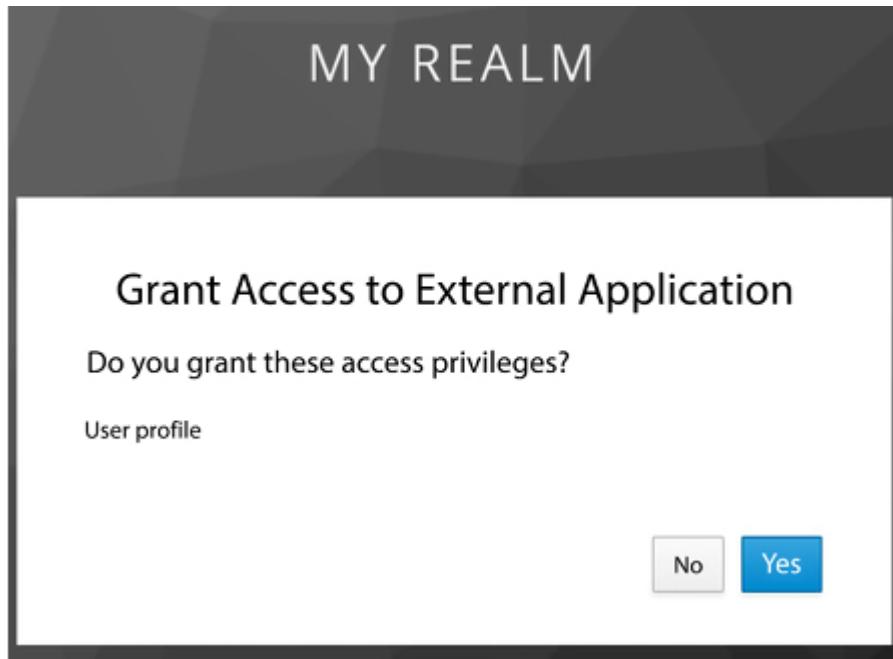


Figure 6.3: User granting access to an external application

You should now understand the difference between an internal and 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 exchange 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 they 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 used nowadays.

The following screenshot shows an example of the Google and Amazon login pages. You can see that they are not embedded in the applications themselves:

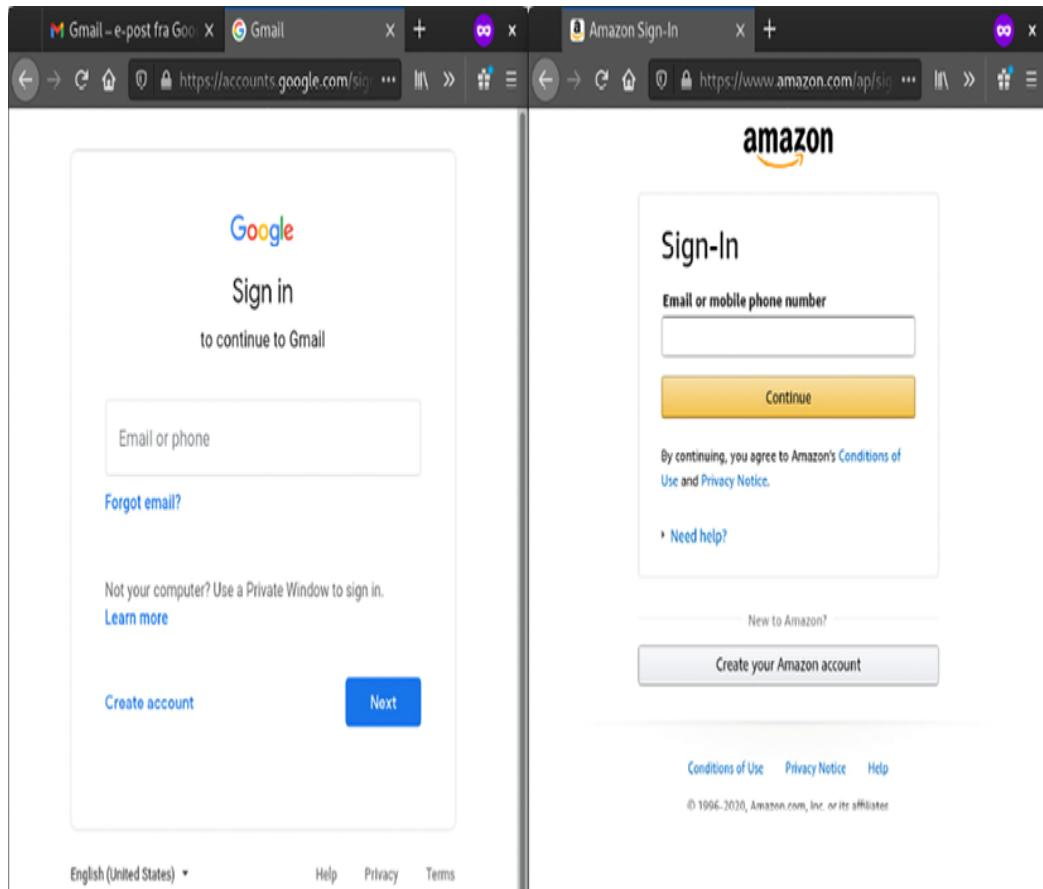


Figure 6.4: Example from Google and Amazon showing external login pages

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 of a server-side web application:

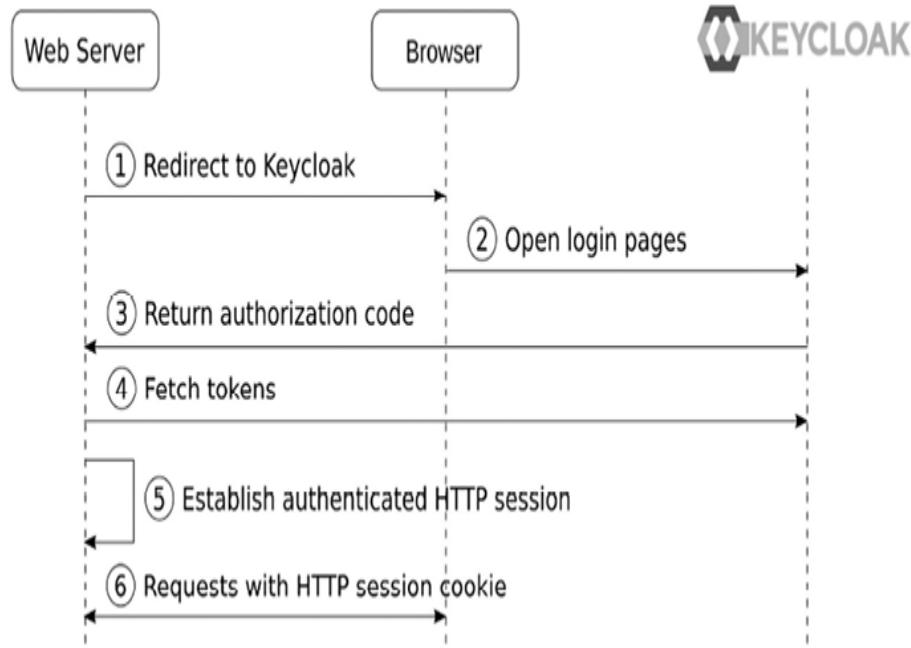


Figure 6.5: Server-side web application

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, and can directly parse and verify 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:

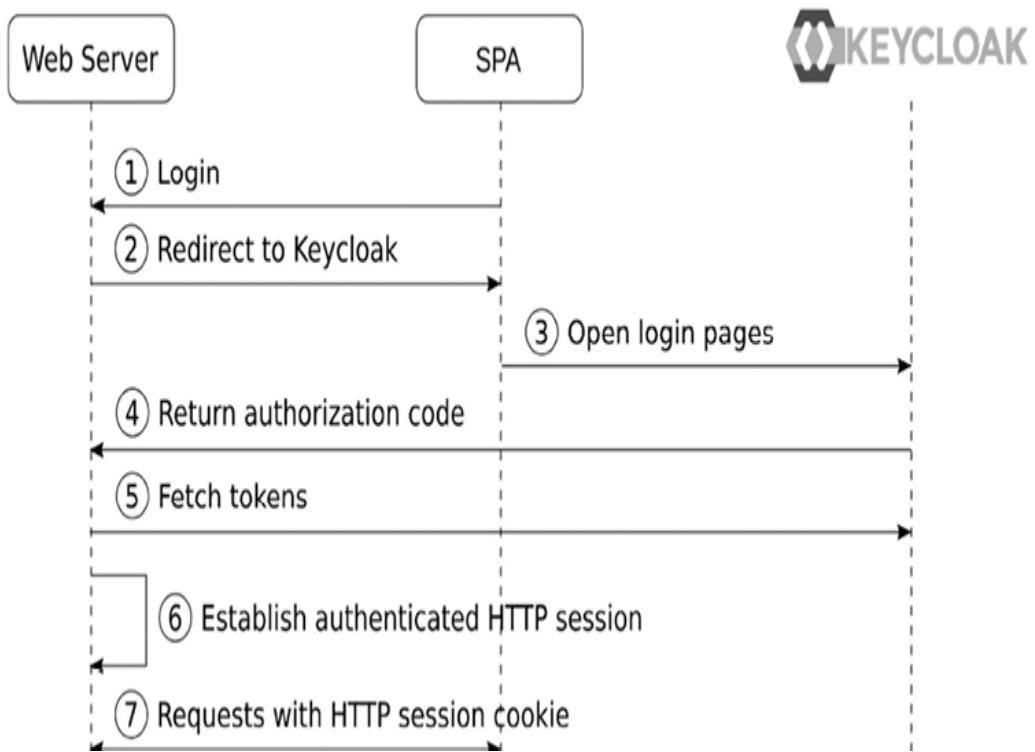


Figure 6.6: A SPA with a dedicated REST API

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. The application retrieves the ID token directly from Keycloak, and can directly parse and verify 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 (BFF)** 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:

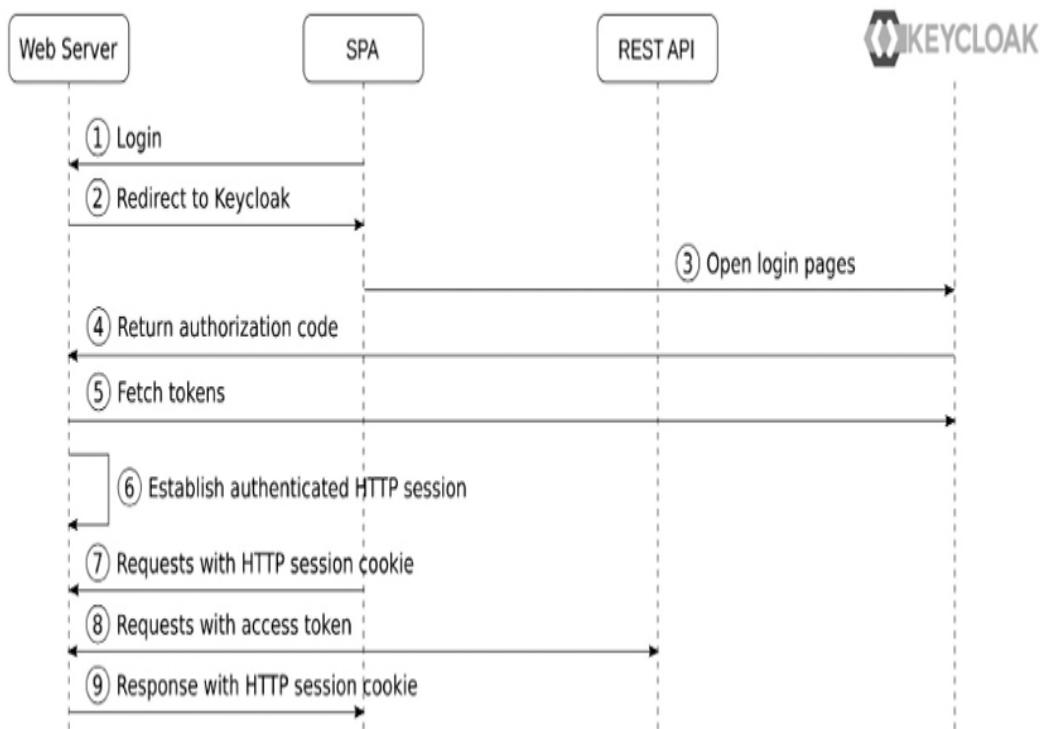


Figure 6.7: SPA with an intermediary REST API

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. The **Web Server** verifies the ID token directly from Keycloak, 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** include 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:

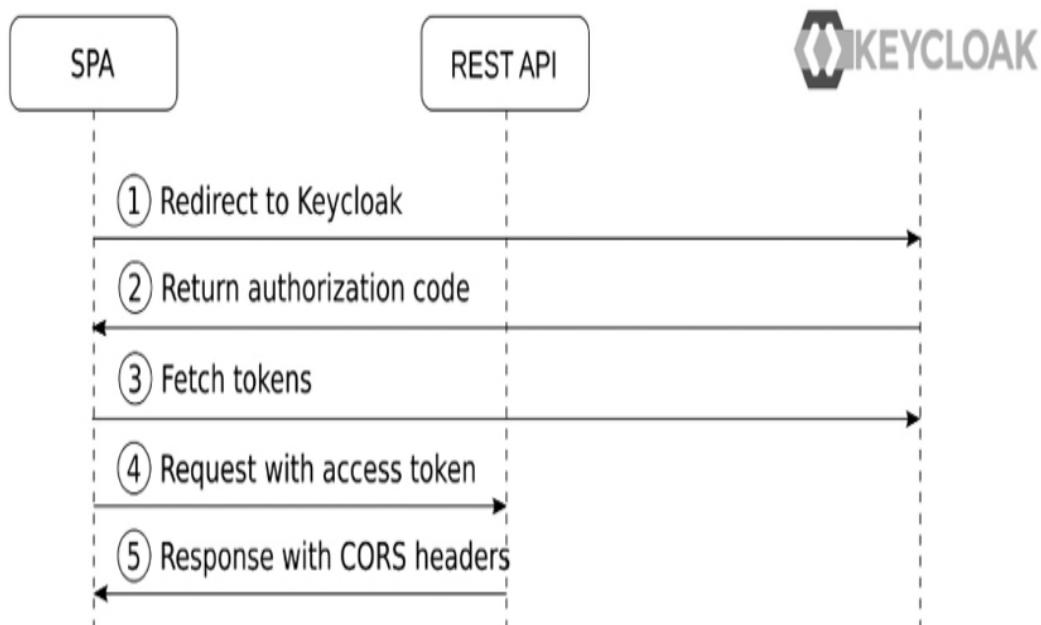


Figure 6.8: A SPA with an external REST API

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.

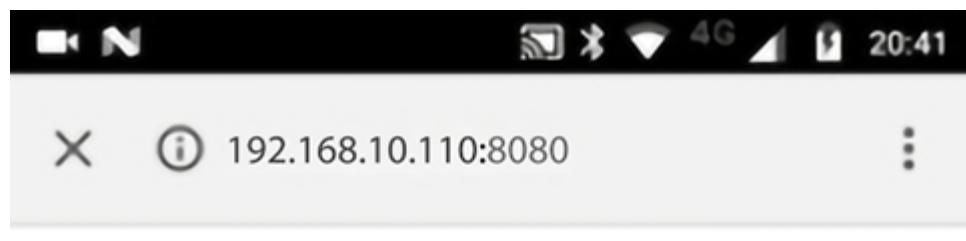
Effectively, this means that your native or mobile application must use a browser to authenticate with Keycloak. In this regard, there are three options available depending on the type of application:

- 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. Users that are concerned about this 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:



EXAMPLE

Log In

Username or email

User

Password

.....|

1 2 3 4 5 6 7 8 9 0

q w e r t y u i o p å

a s d f g h j k l ø æ

⌫ z x c v b n m ⌂

?123 , ⌂ Norsk bokmål . ⌂



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:

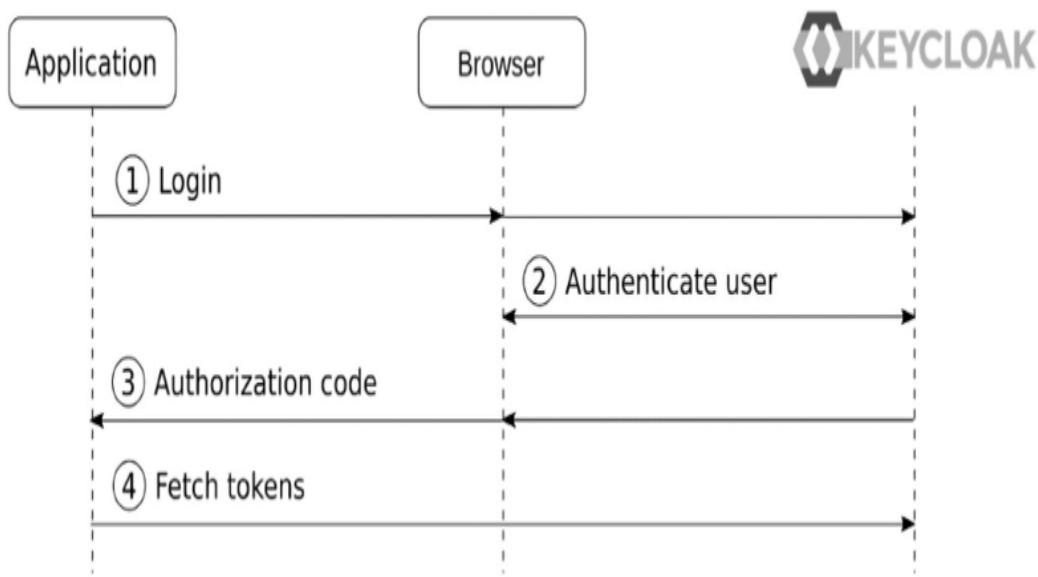


Figure 6.10: Native application

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 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
- **Client authentication:** Off
- **Standard flow:** Enabled
- **Valid Redirect URIs:** http://127.0.0.1/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-I  
$ 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.

Authenticating with an input-constrained device

In summary, the device code grant type works by the application showing a short code that a user enters into a special endpoint in 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 logged in already. After completion, the application is able to retrieve the authorization code from the authorization server.

Let's try this flow out by first registering a client with the following settings:

- **Client ID:** tv
- **Client authentication:** Off
- **OAuth 2.0 Device Authorization Grant:** Enabled

Then, you can initiate the flow by sending a Device Authorization Request by opening a terminal and running the following command:

```
$ curl --data "client_id=tv" http://localhost:8080/i
```

You will get back a JSON response with the following fields:

```
{
  "device_code": "X5o18WzJjsehNRQDzVP7SutFJIGJNdYoq",
  "user_code": "XPHS-PIDG",
  "verification_uri": "http://localhost:8080/realm",
  "verification_uri_complete": "http://localhost:8080/realm/device_code?state=XPHS-PIDG&code=X5o18WzJjsehNRQDzVP7SutFJIGJNdYoq",
  "expires_in": 600,
  "interval": 5
}
```

To continue the flow, open the URL from the `verification_uri` field in a browser. The interesting aspect of the device code flow is that this can be done on a different machine. For example, the flow can be initiated on a Smart TV, while the user can open the URL on a mobile phone.

After opening the `verification_url`, you will be asked to enter the code provided by your device. This code is included in the previous response in the `user_code` field. Enter this code into the form and click **Submit**. You will also be requested to grant access to the application after entering the code.

Once the user has authorized access to the device, you can continue the flow by sending a Device Access Token Request to obtain tokens. To do that, you need the `device_code` from the previous response. Set this to an environment variable by running the following command:

```
$ export CODE=<insert device_code from previous respon
```

After you have set this environment variable, you can send a Device Access Token Request with CURL:

```
$ curl --data "grant_type=urn:ietf:params:oauth:gra
```

You should now get back a JSON response that includes an access token and a refresh token. There are two common scenarios in which this could fail.

If you try this request before the user has approved the applicable `user_code`, you will get back an error saying the authorization request is still pending.

On the other hand, if there is a delay between the user approving the code and the application sending the request (by default 600 seconds) you will get an error saying the device code is not valid.

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 either on the device itself or

on a different device. 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 and 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:

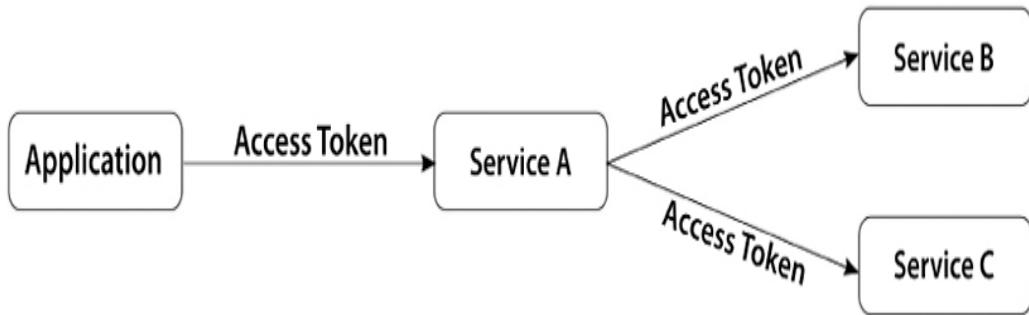


Figure 6.11: End-to-end user authentication for microservices

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 type:** OpenID Connect
- **Client ID:** service
- **Client authentication:** On
- **Standard flow:** Unchecked
- **Implicit Flow Enabled:** Unchecked
- **Direct Access Grants Enabled:** Unchecked
- **Service accounts roles:** Checked

The following screenshot shows the client you should create:

Myrealm

Manage

Clients

Client scopes

Realm roles

Users

Groups

Sessions

Capability config

Client authentication On

Authorization Off

Authentication flow

Standard flow

Direct access grants

Implicit flow

Service accounts roles

OAuth 2.0 Device Authorization Grant

OIDC CIBA Grant

Jump to section

General Settings

Access settings

Capability config

Login settings

Logout settings

Figure 6.12: Service account client in Keycloak

As you have turned off the **Standard flow** option for this client, it is not able to obtain tokens using the authorization code flow, but as it has **Service accounts roles** 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 (   
$ curl --data "client_id=service&client_secret=$SECI
```

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 WebSocket 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 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/>

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/SecNet>



7

Integrating Applications with Keycloak

So far, you have been presented with the main concepts and configuration options in Keycloak. In this chapter, you will learn how to apply them so that you can configure your applications and integrate them with Keycloak.

Through selected integration scenarios and coding examples, you will learn which integration approach works best for you according to the technology stack your applications are using and the platform they are running on. You will be presented with different integration options for applications using Go, Java, JavaScript, and Node.js. If none of the options work for you, then don't worry – you will learn how to integrate with Keycloak using a reverse proxy that sits in front of your application.

During this chapter, keep in mind that Keycloak is not opinionated about the integrations presented herein and how they are implemented. The focus is on showing you how it is possible to integrate Keycloak into your application as long as you are using a language or library that supports OpenID Connect.

By the end of this chapter, you will have a good understanding of some of the available integration options and how they apply to your applications and their runtime, as well as what you should consider if none of the options presented work for you, and you need to look at alternatives.

In this chapter, we are going to cover the following topics and integrations:

- Choosing an integration architecture
- Choosing an integration option
- Integrating with Golang applications
- Integrating with Java applications
- Integrating with JavaScript applications
- Integrating with Node.js applications
- Using a reverse proxy
- Try not to implement your own integration

Technical requirements

The example code for this chapter can be found in the GitHub repository associated with this book. If you have Git installed, you can clone the repository by running this command in a terminal:

```
$ git clone https://github.com/PacktPublishing/Keyc...
```

Alternatively, you can download a ZIP file of the repository from

<https://github.com/PacktPublishing/Keycloak--Identity-and-Access-Management-for-Modern-Applications-2nd-Edition/archive/refs/heads/main.zip>.

After cloning or extracting the repository, look at the `ch7` directory, which is where all the examples are located.

Before we begin, you need to run Keycloak on a different port. For that, start the server, as follows:

```
$ docker run -it -p 8180:8180 \
-e KEYCLOAK_ADMIN=admin -e KEYCLOAK_ADMIN_I
quay.io/keycloak/keycloak:22.0.0 \
start-dev --http-port=8180
```

We are running Keycloak on a different port because the example applications we are about to run will be listening on port `8080`, the default port used by Keycloak. The server should be available at `http://localhost:8180`.

Now that the server has been started, create a new realm called `myrealm`.

Since we are going to integrate Keycloak with different types of applications, we need to create a client in the `myrealm` realm for each of them.

Let's start by creating the `mybrowserapp` client, which we will use to protect browser-based apps:

- Client ID: mybrowserapp
- Root URL: http://localhost:8080
- Valid Redirect URI: /*
- Web origins: +

The mybrowserapp client is a public client that runs on a browser and is not able to securely store its credentials. You can think about it as the minimum configuration you should have to integrate your **Single-Page Applications (SPAs)** with Keycloak. Note that we also need to set the **Web origins** field to allow cross-origin requests from browser-based applications as they are running in different domains or URLs as the Keycloak server. By setting +, we are allowing any requests originating from any **Valid Redirect URI** previously set.

For protecting server-side web applications, we will use a mywebapp client:

- Client ID: mywebapp
- Client authentication: ON
- Root URL: http://localhost:8080
- Valid Redirect URI: /*

The mywebapp client is a confidential client that can securely store its credentials as they run on the server side. You can think about it as the minimum configuration you should have to integrate with Keycloak when your application runs on the backend, and it can handle the authentication flow by itself by relying on browser-level protection mechanisms to authenticate requests – for instance,

cookie-based authentication. This configuration can be used by applications serving both the frontend and backend.

Let's create the client we will be using to protect the backend application:

- **Client ID:** mybackend
- **Client authentication:** ON
- **Direct Access Grants:** ON
- **Root URL:** <http://localhost:8080>

As you will see in later chapters, the **Direct Access Grants** option should be disabled for clients. For the sake of simplicity, we are using it in examples to obtain tokens directly from the token endpoint on behalf of users. This option is basically enabling the OAuth2 resource owner password credentials grant type, which is now considered a bad practice. For more details, please look at the OAuth2 Security Best Current Practice at <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics>.

The `mybackend` client is a confidential client acting as a resource server. Its purpose is to authorize requests based on bearer tokens to allow access to its protected resources.

The last client we are going to create will be used by a reverse proxy running in front of an application. Create a client with the following settings:

- Client ID: proxy-client
- Client authentication: ON
- Root URL: http://localhost
- Valid Redirect URI: /*

The proxy-client is a confidential client that is going to be used by a reverse proxy to authenticate and forward the information about the subject to its backend services.

Finally, create a user in Keycloak:

- Username: alice
- Password: alice

Regardless of the technology you are using in your application, the configuration you just created will not change due to the interoperable nature of Keycloak. As you will see in the following sections, we are reusing the very same client configuration across example applications using different languages and libraries.

Before we dive into the different integrations and examples, let's understand how they are grouped into two main architectural styles, as well as how they impact how your application integrates with Keycloak.

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

<https://packt.link/rEPXu>

Choosing an integration architecture

There are two main integration styles, depending on where the integration code and configuration are located:

- Embedded
- Proxied

Integrations that are **embedded** into your application are usually provided by a third-party library, framework, web container, or application server. In this style, your application talks directly with Keycloak and is responsible for handling OpenID Connect requests and responses. Applications using this style usually need to implement some code or provide some form of configuration to enable support for OpenID Connect. Any setting you need to change will require you to redeploy your application:



Figure 7.1: Embedded integration style

On the other hand, in the **proxied** style, there is a layer of indirection between your application and Keycloak. The integration is managed by a service running in front of your application and is responsible for handling the OpenID Connect requests and responses on behalf of your application. Therefore, your application relies on HTTP headers to fetch tokens or any other security-related data associated with a request. The integration code and configuration are outside your application's boundaries and are managed through an external service:



Figure 7.2: Proxied integration style

There is no rule of thumb when selecting the best integration style. And sometimes, you may be constrained to using a specific one. They are not mutually exclusive, though, and it is perfectly fine to have both within your application's ecosystem.

The proxied style is a good fit if you do not have control over the application code (for example, legacy code) or if your application is behind a reverse proxy or API gateway and you want to leverage its capabilities. It also gives you the ability to control and manage the integration with Keycloak from a single place.

On the other hand, embedding the integration into your code is simpler as it does not require managing an external service, giving you more control over the integration. Your application is self-contained and, if the framework or library you are using provides good support for OpenID Connect, the integration is usually just a matter of writing a few lines of code or providing some configuration files.

You should also consider looking at OAuth 2.0 for Browser-Based Apps (<https://datatracker.ietf.org/doc/html/draft-ietf-oauth-browser-based-apps>) for the different application

architecture patterns you can use to effectively use OpenID Connect in your applications.

In this section, you learned about the two main architectural styles you can use to integrate with Keycloak.

In the next section, we will cover different integration options based on the styles that have been presented.

Choosing an integration option

In addition to choosing between the two architectural styles we've just mentioned, we should also understand some key points when it comes to enabling OpenID Connect in your applications.

There are quite a lot of client-side implementations for OpenID Connect and sometimes, you may find it hard to decide which one works better for you. If none of the options suggested here work for you, it is important to be aware of how to choose alternatives.

As a rule of thumb, the decision for a good integration should be based on implementations that fit into the following requirements:

- Widely adopted, actively maintained, and backed by a strong community of developers.
- Up to date with the latest versions of the OAuth2 and OpenID Connect specifications.
- Aligned with the security best practices for OAuth2 and OpenID Connect.
- Good user experience, a simple configuration, and a simple deployment model.

- Hides details from developers as much as possible while still providing good defaults to make your application aligned with security best practices.
- Avoids vendor lock-in and keeps your application compliant with OAuth2 and OpenID Connect as much as possible. Keycloak can integrate with any client that's compliant with these specifications.

Ideally, you should use whatever comes for free from the technology stack and platform on which your applications are deployed.

You may also consider looking at the OpenID Connect website for a list of certified implementations. The list is available at <https://openid.net/developers/certified/>.

In the next section, we are going to look at how to integrate Keycloak using different technology stacks.

The code examples provided in this chapter are not targeted at being run in production; instead, they demonstrate how to integrate Keycloak using different languages and libraries. We recommend following the instructions of the respective language or library for secure use in production.

Integrating with Golang applications

Go applications can integrate with Keycloak using whatever library you prefer, as long as it complies with the OpenID Connect or OAuth2 specifications.

For the sake of simplicity and to provide a generic example of how to integrate with Keycloak, we are going to use the <https://github.com/coreos/go-oidc> package. The code examples for this section are available in the following directory:

```
$ cd Keycloak---Identity-and-Access-Management-for-
```

In the preceding directory, you will find a `main.go` file that contains all the code you will need to run the example. From this file, you will see a `createConfig` function declaring all the configurations needed to integrate with Keycloak:

```
func createConfig(provider oidc.Provider) (oidc.OidcConfig, oauth2.Config) {
    oidcConfig := &oidc.Config{
        ClientID: "mywebapp",
    }
    config := oauth2.Config{
        ClientID:     oidcConfig.ClientID,
        ClientSecret: "CLIENT_SECRET",
        Endpoint:    provider.Endpoint(),
        RedirectURL: "http://localhost:8080/auth/callback",
        Scopes:       []string{oidc.ScopeOpenID, "profile", "email"},
    }
    return *oidcConfig, config
}
```

You should change the reference to `CLIENT_SECRET` in the function above with the secret that was generated by Keycloak for the `mywebapp` client. For that, navigate to the `mywebapp` client details page in the Keycloak Administration Console and click on the **Credentials** tab. The client secret should be available from the **Client secret** field in this tab.

Let's start the application by running the following command in the root directory of your project:

```
$ go run main.go
```

Your application should start and be available at `http://localhost:8080`. Now, try to access that URL and log in to Keycloak using the credentials for the user we created at the beginning of this chapter.

If the integration is working properly, you should be redirected to Keycloak to authenticate. After providing the user credentials, you should be redirected back to the application, now as an authenticated user, and a page will appear that contains the tokens that have been issued by the server.

In this section, you learned about the basics of how to integrate a Go application with Keycloak. The `go-oidc` package is a well-known package that provides OpenID Connect capabilities for client applications. It provides a good baseline for integrating with

Keycloak and allows you to enable authentication for your application.

At the time this book was written, there was no support from `go-oidc` to prevent **Cross-Site Request Forgery (CSRF)** and **Authorization Code Injection** attacks using **Proof Key for Code Exchange (PKCE)**. That is the reason for using the **state** parameter when making authorization requests to Keycloak.

There are also quite a few third-party libraries that are targeted at integrating with Keycloak. Unfortunately, we cannot recommend any of them since they are not in conformance with some of the recommendations that were mentioned at the beginning of this chapter – mainly because they are not backed by a strong community but by individuals.

In the next section, we are going to look at how to integrate Keycloak with Java applications.

Integrating with Java applications

Frameworks, web containers, and application servers that provide support for OpenID Connect and OAuth2 as part of their offerings should make your life a lot easier since the integration is already available to your application and there is no need to add any other dependencies.

Leveraging what is already in your technology stack is usually the best choice.

In the next few sections, we will look at the different integration options based on the most common Java frameworks.

Using Quarkus

Quarkus provides an OpenID Connect-compliant extension called `quarkus-oidc`. It provides a simple and rich configuration model that can protect both frontend and backend applications. Quarkus has built-in support for the most common **Integrated Development Environments (IDEs)**, such as IntelliJ and Eclipse, and you should be able to quickly create or configure an existing project in order to integrate it with Keycloak.

If you are new to Quarkus or just want to protect your applications using OpenID Connect and Keycloak, please look at the guides available at <https://quarkus.io/guides>. Most of these guides and code examples use Keycloak as an OpenID provider and will help you quickly get started. Search for guides using `OpenID Connect` as a keyword to filter all the available guides related to integrating with Keycloak.

In summary, the `quarkus-oidc` extension allows you to protect two main types of applications: `web-app` and `service`.

The `web-app` type represents applications that authenticate using Keycloak through the browser, using the authorization code grant type. These are usually frontend applications.

On the other hand, the `service` type represents applications that rely on bearer tokens issued by a Keycloak server to authorize access to their protected resources. These are usually backend applications, serving some sort of API to a frontend application.

To use the `quarkus-oidc` extension in your project, add the following dependency to the application's `pom.xml` file:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-oidc</artifactId>
</dependency>
```

The code examples for this section are available in this book's GitHub repository at the following link:

```
$ cd Keycloak---Identity-and-Access-Management-for-I
```

In the preceding directory, you will find a `frontend` directory and a `backend` directory, both of which contain all the code you will need to follow and run the upcoming examples.

In the next section, we are going to start looking at how to protect a Quarkus web application to authenticate users using Keycloak.

Creating a Quarkus client

In this section, we will look at the code examples that are available from the following directory:

```
$ cd Keycloak---Identity-and-Access-Management-for-I
```

Let's start by configuring a `web-app` application by adding the following properties to the

`src/main/resources/application.properties` file:

```
quarkus.oidc.auth-server-url=http://localhost:8180/  
quarkus.oidc.client-id=mywebapp  
quarkus.oidc.client-secret=CLIENT_SECRET  
quarkus.oidc.application-type=web-app  
quarkus.http.auth.permission.authenticated.paths=/*  
quarkus.http.auth.permission.authenticated.policy=al
```

From a configuration perspective, the main configuration options are as follows:

- The `quarkus.oidc.auth-server-url` property defines the URL from which the application should fetch the OpenID Connect Discovery document.
- The `quarkus.oidc.client-id` property maps a client in Keycloak with this application. For this application, we are going to use the `mywebapp` client, which we created at the beginning of this chapter.

- The `quarkus.oidc.client-secret` property is the secret that was generated by Keycloak when the client was created.
- The `quarkus.oidc.application-type` property defines that this application is a web application.
- The
`quarkus.http.auth.permissionauthenticated.paths`
and
`quarkus.http.auth.permissionauthenticated.policy`
properties define that all the paths in the applications require an authenticated user.

You should change the reference to `CLIENT_SECRET` in the configuration above with the secret that was generated by Keycloak for the `mywebapp` client. For that, navigate to the `mywebapp` client details page in the Keycloak Administration Console and click on the **Credentials** tab. The client secret should be available from the **Client secret** field in this tab.

Let's start the application by running the following command in the root directory of your project:

```
$ cd Keycloak---Identity-and-Access-Management-for-I  
$ ./mvnw quarkus:dev
```

Your application should start and be available at `http://localhost:8080`. Try to access that URL and log in to

Keycloak using the credentials for the user we created at the beginning of this chapter.

If the integration is working properly, you should be redirected to Keycloak to authenticate. After providing the necessary user credentials, you should be redirected back to the application, now as an authenticated user.

By default, Quarkus is going to set a cookie that will expire based on the expiration time of the token issued by Keycloak. If you are experiencing the user not being redirected to Keycloak to authenticate, you might want to clear your browser cookies. This behavior is something you can configure. For more details, look at the `quarkus-oidc` extension documentation.

In this section, you learned how to configure a web application in order to authenticate users using Keycloak. At this point, you should be able to create your own application or configure an existing one to authenticate users using Keycloak. For more details, please check out the extension's documentation at <https://quarkus.io/guides/security-oidc-code-flow-authentication-concept>.

In the next section, we will look at how to configure a backend application to authorize access to resources based on tokens issued by Keycloak.

Creating a Quarkus resource server

The code examples that will be presented in this section are available from the following GitHub repository:

```
$ cd Keycloak---Identity-and-Access-Management-for-
```

For backend applications that have been protected using an OAuth2 bearer token, the configuration is similar to configuring frontend applications, except for changing `quarkus.oidc.application-type` to `service`, as well as the `quarkus.oidc.client-id` property so that it maps to a different client in Keycloak:

```
quarkus.oidc.auth-server-url=http://localhost:8180/i  
quarkus.oidc.client-id=mybackend  
quarkus.oidc.application-type=service  
quarkus.http.auth.permission.authenticated.paths=/*  
quarkus.http.auth.permission.authenticated.policy=ai
```

The `quarkus.oidc.application-type` property, which is now set to `service`, indicates that this application should authorize access based on bearer tokens.

Let's start the application by running the following command in the root directory of your project:

```
$ cd Keycloak---Identity-and-Access-Management-for-I  
$ ./mvnw quarkus:dev
```

Your application should start and be available at <http://localhost:8080>. To access the resources in the running application, you will need an access token. To obtain one, use the following command:

```
$ export access_token=$(\  
curl -X POST http://localhost:8180/realm<myrealm> /token  
-d "client_id=mybackend&client_secret=CLIENT_SECRET<br><br>-H "content-type: application/x-www-form-urlencoded"  
-d "username=alice&password=alice&grant_type=password"  
| jq --raw-output ".access_token" \  
)
```

Once you have run this command, an access token will be saved in an `access_token` environment variable, and you can now access the application:

```
$ curl -X GET http://localhost:8080/hello \  
-H "Authorization: Bearer \"$access_token"
```

As a result, you should expect the following output from that command:

```
$ Hello RESTEasy
```

Now, if you try to access the application without a bearer token or use an invalid one, you should get a **401** status code, indicating that your request was forbidden:

```
$ curl -v -X GET http://localhost:8080/hello
```

The `quarkus-oidc` extension validates tokens based on whether they represent a **JSON Web Token (JWT)** or not. If the token is a JWT, the extension will try to validate the token locally by checking its signatures, audience, and expiration date. Otherwise, if the token is opaque and the format is unknown, it will invoke the token's introspection endpoint at Keycloak to validate it.

For Quarkus applications, the `quarkus-oidc` extension is the best option you have. It provides an amazingly simple configuration while providing a lot of other options you can use to customize its behavior.

We only covered the main steps of setting up the `quarkus-oidc` extension here so that you can authenticate your users through Keycloak.

There is a lot more you can do with this extension, such as leveraging capabilities for logout, obtaining information about the subject into your beans, multi-tenancy, and so on. For more details, please check out the extension's documentation at <https://quarkus.io/guides/security-oidc-bearer-authentication-concept>.

In the next section, we will look at how to integrate with Spring Boot applications.

Using Spring Boot

Spring Boot applications can integrate with Keycloak by leveraging Spring Security's OAuth2/OpenID libraries.

There are two main libraries, and each is targeted to a specific type of application: clients and resource servers.

The code examples in this section are available from the following GitHub repository:

```
$ cd Keycloak---Identity-and-Access-Management-for-I
```

In the preceding directory, you will find a `frontend` directory and a `backend` directory containing all the code you will need to follow and run the examples.

In the next section, we are going to start looking at how to enable a web application so that we can authenticate users using Keycloak.

Creating a Spring Boot client

The code examples presented in this section are available from the following GitHub repository:

```
$ cd Keycloak---Identity-and-Access-Management-for-I
```

The configuration for this application is available from the `src/main/resources/application.yaml` file:

```
spring:
  security:
    oauth2:
      client:
        registration:
          myfrontend:
            provider: keycloak
            client-id: mywebapp
            client-secret: CLIENT_SECRET
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/myfrontend"
            scope: openid
        provider:
          keycloak:
            issuer-uri: http://localhost:8180/realm
```

As you can see, the configuration is pretty straightforward. We are basically setting the client configuration we created at the beginning of the chapter and the URL for the **myrealm** realm in Keycloak so that Spring can fetch the OpenID Connect Discovery document

(https://www.keycloak.org/docs/latest/securing_apps/#endpoints) to discover the server endpoints and related metadata.

You should change the reference to `CLIENT_SECRET` in the configuration above with the secret that was

generated by Keycloak for the `mywebapp` client. For that, navigate to the `mywebapp` client details page in the Keycloak Administration Console and click on the **Credentials** tab. The client secret should be available from the **Client secret** field in this tab.

Let's start the application by running the following command in the root directory of your project:

```
$ cd Keycloak---Identity-and-Access-Management-for-  
$ ./mvnw spring-boot:run
```

Your application should start and be available at `http://localhost:8080`. Try to access that URL and log in to Keycloak using the credentials for the user we created at the beginning of this chapter.

If the integration is working properly, you should be redirected to Keycloak to authenticate. After providing the user credentials, you should be redirected back to the application, now as an authenticated user.

In this section, you learned how to configure a web application to authenticate users using Keycloak. With that, you should be able to create your own application or configure an existing one to authenticate users using Keycloak.

In the next section, we will look at how to configure a backend application to authorize access to resources based on tokens issued

by Keycloak.

Creating a Spring Boot resource server

The code examples presented in this section are available from the following GitHub repository:

```
$ cd Keycloak---Identity-and-Access-Management-for-...
```

For backend applications protected using an OAuth2 bearer token, the configuration is similar to configuring frontend applications. But here, the application is going to act as a resource server that validates JWT tokens:

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8180/realm/r...
```

Let's start the application by running the following command in the root directory of your project:

```
$ cd Keycloak---Identity-and-Access-Management-for-I...
$ ./mvnw spring-boot:run
```

Your application should start and be available at `http://localhost:8080`. To access resources in the running application, you now need an access token. To obtain one, use the following command:

```
$ export access_token=$(\
    curl -X POST http://localhost:8180/realms/myrealm \
        -d "client_id=mybackend&client_secret=CLIENT_SECRET" \
        -H "content-type: application/x-www-form-urlencoded" \
        -d "username=alice&password=alice&grant_type=password" \
    | jq --raw-output ".access_token" \
)
```

You should change the reference to `CLIENT_SECRET` in the command above with the secret that was generated by Keycloak for the `mywebapp` client. For that, navigate to the `mywebapp` client details page in the Keycloak Administration Console and click on the **Credentials** tab. The client secret should be available from the **Client secret** field in this tab.

Once you have run this command, an access token will be saved in an `access_token` environment variable, and you can now access the application:

```
$ curl -X GET http://localhost:8080/hello \
    -H "Authorization: Bearer \"$access_token"
```

As a result, you should expect the following output from that command:

```
$ Greetings from Spring Boot!
```

Now, if you try to access the application without a bearer token or use an invalid one, you should get a **401** status code, indicating that your request was forbidden:

```
$ curl -v -X GET http://localhost:8080/hello
```

In this section, you learned about how to use Spring Security's OAuth2/OpenID libraries to integrate with Keycloak. We only covered the main steps for setting up Spring Security here so that you can authenticate your users through Keycloak. For more details, please check out the Spring Security documentation at <https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/jwt.html>.

In the next section, we are going to look at how to integrate Keycloak with JavaScript applications.

Integrating with JavaScript applications

You will find different OpenID Connect client implementations for JavaScript that you can use to integrate Keycloak with your **SPAs**.

In this section, we are going to cover how to use the Keycloak JavaScript adapter, a client implementation provided by Keycloak that is targeted to JavaScript-based applications running in a browser, as well as for those using ReactJS or React Native.

The code examples for this section are available from the following GitHub repository:

```
$ cd Keycloak---Identity-and-Access-Management-for-JS
```

In the preceding directory, you will find all the code you'll need to follow and run the upcoming examples.

The first step to configuring your application with the Keycloak JS adapter is adding the `keycloak.js` library to your page:

```
<script type="text/javascript" src="KC_URL/js/keycloak.js">
```

Here, `KC_URL` is the URL where your Keycloak server is available, such as `http://localhost:8180` if you are running it locally.

By fetching the library from the server as opposed to embedding it in to your application, you are guaranteed to always be using the version of the library that is compatible with the Keycloak server your application is talking to.

Now that the library is available on your page, you need to create a `keycloak` object with the client's information and initialize it when the browser window is loaded:

```
const keycloak = new Keycloak();
await keycloak.init({ onLoad: "login-required" });
```

The `init` method is responsible for bootstrapping the adapter. When this method is invoked, the library is going to load the client configuration from the `public/keycloak.json` file and check whether the user is already authenticated. If they aren't authenticated yet, the adapter is going to redirect the user to Keycloak. Once the user is successfully authenticated and returns to your application, the `showProfile` function is then executed, which, in turn, is going to show a page with information about the user.

Now, let's start the application by running the following code:

```
$ cd Keycloak---Identity-and-Access-Management-for-
$ npm install
$ npm start
```

Your application should start and be available at `http://localhost:8080`. Try accessing that URL and logging into Keycloak using the credentials for the user we created at the beginning of this chapter.

If the integration is working properly, you should be redirected to Keycloak to authenticate. After providing the user credentials, you should be redirected back to the application, now as an authenticated user.

If your application needs to access protected resources in some backend server using a bearer token, you can easily obtain the access token from the `keycloak` object and pass it over when you make HTTP requests:

```
fetch('http://my.api/resource', {
  method: 'GET',
  headers: {
    Authorization: 'Bearer ' + keycloak
  }
}).then((data) => {console.log(data)}).catch((err) => {console.error(err)})
```

The Keycloak JavaScript adapter allows you to quickly integrate with Keycloak. This library was built due to the lack of good JavaScript libraries for OpenID Connect at the time it was created, which does not hold true anymore due to the number of libraries available today. This adapter is actively maintained under the Keycloak umbrella and is well documented, but still specific to integrating with Keycloak as opposed to being a generic and fully compliant OpenID Connect library.

Using OpenID Connect and OAuth2 in browser-based applications is surrounded by security concerns due to their nature. When it comes to

choosing a good library, you should follow the best practices as per the OAuth2 Security Best Practices for Browser-Based Apps, available at
<https://tools.ietf.org/html/draft-ietf-oauth-browser-based-apps>.

We have only scratched the surface here and there is far more you can do with OpenID, such as obtaining tokens issues from the server, refreshing tokens, or automatically doing this based on a certain period of time, and logouts.

For more details about the Keycloak JavaScript adapter, check out the documentation at

https://www.keycloak.org/docs/latest/securing_apps/#_javascript_adapter.

In the next section, we are going to look at how to integrate with Node.js applications.

Integrating with Node.js applications

For Node.js applications, Keycloak provides a specific adapter called the Keycloak Node.js adapter. Like other adapters, it is targeted to integrate with Keycloak rather than a generic OpenID Connect client implementation.

The Keycloak Node.js adapter hides most of the internals from your application through a simple API that you can use to protect your

application resources. The adapter is available as an `npm` package and can be installed into your project as follows:

```
$ npm install keycloak-connect
```

The code examples for this section are available from the following GitHub repository:

```
$ cd Keycloak---Identity-and-Access-Management-for-I
```

In the preceding directory, you will find a `frontend` directory and a `backend` directory, which contain all the code you'll need to follow and run the following examples.

Now that you have installed the `keycloak-connect` dependency in your application, we are going to look at how to configure your application as a client and as a resource server.

Creating a Node.js client

Once you've installed the `keycloak-connect` package, you need to change your application code so that it creates a `keycloak` object:

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

Since we are protecting a frontend application, we want to create a local session for our users so that they are not redirected to

Keycloak once they are authenticated. For that, note that the `Keycloak` object is created with a `memoryStore`:

```
var memoryStore = new session.MemoryStore();
```

Just like other Keycloak adapters, the configuration is read from a `keycloak.json` file containing the client configuration:

```
{
  "realm": "myrealm",
  "auth-server-url": "${env.KC_URL}:http://localhost",
  "resource": "mywebapp",
  "credentials" : {
    "secret" : "CLIENT_SECRET"
  }
}
```

You should change the reference to `CLIENT_SECRET` in the `keycloak.json` file with the secret that was generated by Keycloak for the `mywebapp` client. For that, navigate to the `mywebapp` client details page in the Keycloak Administration Console and click on the **Credentials** tab. The client secret should be available from the **Client secret** field in this tab.

The next step is to install the adapter as middleware so that you can use it to protect the resources in your application:

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

Now that the middleware has been installed, protecting the resources in your application should be as simple as doing the following:

```
app.get('/', keycloak.protect(), function (req, res) {
  res.setHeader('content-type', 'text/plain');
  res.send('Welcome!');
});
```

The `keycloak.protect` method automatically adds the necessary capabilities to your endpoints, to check whether users are authenticated yet or not so that they can be redirected to Keycloak if not. After successful authentication, the middleware will automatically process the response from Keycloak and establish a local session for the user based on the tokens issued by the server.

Now, let's start the application:

```
$ cd Keycloak---Identity-and-Access-Management-for-I
$ npm install
$ npm start
```

Your application should start and be available at <http://localhost:8080>. Try to access that URL and log in to

Keycloak using the credentials for the user we created at the beginning of this chapter.

If the integration is working properly, you should be redirected to Keycloak to authenticate. After providing the user credentials, you should be redirected back to the application, now as an authenticated user.

Creating a Node.js resource server

The code examples presented in this server are available from the following GitHub repository:

```
$ cd Keycloak---Identity-and-Access-Management-for-
```

For backend applications, you can create a `keycloak` object as follows:

```
var keycloak = new Keycloak({});
```

Compared to frontend applications, we do not need to track user sessions; instead, we must rely on bearer tokens to authorize requests.

Similar to the previous example, we also need to update the `keycloak.json` file with the client configuration:

```
{  
  "realm": "myrealm",
```

```
"bearer-only": true,  
"auth-server-url": "${env.KC_URL:http://localhost  
"resource": "mybackend"  
}
```

In this configuration, we are explicitly marking this application as **bearer-only** so that it only accepts bearer tokens, forcing the adapter to check whether a request can access resources in the application by performing local validations and introspections on the token.

The next step is to install the adapter as middleware so that you can use it to protect the resources in your application:

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

Now that the middleware has been installed, protecting the resources in your application should be as simple as doing the following:

```
app.get('/hello', keycloak.protect(), function (req.  
    res.setHeader('content-type', 'text/plain');  
    res.send('Access granted to protected resource'  
});
```

The **keycloak.protect** method automatically adds bearer token authorization to your endpoints so that requests containing an

authorization header with a valid token can fetch the protected resources in your application.

Now, let's start the application:

```
$ cd Keycloak---Identity-and-Access-Management-for-I  
$ npm install  
$ npm start
```

Your application should start and be available at <http://localhost:8080>. To access the resources in the running application, you will need an access token. To obtain one, use the following command:

```
$ export access_token=$(\  
curl -X POST http://localhost:8180/realms/myrealms/protocol/openid-connect/token  
-d "client_id=mybackend&client_secret=CLIENT_SECRET  
-H "content-type: application/x-www-form-urlencoded  
-d "username=alice&password=alice&grant_type=password"  
| jq --raw-output ".access_token" \  
)
```

You should change the reference to `CLIENT_SECRET` in the preceding command with the secret generated by Keycloak for the `mybackend` client. For that, navigate to the `mybackend` client details page in Keycloak and click on the **Credentials** tab. The client

secret should be available from the **Client secret** field in this tab.

Once you've run that command, an access token will be saved in an `access_token` environment variable, which means you can now access the application:

```
$ curl -v -X GET http://localhost:8080/hello \
-H "Authorization: Bearer $access_token"
```

As a result, you should expect the following output:

```
$ Access granted to protected resource
```

Now, if you try to access the application without a bearer token or use an invalid one, you should get a **403** status code, indicating that your request was forbidden:

```
$ curl -v -X GET http://localhost:8080/hello
```

There is much more you can do with the Keycloak Node.js adapter in terms of configuration and usage. You should be able to use `keycloak.protect` to perform role-based access control and obtain the tokens representing the authenticated subject.

For more details about the Keycloak Node.js adapter, check out the available documentation at

https://www.keycloak.org/docs/latest/securing_apps/#_no_dejs_adapter.

In this section, you learned how to configure your Node.js application so that you can integrate with Keycloak using the `keycloak-connect` library. Next, you will learn to leverage the proxied architectural style, which is useful if none of the options that have been presented so far are enough to address your requirements.

Using a reverse proxy

By running Keycloak in front of your application, you can use reverse proxies to add additional capabilities to your application. The most common proxies provide support for OpenID Connect where enabling authentication is a matter of changing the proxy configuration.

Whether using a proxy is better than having the integration code and configuration within your application really depends on the use case and, depending on the circumstances, it might be your only option or the option that will save you precious time from implementing your own integration code, even if you have a library available for the technology stack your application is using.

Nowadays, OpenID Connect and OAuth2 support is a mandatory capability for proxies, and you find support for these protocols in most of them, regardless of whether they're open source or proprietary. As an example, two of the most popular proxies,

Apache HTTP Server and Nginx, provide the necessary extensions for these protocols.

In this section, we are going to cover how to set up Apache HTTP Server in front of our application so that we can integrate it with Keycloak and authenticate users using `mod_auth_oidc`. The documentation on how to install it is available at https://github.com/zmartzone/mod_auth_openidc.

Once both Apache HTTP Server and the module have been installed, we need to configure the server so that we can proxy our application and use the module to make sure users are authenticated through Keycloak:

```
LoadModule auth_openidc_module modules/mod_auth_openidc.so
ServerName localhost
<VirtualHost *:80>
    ProxyPass / http://localhost:8000/
    ProxyPassReverse / http://localhost:8000/
    OIDCCryptoPassphrase CHANGE_ME
    OIDCProviderMetadataURL http://localhost:8180/realm-your-realm
    OIDCClientID proxy-client
    OIDCClientSecret CLIENT_SECRET
    OIDCRedirectURI http://localhost/callback
    OIDCCookieDomain localhost
    OIDCCookiePath /
    OIDCCookieSameSite On
<Location />
    AuthType openid-connect
    Require valid-user
```

```
</Location>  
</VirtualHost>
```

You should change the reference to `CLIENT_SECRET` in the preceding configuration with the secret generated by Keycloak for the `mywebapp` client. For that, navigate to the `mywebapp` client details page in Keycloak and click on the **Credentials** tab. The client secret should be available from the **Client secret** field in this tab.

Now, let's start the application:

```
$ cd Keycloak---Identity-and-Access-Management-for-  
$ npm install  
$ npm start
```

Your application should start and be available at `http://localhost`. Try to access that URL and log in to Keycloak using the credentials for the user we created at the beginning of this chapter.

If the integration is working properly, you should be redirected to Keycloak to authenticate. After providing the user credentials, you should be redirected back to the application, now as an authenticated user.

Try not to implement your own integration

OAuth2 and OpenID Connect are simple protocols, and their simplicity is, in part, due to the effort that's been made to make the protocol easier to *use* by client applications, but not necessarily to *implement* them from scratch. You may feel tempted to write your own code to integrate with Keycloak, but this is usually a bad choice.

You should rely on well-known and widely used libraries, frameworks, or capabilities provided by the platform where your application is deployed.

By doing that, you can focus on your business and, most importantly, delegate to people who are specialized and focused on these standards to keep their implementations always up to date with the latest versions of the specifications, as well as with any fixes for security vulnerabilities and security best practices.

Also, remember that the more people there are using an implementation, the less likely it is that you will face bugs and security vulnerabilities, which can impact not only your application but also your organization.

Summary

In this chapter, you learned how to integrate Keycloak with different types of applications, depending on the technology stack they are using, as well as the platform they are running. You also

learned about the importance of using well-known and established open standards and what that means in terms of interoperability. This means you are free to choose the OpenID Connect client implementation that best serves your needs, while still respecting compliance and keeping your applications up to date with the OAuth2 and OpenID Connect best practices and security fixes.

Finally, you learned why you should avoid implementing your own integration, as well as the things you should consider when you're looking for alternatives if none of the other options work for you.

In the next chapter, you will learn about the different authorization strategies you can use to protect your application resources.

Questions

1. What is the best way to integrate with Keycloak?
2. Should I always consider using the Keycloak adapters if they fit into my technology stack?
3. How should you secure a native or mobile application with Keycloak?
4. What is the best integration option for cloud-native applications?

Further reading

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

- Certified OpenID Connect Implementations:
<https://openid.net/developers/certified>
- OAuth 2.0 for Browser-Based Apps:
<https://tools.ietf.org/html/draft-ietf-oauth-browser-based-apps-07>
- OAuth 2.0 Security Best Current Practice:
<https://tools.ietf.org/html/draft-ietf-oauth-security-topics-16>
- Keycloak Quickstarts:
<https://github.com/keycloak/keycloak-quickstarts>
- Securing Applications and Services Guide:
https://www.keycloak.org/docs/latest/securing_apps

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/SecNet>



8

Authorization Strategies

In the previous chapter, you learned about the options for integrating with Keycloak using different programming languages, frameworks, and libraries. You learned how to obtain tokens from Keycloak and use these tokens to authenticate users.

This chapter will focus on the different authorization strategies you can choose from and how to leverage them to enable authorization for your applications using different access control mechanisms such as **role-based access control (RBAC)**, **group-based access control (GBAC)**, OAuth2 scopes, and **attribute-based access control (ABAC)**, as well as learning how to leverage Keycloak as a centralized authorization server to externalize authorization from your applications. You will also learn about the differences between these options and how to choose the best strategy for you.

By the end of this chapter, you will have a good understanding of how you can leverage Keycloak authorization capabilities and choose the right authorization strategy for your applications.

We will be covering the following topics in this chapter:

- Understanding authorization
- Using RBAC
- Using GBAC

- Using OAuth2 scopes
- Using ABAC
- Using Keycloak as a centralized authorization server

Understanding authorization

Any authorization system will try to help you to answer the question of whether a user can access a resource and perform actions on it.

The answer to this question usually involves questions such as the following:

- Who is the user?
- What data is associated with the user?
- What are the constraints for accessing the resource?

By getting the answers to these three questions, we can then decide if access should be granted based on the data associated with the user and the constraints that govern access to the resource.

As an identity provider, Keycloak issues tokens to your applications. As such, applications should expect authorization data from these tokens. Tokens issued by Keycloak carry information about the user and the context in which the user was authenticated; the context may contain information about the client the user is using or any other information gathered during the authentication process.

The constraints, however, may involve evaluating different types of data, from a single attribute the user has to a set of one or more

roles, or even data associated with the current transaction. By relying on the information carried by tokens, applications can opt for different access control mechanisms, depending on how they interpret the claims within a token when enforcing access to protected resources.

There are two main authorization patterns for implementing and enforcing the access constraints imposed on protected resources. The first, and probably the most common, is to enforce access control at the application level, either declaratively – using some metadata and configuration – or programmatically. On the other hand, applications can also delegate access decisions to an external service and enforce access control based on the decisions taken by this service, a strategy also known as centralized authorization. These two patterns are not mutually exclusive, though, and it is perfectly fine to use both in your applications. We are going to cover that in more detail later when understanding how to use Keycloak as a centralized authorization server.

As we will see in the following sections, Keycloak is very flexible and allows you to exchange any information you might need to protect resources at the application level using different access control mechanisms. It also allows you to choose from different authorization patterns for managing and enforcing access constraints.

In the next sections, we will look at how Keycloak can be used to enable different authorization strategies for your applications.

Using RBAC

Probably one of the most-used access control mechanisms, **RBAC** allows you to protect resources depending on whether the user is granted a **role**. As you learned in previous chapters, Keycloak has built-in support for managing roles, as well as for propagating those roles to your applications using tokens.

Roles usually represent a role a user has in either your organization or in the context of your application. As an example, users can be granted an **administrator** role to indicate they act as someone allowed to access and perform actions on any resource in your application. Or, they can be granted a **people-manager** role to indicate that they act as someone allowed to access and perform actions on resources related to their subordinates.

As you learned from previous chapters, Keycloak has two categories of roles: realm and client roles. Roles defined at the realm level are called **realm roles**. These roles usually represent the user's role within an organization, regardless of the different clients that co-exist in a realm.

On the other hand, **client roles** are specific to a client, and their meaning depends on the semantics used by the client.

The decision of when to define a role as a realm or client role depends on the scope the role has. If it spans multiple clients in a realm while keeping the same meaning, then a realm role makes sense. Otherwise, if only a specific client is supposed to interpret the role, having it as a client role makes more sense.

When using roles, you should also avoid role *explosion*. In other words, too many roles in your system make things hard to manage. One way to avoid this is to create roles very carefully, having in

mind the scope they are related to (realm- or client-wide) and the granularity of the permissions associated with them in your applications. The more fine-grained the scope of a role is, the more roles you will have in your system. As a rule of thumb, do not use roles for fine-grained authorization in your system. They are just not meant for that.

In Keycloak, you can grant roles to groups. That is a powerful capability where members of a group are automatically granted roles for the group they belong to. By leveraging this capability, you should be able to overcome some of the role management issues by avoiding granting privileges individually to many users.

Keycloak also provides the concept of **composite roles**, a special type of role that chains other roles, where a user granted a composite role is automatically granted any role in this chain (a regular role or even another composite role). Although it is a powerful and unique feature that Keycloak has, you should use it carefully to avoid performance issues – such as when chaining multiple composite roles – as well as manageability issues due to the proliferation of roles in your system and the granularity of the permissions associated with them. As a recommendation, if you need to grant multiple roles to your users, you should consider using groups and assigning roles to these groups. This is a more natural permission model than using composite roles.

The way you model your system roles also has an impact on the size of tokens issued by Keycloak. Ideally, tokens should contain the minimum set of roles the client needs to authorize their users

either locally or when accessing another service that consumes these tokens.

Keep in mind that the more roles your system has, the more complex it will become to maintain and manage.

In this topic, you learned about the main concepts when using RBAC in Keycloak. You also learned about some recommendations and considerations when using roles that may impact your applications in terms of maintainability and performance.

In the next topic, we will look at how Keycloak helps you to implement GBAC and recommendations when using it in applications.

Using GBAC

Keycloak allows you to manage groups for your realms. Users are put into groups to represent their relationship with a specific business unit in your organization (mapping your organization tree) or just grouped together according to their role in your applications, such as when you want to have a specific group for users that can perform administrative operations.

Usually, groups and roles are used interchangeably, and this causes some confusion when defining a permission model. In Keycloak, there is a clear separation between these two concepts where, different from roles, groups are meant to organize your users and grant permissions according to the roles associated with a group.

By allowing assigning roles to groups, Keycloak makes it a lot easier to manage roles for multiple users without forcing you to grant and

revoke roles for each individual user in your realm.

Groups in Keycloak are hierarchical, and when tokens are issued, you can traverse the hierarchy by looking at the path of the group. For instance, suppose you have a `human` `resource` group. As a child of this group, you have a `manager` group. When Keycloak includes information about groups in tokens, you should expect this information in the following format: `/human` `resource/manager`. This information should be available for every token issued by the server where the subject (the user) is a member of the group.

Different from roles, group information is not automatically included in tokens. For that, you should associate a specific protocol mapper with your client (or a client scope with the same mapper).

In the next sections, you will learn how to include group information about users into tokens.

Mapping group membership into tokens

Different from roles, there is no default protocol mapper that automatically includes group information in tokens. To do that, we need to create a protocol mapper for your client.

Alternatively, you can also create a client scope and assign it to any client in your realm.

Let's start by creating the `myclient` client:

- Client ID: `myclient`

Now, create a user in Keycloak:

- **Username :** `alice`

Navigate to the **myclient** settings and click on the **Client scopes** tab. In this tab, you will see a list of all the client scopes configured for the client. Click on the **myclient-dedicated** link to create a add a new mapper to the client.

The screenshot shows the 'Client scopes' tab selected in the 'myclient' client settings. The top navigation bar includes tabs for 'Settings', 'Roles', 'Client scopes' (which is active), 'Sessions', and 'Advanced'. Below the tabs, there are two buttons: 'Setup' and 'Evaluate'. A search bar at the top right contains the placeholder 'Search by name' and a blue 'Add client scope' button. The main area displays a table of client scopes. The columns are 'Name', 'Assigned client...', 'Assigned type', and 'Description'. Three rows are visible: 'myclient-dedicated' (assigned to 'none'), 'acr' (assigned to 'Default'), and 'address' (assigned to 'Optional'). Each row has a three-dot menu icon on the far right.

Figure 8.1: Adding a mapper to the client's dedicated client scope

To create a new mapper, click on the **Configure a new mapper** button and select the **Group Membership** mapper from the list. Now, configure it as follows:

Group Membership

Action ▾

d71819de-fa5a-43c1-b501-9750ddce6c7c

Mapper type

Group Membership

Name * ⓘ

groups

Token Claim Name ⓘ

groups

Full group path ⓘ

On

Add to ID token ⓘ

On

Add to access token ⓘ

On

Add to userinfo ⓘ

On

Save

Cancel

Figure 8.2: Creating a group membership protocol mapper

On this page, create a new mapper with the following information:

- **Name:** groups
- **Mapper Type:** Group Membership
- **Token Claim Name:** groups

Now, click on the **Save** button to create the mapper.

Let's now create a group for this user. For that, click on the **Groups** link in the left-hand menu:

Groups

A group is a set of attributes and role mappings that can be applied to a user. You can create, edit, and delete groups and manage their child-parent organization. [Learn more](#) 



No groups in this realm

You haven't created any groups in this realm. Create a group to get started.

[Create group](#)

Figure 8.3: Listing groups

To create a new group, click on the **Create group** button:

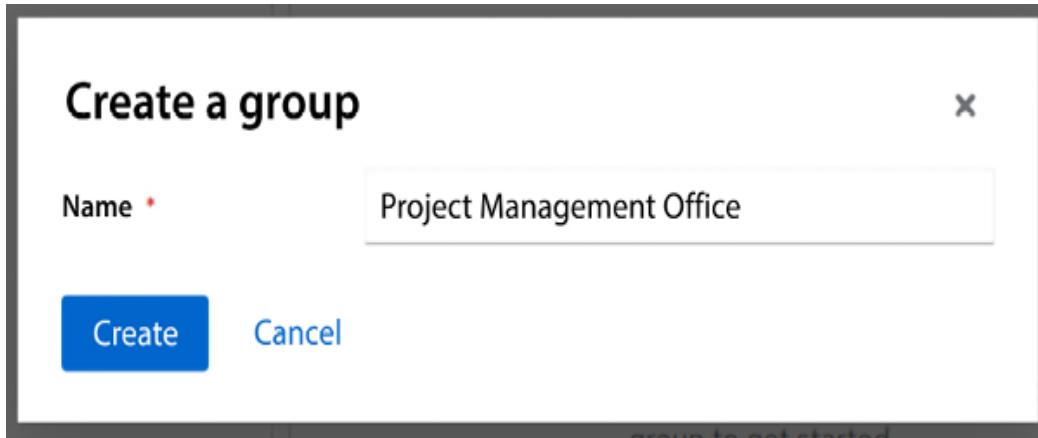


Figure 8.4: Creating a new group

Let's create a group named **Project Management Office**. Type this name in the **Name** field and then click on the **Create** button.

Now, let's add the `alice` user as a member of this group. For that, navigate to the `alice` user details page and click on the **Groups** tab:

The screenshot shows a user profile for 'alice'. At the top, there is a blue toggle switch labeled 'Enabled' and a dropdown menu labeled 'Action ▾'. Below the header, a navigation bar contains tabs: 'Details', 'Attributes', 'Credentials', 'Role mapping', 'Groups' (which is highlighted in blue), and 'Consents'. A large 'plus' icon is centered above the text 'No groups'. Below this, a message reads: 'You haven't added this user to any groups. Join a group to get started.' A prominent blue button at the bottom is labeled 'Join Group'.

Figure 8.5: Managing group membership for a user

From this page, click on the **Join Group** button to associate the user as a member of the group. On this page, select the **Project Management Office** group and click on the **Join** button.

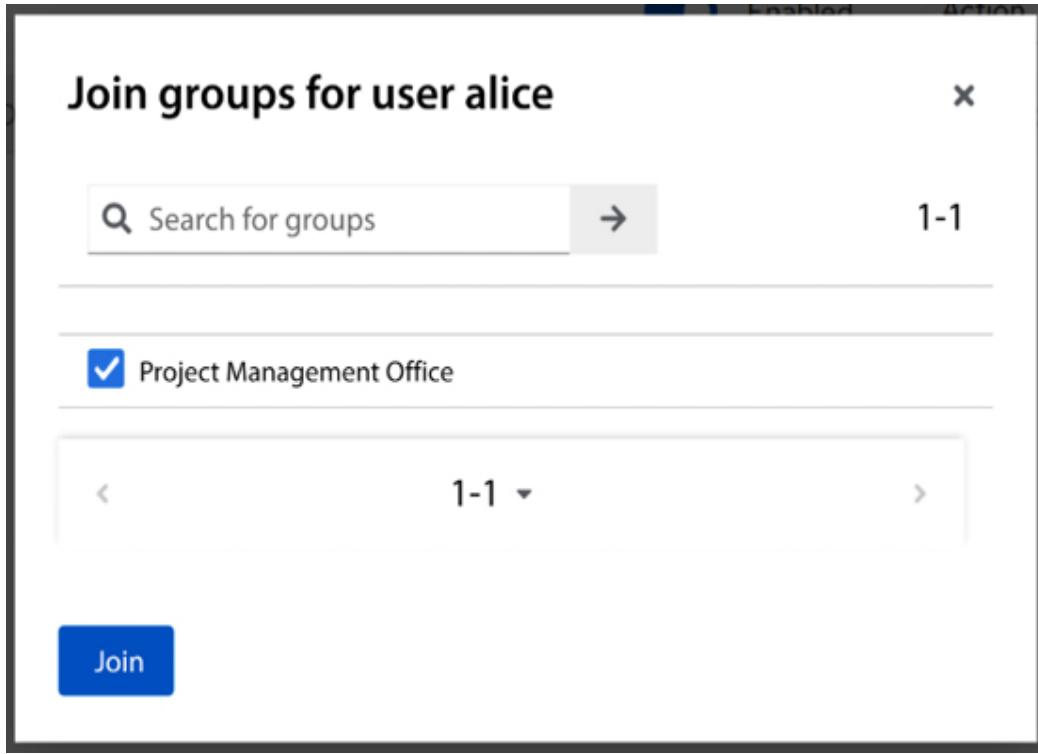


Figure 8.6: Assigning users as a member of a group

The `alice` user is now a member of **Project Management Office**.

Let's now go back to the **myclient** details page and use the evaluation tool to see how group information will be added to tokens.

Click on the **Client Scopes** tab. In this tab, click on the **Evaluate** sub-tab:

myclient OpenID Connect Enabled Action

Clients are applications and services that can request authentication of a user.

Settings Roles Client scopes Sessions Advanced

Setup Evaluate

This page allows you to see all protocol mappers and role scope mappings

Scope parameter ?

openid X Select scope parameters

User ?

alice X

Figure 8.7: Using the evaluation tool to check group information

Search for the `alice` user in the **User** field and then click on the **Generated access token** link at the bottom left of the page to see if the generated token includes information about the groups that the user belongs to:

Setup Evaluate

This page allows you to see all protocol mappers and role scope mappings

Scope parameter ⓘ Select scope parameters

User ⓘ

```
{
  "sub": "urn:oid:342405b3-1dec-47c6-839f-d9208c9e1395",
  "acr": "1",
  "sid": "342405b3-1dec-47c6-839f-d9208c9e1395",
  "email_verified": false,
  "groups": [
    "/Project Management Office"
  ],
  "preferred_username": "alice",
  "given_name": "",
  "family_name": ""
}
```

Effective protocol mappers ⓘ

Effective role scope mappings ⓘ

Generated access token ⓘ

Generated ID token ⓘ

Generated user info ⓘ

Figure 8.8: Evaluation result

As you can see, the generated token now includes a **groups** claim with a list of groups the user is a member of. In this case, the user **alice** is a member of a single **Project Management Office** group.

In this section, you learned how to manage groups and how to make a user a member of a group. You also learned how to use a protocol mapper to include group information in tokens so that your application can use this information to enforce access control using the groups that a user belongs to.

In the next section, we are going to look at how your applications can use custom claims to enforce access to their resources.

Using OAuth2 scopes

At its core, Keycloak is an OAuth2 authorization server. In pure OAuth2, there are two main types of applications: clients and resource servers.

As you learned from previous chapters about OAuth2, access tokens are issued to clients so that they can act on behalf of a user, where these tokens are limited to a set of scopes based on the user's consent.

On the other hand, resource servers are the consumers of access tokens, which they need to introspect to decide whether the client can access a protected resource on the resource server accordingly to the scopes granted by the user.

As you can see, authorization using OAuth2 scopes is solely based on user consent. It is the best strategy when you want third parties integrating with your APIs so that you delegate to your users the decision on whether a third-party application can access their resources. In this strategy, the main point is to protect user information rather than regular resources at the resource server. There is a fundamental difference between using OAuth2 scopes and the other authorization strategies you learned so far, mainly in terms of the entity you are protecting your system from. By using OAuth2 scopes, you are protecting your system from clients, whereas when using RBAC, for instance, you are protecting your

system from users. In a nutshell, you are basically checking whether a client is allowed to perform some action or access a resource on behalf of the user, the usual delegation use case solved by OAuth2.

By default, clients in Keycloak are configured to not ask for user consent. The reason for that is that Keycloak is usually used in enterprise use cases. In contrast with the delegation use case, there is no need for user consent because clients are within the enterprise boundaries and the resources they need to access do not depend on any consent from users but on the permissions granted to them by a system administrator. Here, clients are more interested in authenticating users, where the scope of access is defined according to the roles, groups, or even specific attributes associated with a user.

In this topic, you learned about the concepts of authorizing access using OAuth2 scopes. You also learned that this authorization strategy is more suitable for allowing access from third parties to information about your users through your APIs.

In the next topic, we will look at how to authorize access based on claims mapped to tokens.

Using ABAC

When users authenticate through Keycloak, tokens issued by the server contain important information about the authentication context. Tokens contain information about the authenticated user and the client to which tokens were issued, as well as any other

information that can be gathered during the authentication process. With that in mind, any information carried by a token can be used to authorize access to your applications. They are just claims mapped to tokens.

ABAC involves using the different attributes associated with an identity (represented by a token), as well as information about the authentication context, to enforce access to resources. It is probably the most flexible access control mechanism you can choose, with natural support for fine-grained authorization. Together with token-based authorization, applications using Keycloak can easily enable ABAC to protect their resources.

Token-based authorization is based on introspecting tokens and using the information there to decide whether access should be granted. This information is represented as a set of attributes, or claims, and their values can be used to enforce access.

Let's take as an example how roles are used to enforce access in your application. As you learned from the previous chapters and topics, roles are mapped to tokens using a specific set of claims. To enforce access using roles, your application only needs to use these claims to calculate what roles were granted to the user and then decide whether access should be granted to a particular resource.

This is no different from any other claim within a token, where your applications can use any claim and use it to enforce access. For each client, you can tailor what claims and assertions are stored in tokens. For that, Keycloak provides a functionality called protocol mappers. For more details, check out the Keycloak documentation at

https://www.keycloak.org/docs/latest/server_admin/#_protocol-mappers.

In this topic, you learned about how to leverage claims mapped into tokens to perform ABAC. You also learned that Keycloak allows you to map any information you want to tokens so that they can be used to enforce access at the application level. Although ABAC is flexible enough to support multiple access control mechanisms, it is not easy to implement and manage.

In the next topic, we are going to look at how to leverage ABAC using Keycloak as a centralized authorization server.

Using Keycloak as a centralized authorization server

So far, you have been presented with authorization strategies that rely on a specific access control mechanism. Except for ABAC, these strategies rely on a specific set of data about the user to enforce access to applications. In addition to that, these strategies are tightly coupled with your applications; changes to your security requirements would require changes in your application code.

As an example, suppose you have the following pseudo-code in your application:

```
If (User.hasRole("manager")) {  
    // can access the protected resource  
}
```

In the preceding code, we have a quite simple check using RBAC where only users granted a `manager` role can access a protected resource. What would happen if your requirements changed and you also needed to give access to that same resource to a specific user? Or even grant access to that resource for users granted some other role? Or perhaps leverage ABAC to look at the different information about the context where a resource is being accessed?

At the very least, you would need to change your code and redeploy your application, not to mention go through your continuous integration and delivery process to make sure the change is ready for production.

Centralized authorization allows you to externalize access management and decisions from your applications using an external authorization service. It allows you to use multiple access control mechanisms without coupling your application to them, and enforce access using the same semantics used by your applications to refer to the different resources that should be protected.

Let's take a look at the following code, which provides the same access check as the previous example:

```
If (User.canAccess("Manager Resource")) {  
    // can access the protected resource  
}
```

As you can see from the preceding code snippet, there is no reference to a specific access control mechanism; access control is

based on the resource you are protecting, and your application is only concerned with the permissions granted by an external authorization service.

Changes to how **Manager Resource** can be accessed should not impact your application code, but changing the policies that govern access to that resource through the authorization service should.

Keycloak can act as a centralized authorization service through a functionality called **Authorization Services**. This functionality is based on a set of policies representing different access control mechanisms that you associate with the resources you want to protect. All this is managed through the Keycloak administration console and REST API.

The Keycloak Authorization Services functionality leverages ABAC to enable fine-grained authorization for your applications. By default, a set of policies representing different access control mechanisms is provided out of the box, with the possibility to aggregate these policies to easily support multiple authorization strategies when protecting resources. The Keycloak Authorization Services functionality also allows you to control access to specific actions and attributes associated with the resources you are protecting.

A common issue when using a centralized authorization server is the need for additional round trips to obtain access decisions. By leveraging token-based authorization, the Keycloak Authorization Services functionality allows you to overcome this issue by issuing tokens with all the permissions granted by the server so that applications consuming these tokens do not need to perform

additional network calls but introspect the token locally. It also supports incremental authorization, where tokens are issued a narrow set of permissions with the possibility to obtain new permissions as needed.

For more details about Keycloak Authorization Services, check the documentation at

https://www.keycloak.org/docs/latest/authorization_services/.

In this section, you learned about centralized authorization and that Keycloak Authorization Services can be used to implement this form of authorization. You also learned that together with token-based authorization, Keycloak Authorization Services helps applications enable fine-grained authorization for applications.

Summary

In this chapter, you learned about the different strategies you can choose from to authorize access to protected resources in your applications. By leveraging token-based authorization, applications should be able to introspect tokens – either locally or through the introspection endpoint – and use their claims to support different access control mechanisms, such as RBAC, GBAC, and ABAC, or use the scopes granted by users to the client application acting on their behalf.

You also learned that Keycloak can be used as a centralized authorization service to decouple authorization from applications,

where access decisions are taken by Keycloak based on the resources and policies managed through the server.

In the next chapter, we are going to look at the main steps for running Keycloak in production.

Questions

1. How do you prevent tokens from becoming too big while still providing the necessary data to enforce access to resources at the application level?
2. How do you decide whether a role should be a realm or client role?
3. Is it possible to enforce access based on information gathered during authentication?
4. Is it possible to change how Keycloak maps roles to tokens?
5. Are the preceding two strategies mutually exclusive?

Further reading

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

- Keycloak roles:
https://www.keycloak.org/docs/latest/server_admin/#roles
- Keycloak groups:
https://www.keycloak.org/docs/latest/server_admin/#groups

- Keycloak protocol mappers:
https://www.keycloak.org/docs/latest/server_admin/#protocol-mappers
- Keycloak client scopes:
https://www.keycloak.org/docs/latest/server_admin/#client_scopes
- Keycloak Authorization Services:
https://www.keycloak.org/docs/latest/authorization_services

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/SecNet>



9

Configuring Keycloak for Production

Regardless of where you host the cluster, whether on-premises or in the cloud, the configuration options and the steps to start the server in production mode are the same. In this chapter, you will create a pre-production Keycloak cluster to understand all the various aspects and steps involved when configuring it for production.

In the next sections, you will be introduced to each of these aspects and how they fit into a real production deployment of Keycloak. By the end of this chapter, you should be able to apply the same steps and recommendations presented to deploy Keycloak in your own production environment using a high-availability profile, considering different non-functional aspects such as availability, performance, and failover.

For that, we will be covering the following topics:

- Setting the hostname for Keycloak
- Enabling TLS
- Configuring a database
- Enabling clustering
- Configuring a reverse proxy

- Evaluating your environment

Technical requirements

For this chapter, you 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:

```
$ cd $KC_HOME  
$ git clone https://github.com/PacktPublishing/Keyc:  
◀ ▶
```

Alternatively, you can download a ZIP of the repository from the following URL:

<https://github.com/PacktPublishing/Keycloak---Identity-and-Access-Management-for-Modern-Applications-2nd-Edition/archive/main.zip>

Make sure to either clone or extract the repository into the Keycloak distribution directory.

In order to secure incoming requests to the cluster as well as load balance requests across the different nodes to achieve high availability and scalability, we are going to configure a reverse proxy using a local domain name other than `localhost`. This domain name will be used as the public domain name where Keycloak is exposed for your users and applications.

If you are using Linux, you should be able to do that by changing your `/etc/hosts` file and including the following line:

```
127.0.0.1 mykeycloak admin.mykeycloak mypostgres
```

As a reverse proxy, we are going to use HAProxy in front of multiple Keycloak instances. If you are using CentOS or Fedora Linux, you should be able to install HAProxy as follows:

```
$ sudo dnf -y install haproxy
```

As a database, we are going to run a PostgreSQL container. For that, make sure you have Docker installed and execute the following command:

```
$ docker run --name mypostgres --network host -e PO
```

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

<https://packt.link/UCzBQ>

In the next topic, we will be looking at how to configure the URLs where Keycloak is exposed.

Setting the hostname for Keycloak

Keycloak exposes different endpoints to talk with applications as well as to allow managing the server itself. These endpoints can be

categorized into three main groups:

- Frontend
- Backend
- Administration

The base URL (e.g., scheme, host, port, and path) for each group has an important impact on how tokens are issued and validated, on how links are created for actions that require the user to be redirected to Keycloak (for example, when resetting passwords through email links), and, most importantly, how applications will discover these endpoints when fetching the OpenID Connect Discovery document from `/realms/{realm-name}/.well-known/openid-configuration`.

In the next topics, we will be looking into each of these groups, how to define a base URL for each one, and the impact it has on users and applications using Keycloak.

Setting the frontend URL

The **frontend URL** is used to infer the URL used by users and applications to access Keycloak through the front channel when executing browser-based flows, like when accessing the login page or clicking on a link to reset the password, and when binding tokens to a specific security domain by setting their issuer (e.g., the `iss` claim in the ID, Access, and Refresh Tokens). The main goal is to logically group all instances of Keycloak in a cluster under a single security domain and issuer by using a shared base URL with the same HTTP scheme, host, and port.

Without a frontend URL set, if your clients are accessing the server through TLS using `https://mykeycloak` and through plain HTTP using `http://mykeycloak`, the resulting frontend URL will not be consistent as they might be using different HTTP schemes when interacting with the server. The lack of consistency, in this case, might result in clients not sharing the same token issuer and being treated as if they were in distinct security domains, as well as blocking users from completing flows like resetting a password or accessing the login page.

By setting a base frontend URL you are also preventing common vulnerabilities and attacks like Host Injection so that any URL created by the server is fixed and under your control. This is especially true when using a reverse proxy or load balancer that does not allow configuring how request headers are overridden before forwarding requests to backend nodes.

Last but not least, without a base frontend URL set, cookies will not be recognized by nodes other than the one where they were created, therefore impacting some key functional and non-functional aspects like single sign-on, high availability, and scalability.

Examples of endpoints in the frontend group include the following:

- Authorization endpoint
- Logout endpoint and others related to session management
- Endpoints that manage flows that rely on end-users clicking on a link, like when resetting a password
- Endpoints that rely on creating responses to redirect users to another location, like when redirecting users to the login page

The expected behavior when accessing a production-ready cluster is that regardless of the node processing the request, the base URL should enforce the communication through a secure channel using HTTPS/TLS and be aligned with the public-facing URL where Keycloak is being exposed. By doing that, instances are going to work as if they were one so that users and applications can benefit from all the improvements we will cover later to the security, general availability, performance, scalability, and failover aspects of Keycloak.

The frontend URL can be configured by setting the `hostname` or `hostname-url` option.

For most use cases, setting the frontend URL through the `hostname` option is enough, because Keycloak is usually running behind a TLS termination proxy and exposed using the `https` scheme and the standard HTTPS port, 443.

Setting only the `hostname` is the easiest way to define a base frontend URL as you only need to provide the domain part of the URL where Keycloak is exposed and let it resolve the HTTP scheme and the port automatically based on secure defaults. Edit the `$KC_HOME/conf/keycloak.conf` file and add the following option:

```
hostname=mykeycloak
```

By setting the `hostname` option, Keycloak will enforce a secure communication channel by forcing the `https` scheme and the default `8443` HTTPS port to its URLs. As a result, the base URL will be `https://mykeycloak:8443`.

Under some circumstances, you might want to provide the full base frontend URL. For instance, if you are exposing Keycloak through a reverse proxy using a different port or path.

For that, you can use the `hostname-url` option to force a specific URL:

```
hostname-url=https://mykeycloak:1234/mypath
```

When explicitly setting a base URL, the server is going to use it as is to build the frontend URLs. In this case, the base URL will be `https://mykeycloak:1234/mypath` and it is expected that there is a reverse proxy listening on a secure port at `1234` and forwarding requests to backend nodes accordingly.

In this section, you learned that setting a base frontend URL allows you to define the base URL where Keycloak is publicly accessible. You also learned that setting this configuration is crucial to group all instances of Keycloak under a single and logical domain and issuer. Lastly, you learned that Keycloak relies on secure defaults when creating and exposing the server URLs by enforcing HTTPS for secure communication.

In the next topic, we will be looking at how to configure the URL for backend endpoints.

Setting the backend URL

The backend endpoints are those related to direct communication between Keycloak and applications.

Examples of endpoints in the backend group include the following:

- Token introspection
- User info
- Token endpoint
- JWKS

By default, the backend base URL is based on the incoming request URL. Usually, you also want the endpoints mentioned above to use a public domain name, so that applications can reach your cluster from outside its internal network. If your clients are reaching your cluster through a proxy that is properly overriding the HTTP scheme, host, and port headers, then the URL for the backend endpoints is going to be based on the frontend URL, where your cluster is publicly exposed. Most of the time, you do not need to set any specific option to configure the backend base URL and the default behavior is enough.

However, if clients might eventually reach your cluster directly through the internal network and you want to force them to use the same base URL used in the frontend endpoints when accessing the backend endpoints, then you can set the `hostname-strict-backchannel` option.

This option allows you to force the frontend URL to be the base URL for backend endpoints by setting its value to `true`:

`hostname-strict-backchannel=true`

When the `hostname-strict-backchannel` property is set, Keycloak will advertise backend endpoints using whatever you defined as a frontend URL, thus giving applications an accessible URL and not something else based on your internal network.

In this topic, you learned how to configure the base URL for backend endpoints and how they influence applications when they need to talk to Keycloak using the backend endpoints.

In the next topic, you will learn how to set the base URL for the administration endpoints.

Setting the admin URL

You usually do not want to make the Keycloak administration console publicly available. Similarly to how you set the base frontend URL, you can also set the base URL for resources and endpoints available through the management interface such as the administration console.

By default, the admin base URL is also based on the incoming request and as such, you need to explicitly configure it if you want to expose it only from a specific URL.

The admin URL can be configured by setting the `hostname-admin` or `hostname-admin-url` option. Setting only `hostname-admin` is the easiest way to define a base frontend URL as you only need to provide the domain part of the URL where the administration console should be exposed and let Keycloak resolve the HTTP

scheme and the port automatically based on secure defaults. Edit the `$KC_HOME/conf/keycloak.conf` file and add the following option:

```
hostname-admin=admin.mykeycloak
```

By setting the `hostname-admin` option, Keycloak will enforce a secure communication channel by forcing the `https` scheme and the default `8443` HTTPS port to its URLs. As a result, the base admin URL will be `https://admin.mykeycloak:8443`.

To set the full base admin URL you can use the `hostname-admin-url` option as follows:

```
hostname-admin-url=https://admin.mykeycloak
```

By setting a base admin URL, any URL used by the Admin Console will be based on the value you provided. That said, links and static resources used to render the console will only be accessible using this base URL.

Setting the admin URL is especially useful when making the administration console only accessible through a secure network and avoiding exposing it publicly. Although it makes it difficult to access the console from a public network by using a URL not accessible from there, you still want to enforce specific rules in your reverse proxy to add an additional security layer when accessing the administration console. For instance, you may want to restrict access based on a whitelist.

In the next topic, we will be looking at how to enable TLS.

Enabling TLS

Any request to and from Keycloak should be made through a secure channel. For that, you must enable HTTP over TLS, also known as HTTPS. In a nutshell, you should never expose Keycloak endpoints through plain HTTP.

Keycloak exchanges sensitive data all the time with user agents and applications. Enabling HTTPS is crucial to prevent several forms of attacks, as well as to benefit from different forms of authentication that rely on a TLS session established with the server.

The current best practice is to select a key size of at least 2,048 bits. In terms of protocol, Keycloak advertises the most secure protocols, such as TLS v1.2 and TLS v1.3. You should also be able to restrict the list of protocols to only advertise those you want by setting the `https-protocols` option. For more details, look at the documentation available from

https://www.keycloak.org/server/enabletls#_relevant_options.

The first step to enable HTTPS is to create or reuse a Java KeyStore where the server's private key and certificates are stored. If you are planning to deploy Keycloak in production, you probably have all the key material to enable TLS, as well as your certificates signed by a trusted **Certificate Authority (CA)**. The next and last step is to configure the HTTPS listener to use the key material from your Java KeyStore.

In this section, you are going to use a Java KeyStore available from the GitHub repository of the book at `$KC_HOME/Keycloak---Identity-and-Access-Management-for-Modern-Applications-2nd-Edition/ch9/mykeycloak.keystore`. This KeyStore was built for example purposes using a self-signed certificate and you should not use it in production. Instead, you should replace it with a KeyStore using your own private key and certificate.

To enable HTTPS using a Java KeyStore, you can use the `https-key-store-file` to provide the path to the KeyStore file and `https-key-store-password` to provide its password.

For that, copy the key store from the Git repository to the `conf` directory:

```
$ cp $KC_HOME/Keycloak---Identity-and-Access-Manager
```

Then, edit the `$KC_HOME/conf/keycloak.conf` file and add the following options:

```
https-key-store-file=${kc.home.dir}/conf/mykeycloak  
https-key-store-password=password
```

Now you can start the server by running:

```
$ cd $KC_HOME  
$ bin/kc.sh start
```

If everything is OK, you should be able to access Keycloak at `https://mykeycloak:8443`, and you should be able to see that the certificate being used is a self-signed certificate.

Alternatively, you can also provide the certificate and its corresponding private key from a PEM file by using `https-certificate-file` and `https-certificate-key-file`, respectively.

In addition to enabling HTTPS, Keycloak also allows you to define TLS constraints on a per-realm basis. Basically, for each realm, you can set whether Keycloak should require HTTPS for incoming requests:

The screenshot shows the 'General' tab selected in the navigation bar of the Keycloak configuration interface. The 'master' realm is currently active. The 'Enabled' status is set to 'Enabled'. Below the tabs, there are several configuration fields:

- Realm ID ***: master
- Display name**: Keycloak
- HTML Display name**: <div class="kc-logo-text">Keycloak</div>
- Frontend URL**: (empty input field)
- Require SSL**: External requests

Figure 9.1: Enforcing HTTPS on a per-realm basis

By default, Keycloak is going to enforce TLS for any **External requests**. That means clients using the public network can only access Keycloak through HTTPS.

Ideally, you should set the **Require SSL** setting to **All requests**, so that any request to Keycloak is guaranteed to be using a secure protocol.

In this topic, you learned how to enable HTTPS and the importance of doing so. You also learned that Keycloak allows you to define HTTPS constraints on a per-realm basis.

In the next topic, we will be looking at how to configure a production-grade database.

Configuring a database

Keycloak relies on a single database to store all its data. Even when running multiple instances of Keycloak, all of them will be talking to the same database. A database is crucial for the overall performance, availability, scalability, reliability, and integrity of Keycloak. Although Keycloak provides a caching layer to avoid database hits as much as possible, a performant database will help to make the system behave better when data needs to be loaded from the database.

In this topic, you are going to configure a PostgreSQL database. The same steps should work for any other database you choose. By default, Keycloak is configured with a very simple file-based H2 database that should not be used in production, by any means. Instead, you should configure a more robust database such as the following:

- MariaDB
- MariaDB Galera
- MySQL
- Oracle
- Microsoft SQL Server
- PostgreSQL

Make sure to check the documentation available at
<https://www.keycloak.org/server/db> for more details about

the official supported databases and their version.

The `db`, `db-url-host`, `db-username`, and `db-password` options are the bare minimum options you need to set to connect to a database. Edit the `$KC_HOME/conf/keycloak.conf` file and add the following options:

```
# Configuring a PostgreSQL database
db=postgres
db-url-host=mypostgres
db-username=keycloak
db-password=keycloak
```

The `db` option sets the database vendor and activates some default connection settings so that you only need to specify the hostname where your database is listening and the credentials. The port and other connection details are automatically resolved and used to create the JDBC URL to connect to the database.

By default, the server will try to connect to a database running on `localhost` using the default ports for a particular vendor.

However, most of the time your database is running on a host other than `localhost`, and for that you can use the `db-url-host` option to set the database host. The same goes for the `db-username` and `db-password` options in order to set the database username and password, respectively.

If everything is OK, the next time you start the server you should connect to the database you have configured.

In addition to these basic settings to connect to an external database, there are other settings you should consider before going to production. Probably one of the most important ones is the size of the connection pool; it should be sized according to the load you expect in your system, and how many concurrent requests should be allowed at a given point in time.

By default, the pool is configured with a max of one hundred connections. This value should be enough for most deployments, but if you are facing errors in logs due to connections not being available in the pool when under an unexpected load, you may change the pool size by setting the `db-pool-min-size` and `db-pool-max-size` options:

```
db-pool-min-size=150  
db-pool-max-size=150
```

In the preceding example, we are increasing the pool size to a maximum (`db-pool-max-size`) of 150 connections. We are also defining the minimum size (`db-pool-min-size`) with the same value.

In this section, you learned about the basic steps to configure a production-grade database in Keycloak. You also learned about the different databases you can use based on the list of supported databases.

In the next section, you will learn how to configure Keycloak for high availability, starting with the necessary configuration to configure a reverse proxy or load balancer.

Enabling clustering

Most of the time, you will be running a Keycloak cluster when going to production. To respect some key non-functional aspects, as well as the **Service-Level Agreements (SLAs)** defined for your services, enabling clustering is crucial.

In terms of availability, clustering allows you to run multiple Keycloak instances, possibly in different availability zones, so that uptime is not impacted if nodes (or availability zones) go down.

From a scalability perspective, clustering allows you to scale your nodes up and down according to the load on your system, helping to keep a consistent response time and throughput.

In terms of failover, a cluster helps you to recover and survive from failures, therefore preventing data loss (mainly that kept in caches) as well as avoiding impacts on general availability.

Keycloak is designed for high availability and fast response times, where, in addition to the persistent data kept in the database, it also uses a cache layer to replicate and keep state in-memory for fast data access. The clustering capabilities of Keycloak are based on a distributable cache layer built on top of **Infinispan**, a high-performance key-value in-memory data store.

To enable clustering and full high availability, you should do the following:

- Run the server using a distributed cache.
- Make sure the reverse proxy is configured to distribute load across the different instances and respect session affinity.

Clustering is enabled by default when you run the server in production mode using the `start` command. The default cache configuration is read from the `$KC_HOME/conf/cache-ispn.xml` file but you can also provide your own file by using the `cache-config-file` option.

For the sake of simplicity, let's start two server instances in the same host using different ports and the default cache configuration. To start the first instance, run the following command:

```
$ cd $KC_HOME  
$ bin/kc.sh start
```

This command will start the first instance in our cluster. The server will be listening on the default ports, and you should be able to access it at `http://localhost:8443`.

Let us now start a second instance by specifying a different port:

```
$ bin/kc.sh start --http-port=8180 --https-port=8543
```

Now, perform the same steps to start the third node as follows:

```
$ bin/kc.sh start --http-port=8280 --https-port=8643
```

After executing this last command, you should now have three Keycloak instances running on ports `8443`, `8543`, and `8643`, respectively.

By default, the nodes in a cluster are using the UDP protocol to send and receive messages and using multicast for node discovery. This communication is based on **JGroups**, a reliable and secure group communication toolkit. For this example, using UDP is simpler and enough to achieve our goals, but in a real production deployment, you might want to use a different protocol for transport and node discovery.

To choose a different communication protocol, you can use the **cache-stack** option to choose from a list of built-in protocols targeted for different environments, like when deploying Keycloak on Kubernetes. For more details about the cache configuration, look at the guide available at

<https://www.keycloak.org/server/caching>.

Keycloak relies on specific caches for failover where the state is shared across the different nodes in the cluster. One important configuration you should consider when enabling clustering is to configure how many replicas you need in your cluster and adjust it according to your failover requirements.

By looking at the `$KC_HOME/conf/cache-ispn.xml` file, you should see `distributed-cache` definitions like:

```
<distributed-cache name="sessions" owners="";  
    <expiration lifespan="-1"/>  
</distributed-cache>
```

Entries in the distributed caches are automatically replicated across a specific set of cluster nodes. Depending on your availability and

failover requirements, you might want to increase the number of **owners** to **N** – the nodes where the state is replicated – to support **N-1** node failures without losing any state. The number of owners has a direct impact on the overall performance of Keycloak in terms of network and CPU. As you add more owners, you should expect additional overhead to replicate the state across nodes. You should consider this when defining the number of owners to balance both performance and failover aspects of your deployment. By default, the number of owners is set to **2** so that the state is fully replicated to at least 2 cluster nodes, therefore avoiding losing state if a single cluster node is down.

Another important characteristic of clustering is how Keycloak caches realms data to avoid unnecessary roundtrips to the database, therefore increasing the overall performance of the server when caches are hot. By looking at the `$KC_HOME/conf/cache-ispn.xml` file, you should see the following cache definitions:

```
<local-cache name="realms">
    <encoding>
        <key media-type="application/x-java-
        <value media-type="application/x-ja
    </encoding>
    <memory max-count="10000"/>
</local-cache>
<local-cache name="users">
    <encoding>
        <key media-type="application/x-java-
        <value media-type="application/x-ja
    </encoding>
```

```
<memory max-count="10000"/>  
</local-cache>
```

In contrast with the previous caches, the `realms` and `users` caches are local caches, and their entries are not replicated but only kept in memory on each node in the cluster. The `realms` cache is responsible for caching any kind of realm data, such as clients, groups, roles, identity providers, and authentication flows. On the other hand, the `users` cache is responsible for caching any kind of user data, such as credential metadata, attributes, roles, and group mappings.

By default, Keycloak defines a maximum size of 10,000 entries for both caches. For most deployments, this limit should be enough to completely avoid roundtrips to the database when caches are hot without allocating too much memory. But depending on how much data you have in Keycloak, you might want to adjust this limit accordingly.

In this topic, you learned about the basic steps to enable clustering, where nodes will communicate with each other to share state and work together as if you were running a single instance. You also learned about the importance of clustering in terms of availability and scalability.

In the next topic, you will learn about the main configuration aspects when setting up a reverse proxy in front of a Keycloak cluster so that users can access your cluster through a public domain name with high availability in mind.

Configuring a reverse proxy

When running in production, a reverse proxy is a key component to enable high availability. A reverse proxy provides a single and public access point for the different Keycloak instances, distributing the load across them using a set of policies. These instances are usually running in a private network so that they are only reachable through the proxy.

By distributing the load across instances, a reverse proxy helps you to scale your deployment by adding or removing more instances as needed, as well as helping to survive failures when specific nodes are failing to serve requests.

Keycloak can be used with any reverse proxy vendor, so you are free to use whatever you are comfortable with. Examples of widely used reverse proxies are Apache HTTP Server, Nginx, F5, and HAProxy.

Regardless of your preference, there is a set of basic requirements that you should be aware of when setting up a proxy:

- TLS termination and re-encryption
- Load balancing
- Session affinity
- Forwarding headers

Before moving on to the next topics, make sure to update your HAProxy installation with the configuration file available from your local copy of the GitHub repository at `$KC_HOME/Keycloak---Identity-and-Access-Management-for-Modern-Applications-2nd-Edition /ch9/haproxy.cfg`:

```
$ cd $KC_HOME  
$ sudo cp Keycloak---Identity-and-Access-Management  
$ sudo cp Keycloak---Identity-and-Access-Management
```

In the next topics, we will be looking at each of the requirements mentioned here and how to address them using HAProxy.

Distributing the load across nodes

One of the first things you usually do when configuring a reverse proxy is to configure the backend nodes that are going to serve requests from clients. That is one of the main problems solved by reverse proxies. Despite the implementation you choose, you should be able to configure load balancing so that requests are distributed across these nodes using a specific algorithm for optimal throughput, response time, and failover.

Load balancing does not require any specific configuration on the Keycloak side. But here are some things to keep in mind when configuring it:

- The number of backend nodes should respect the expected load, availability, and failover scenarios.
- There are several algorithms that you can choose from to distribute the load across nodes. You should choose what works best for you after running some load tests to make sure you can achieve the desired goals in terms of response time and throughput.

In our HAProxy configuration, the configuration related to load balancing is the following:

```
balance roundrobin
server kc1 127.0.0.1:8443 check ssl verify none cool
server kc2 127.0.0.1:8543 check ssl verify none cool
server kc3 127.0.0.1:8643 check ssl verify none cool
```

In this configuration, we are defining three Keycloak instances as backend nodes as well as using `roundrobin` to distribute the requests across these nodes. We are also using HAProxy to re-encrypt connections to the backend nodes.

In this topic, you learned about the importance of load balancing and how it affects your deployment in terms of performance, availability, and failover.

In the next topic, we will be looking at how to configure your proxy to forward information about clients connecting to Keycloak.

Forwarding client information

When running behind a reverse proxy, Keycloak does not talk directly to the client that originated the request, but rather to the reverse proxy itself. This fact has an important consequence for how Keycloak obtains information about the client, such as the IP address.

To overcome this limitation, reverse proxies should be able to forward specific headers to provide Keycloak information about the

client where the request originated from. The main headers Keycloak requires from proxies are the following:

- **Forwarded**: A standard header containing all the information about the client making a request. For more details, look at <https://www.rfc-editor.org/rfc/rfc7239.html>.
- **X-Forward-For**: A non-standard header indicating the address of the client where the request originated from
- **X-Forward-Proto**: A non-standard header indicating the protocol (for example, HTTPS) that the client is using to communicate with the proxy
- **X-Forward-Host**: A non-standard header indicating the original host and port number requested by the client

Particular care should be taken when making sure the proxy is setting all these headers properly, and not just forwarding these headers to Keycloak as they are sent by clients. You should also prefer using the `Forwarded` header instead of the non-standard `X-Forwarded-*` headers.

On Keycloak, the configuration you need to integrate with a proxy is quite simple. Basically, you need to tell Keycloak that it should infer client and request information based on the headers we just discussed. For that, edit the `$KC_HOME/conf/keycloak.conf` file and add the following option:

```
proxy=reencrypt
```

In this example, we are using a proxy mode called **reencrypt** to indicate to Keycloak that the reverse proxy is using a different certificate than Keycloak and it is re-encrypting traffic to the backend nodes. For more details about the proxy configuration, look at the proxy guide available at <https://www.keycloak.org/server/reverseproxy>.

On the reverse proxy side, we have the following configuration defined:

```
option forwardfor
http-request add-header X-Forwarded-Proto https
http-request add-header X-Forwarded-Port 443
http-request add-header X-Forwarded-Host mykeycloak
```

This configuration will make sure that HAProxy sets the mentioned headers so that Keycloak can obtain information about clients making the requests.

In this section, you learned about the importance of configuring your proxy to forward client information to Keycloak through specific HTTP headers. You also learned how to configure Keycloak to respect these headers and use this information when processing requests.

In the next section, we will be looking at the importance of session affinity and its impact on the overall performance of Keycloak.

Keeping session affinity

Another important configuration you should consider is how the proxy is going to respect session affinity. Session affinity is about the proxy using the same backend node to serve requests to a particular client. This capability is especially useful when clients are using flows that require multiple interactions with Keycloak, such as when using the user agent to authenticate users through the authentication code flow.

As you learned in the *Enabling clustering* section, Keycloak tracks state about user and client interactions with the server. This state is kept in in-memory caches and shared across different nodes in the cluster. Session affinity helps to minimize the time taken by Keycloak to look up data on these caches, where clients connecting to these nodes do not need to look up data on other nodes in the cluster.

To configure session affinity, edit the `$KC_HOME/conf/keycloak.conf` file and add the following option:

```
spi-sticky-session-encoder-infinispan-should-attach
```

By doing that, Keycloak is going to rely on the proxy to keep session affinity between clients and backend nodes.

By default, Keycloak uses a different strategy for session affinity, indicating to the proxy the node to which a client should be tied. We recommend, though, to always rely on the session affinity provided by your proxy and set the `spi-sticky-session-`

```
encoder-infinispan-should-attach-route  
property to false.
```

Session affinity has a direct impact on the overall performance. As mentioned before, state is shared across the different nodes in the cluster, so keeping a client connected to a specific backend node is crucial to prevent additional network and CPU overhead.

Now, on the reverse proxy side, we have the following configuration to guarantee that clients are tied to a specific node:

```
cookie KC_ROUTE insert indirect nocache
```

With the preceding configuration, HAProxy is going to set a `KC_ROUTE` cookie where its value is the first node that the client made the request to. Subsequent requests from the same client will always be served by the same node.

In this topic, you learned about session affinity and the importance of configuring it properly in your proxy as well as in Keycloak.

In the next section, we are going to run some basic tests to make sure the configuration we've done so far is working as expected.

Testing your environment

By this point, you should have a local environment very close to what will become your production environment.

In the previous topics in this chapter, we have covered the following:

- Setting up Keycloak to use a public domain name for frontend and backend endpoints, as well as logically grouping the different Keycloak instances under a single issuer
- Setting up Keycloak to listen on HTTPS so that all traffic to and from Keycloak is secure
- Setting up Keycloak to use a production-grade database using PostgreSQL
- Setting up clustering so that multiple instances of Keycloak can share the state kept by their caches
- Setting up a reverse proxy, using HAProxy, so that we can finally access all Keycloak instances through a single public domain name

In the following topics, you are going to perform some basic tests on the environment to make sure everything is working as expected.

Before we begin, make sure HAProxy is started by running the following command:

```
$ sudo systemctl restart haproxy
```

Testing load balancing and failover

Firstly, try to access Keycloak at <https://mykeycloak> and log in to the administration console.

Depending on the browser you are using, you should be able to see which backend node is serving your requests. In Firefox, you can

open the development tools and look at the cookies sent by your browser when making requests to Keycloak:

The screenshot shows the Network tab of a browser's developer tools. The 'Cookies' tab is selected. A table lists network requests with their initiator, type, status, and size. To the right, a sidebar shows the 'Request Cookies' section, which contains three entries: AUTH_SESSION_ID: 748, AUTH_ROUTE: 'kc1', and KC_ROUTE: 'kc1'. The table data is as follows:

S...	Me	Domain	File	Initiator	T...	Transf...	S...	Headers	Cookies
200	GET	my...	step2.html	step1...	h...	1.57 KB	1...		Filter Cookies
302	GET	my...	auth?client_id=security-a	keyclo...	h...	10.72 KB	7...		Response Cookies
200	GET	my...	/auth/admin/master/cons	docu...	h...	7.62 KB	7...		Request Cookies
200	GET	my...	jquery.min.js	script	js	cached	8...		AUTH_SESSION_ID: 748
200	GET	my...	select2.js	script	js	cached	1...		AUTH_ROUTE: 'kc1'
200	GET	my...	angular.min.js	script	js	cached	1...		KC_ROUTE: 'kc1'

Figure 9.2: Looking at the cookies sent by the browser

Your browser should be sending a `KC_ROUTE` cookie where its value is the node chosen by the reverse proxy to indicate which Keycloak instance should be serving that request. From the preceding screenshot, requests should be forwarded to the `kc1` node.

Now, try to shut down the Keycloak instance to map to the `kc1` node in the HAProxy configuration file. If you see a different value for the `KC_ROUTE` cookie, you need to shut down the corresponding node.

After shutting down the node, try to refresh the administration console page. If everything is properly configured, you should still be able to access the administration console without having to authenticate again. That is only possible due to teamwork between

both the reverse proxy and Keycloak, where Keycloak makes sure the state is replicated across instances, and the reverse proxy is able to transparently forward requests to another node.

Testing the frontend and backchannel URLs

Lastly, let's check the OpenID Discovery document and look at how Keycloak is exposing its endpoints. For that, open Keycloak at <https://mykeycloak/realm/master/.well-known/openid-configuration>. As a result, you get a JSON document as follows:

```
issuer: "https://mykeycloak/realm/master"
authorization_endpoint: "https://mykeycloak/realm/master/protocol/openid-connect/auth"
token_endpoint: "https://mykeycloak/realm/master/protocol/openid-connect/token"
introspection_endpoint: "https://mykeycloak/realm/master/protocol/openid-connect/token/introspect"
userinfo_endpoint: "https://mykeycloak/realm/master/protocol/openid-connect/userinfo"
end_session_endpoint: "https://mykeycloak/realm/master/protocol/openid-connect/logout"
frontchannel_logout_session_supported: true
frontchannel_logout_supported: true
jwks_uri: "https://mykeycloak/realm/master/protocol/openid-connect/certs"
check_session_iframe: "https://mykeycloak/realm/master/protocol/openid-connect/login-status-iframe.html"
```

Figure 9.3: The OpenID Discovery document

If everything is set correctly, you should see that, regardless of the node serving the request, Keycloak will advertise all its endpoints using the <https://mykeycloak> base URL.

Summary

In this chapter, we covered the main steps to configure Keycloak for production. With the information provided herein, you should now be aware of the main steps and configuration to successfully deploy

Keycloak for high availability. You learned that when deploying Keycloak in production, you should always use a secure channel using HTTPS, as well as the importance of setting up the hostname provider to configure how Keycloak issues tokens and exposes its endpoints through the OpenID Connect Discovery document. You also learned about the importance of using a production-grade database and its impact on the overall performance and availability of Keycloak, as well as on data consistency and integrity. Lastly, you learned how to configure and run a cluster with multiple Keycloak instances and how to use a reverse proxy to distribute load across these instances.

In the next chapter, you will learn how to manage users in Keycloak, as well as integrating Keycloak with different identity stores.

Questions

1. Is the database a single point of failure?
2. Does the default clustering configuration work in whatever platform I choose to deploy Keycloak?
3. What is the best way to deploy Keycloak in Kubernetes or OpenShift?
4. How secure is the communication between nodes in a cluster?
5. Do I need HTTPS when making requests from the reverse proxy?
6. My Keycloak nodes have a high CPU usage. Is that normal?
7. How much memory does Keycloak need?
8. Is there a tool to perform load tests?

Further reading

- Production guide:
<https://www.keycloak.org/server/configuration-production>
- Clustering guide:
<https://www.keycloak.org/server/caching>
- Proxy guide:
<https://www.keycloak.org/server/reverseproxy>
- Hostname guide:
<https://www.keycloak.org/server/hostname>
- Database guide: <https://www.keycloak.org/server/db>
- HAProxy documentation: <https://www.haproxy.org/>

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/SecNet>



10

Managing Users

In the previous chapters, you learned how to deploy, run, and use Keycloak to authenticate and authorize users in your applications. You also learned how to manage users in Keycloak to run some of the examples in this book.

In this chapter, we are going to take a closer look at the capabilities provided by Keycloak that are related to identity management and federation, such as how users are created and managed, how users can manage their own accounts, how to manage credentials, and how to integrate with different identity stores and identity providers to authenticate users and fetch their information through open protocols such as OpenID Connect, **Security Assertion Markup Language (SAML)**, and **Lightweight Directory Access Protocol (LDAP)**.

In this chapter, we will cover the following topics:

- Managing local users
- Integrating with LDAP and Active Directory
- Integrating with social identity providers
- Integrating with third-party identity providers
- Allowing users to manage their data

By the end of this chapter, you will be able to leverage these capabilities to effectively manage your users, as well as understand how they can be used to solve common problems related to identity management and federation.

Technical requirements

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

<https://packt.link/wrTJi>

Managing local users

In the previous chapters, you had to create users in Keycloak to run some of the examples provided in this book. In this section, we are going to deep-dive into some key capabilities provided by Keycloak to manage your users once they are stored in Keycloak's internal database. From now on, whenever you read about a local user, you can think of it as a user stored in a Keycloak database.

As an identity management solution, Keycloak gives you several capabilities to manage user identities. In this section, we will look at the following topics:

- How to create users
- How to manage user credentials
- How to obtain and validate user information
- How to enable user self-registration
- How to extend user information using attributes

In the next section, we are going to start our journey by looking at how to create a local user in Keycloak.

Creating a local user

To create a new user in Keycloak, click on the **Users** link on the left-hand side panel. Once you've done that, you will be presented with a list showing all the users that are available in the realm. As you are creating the first user in this realm, you will be presented with a **Create new user** button instead. By clicking on this button, you will be presented with the user creation page.

When creating a new user, you are only asked for a few pieces of information. In fact, you should be able to create a new user by providing only a **Username**. Let's create a user whose username is set to **alice**:

Create user

Required user actions

Select action ▾

Username *

alice

Email

Email verified ⓘ No

First name

Last name

Groups ⓘ

Join Groups

Create Cancel

The screenshot shows a 'Create user' interface. At the top left, there's a breadcrumb navigation: 'Users > Create user'. Below it is the title 'Create user'. On the left, a section titled 'Required user actions' contains a 'Select action' dropdown with a downward arrow icon. The main area has several input fields: 'Username *' with the value 'alice', 'Email' (empty), 'First name' (empty), 'Last name' (empty), and 'Groups' with a link to 'Join Groups'. At the bottom right are two buttons: a blue 'Create' button and a light blue 'Cancel' button.

Figure 10.1: Creating a new user

Click on **Create** to create the user.

Creating a user is a trivial task. Keycloak depends on a few pieces of basic information about users while still allowing you to decorate them with additional information, as we will see later. This basic set

of information is what Keycloak needs to identify a user, correlate the user with other functionalities, and issue tokens after authenticating users.

When you create a new user, that user belongs to the realm you are managing. Users created in a realm can only authenticate through the realm they belong to.

Creating a user using the administration console is useful when the administrator has all the information about a user beforehand. However, depending on the use case, that is not always the case, so you may want to either allow your users to self-register in your realm or ask them for their information as part of the authentication process.

In this section, you learned how to create a user in Keycloak. You also learned that Keycloak depends on a few pieces of information about users so that they can authenticate in a realm. You also learned that Keycloak allows you to decorate your users with additional information and that once they are created, they can only authenticate through the realm they belong to.

In the next section, we will look at how to manage user credentials.

Managing user credentials

After creating users, they should be able to authenticate in the realm. For that, we need to set up credentials for the user. Keycloak supports different forms of authentication using different types of credentials. As we will see in *Chapter 11, Authenticating Users*, users

can authenticate in different ways, such as by using passwords, one-time keys, security devices, X.509 certificates, or any combination of these credentials.

To manage user credentials, click on the **Credentials** tab. In this tab, you should be able to see all the credentials associated with a user, as well as perform specific actions such as deleting or modifying credentials. You will also be provided with shortcuts to easily set a password (the simplest form of authentication, although not the strongest) for a user.

Keycloak does not expose sensitive data associated with credentials, only the basic and non-sensitive data associated with them. Depending on your security requirements, you may also want to encrypt data at rest in the database.

We are going to look at other types of credentials in the next chapter, but for now, let's create a password for the `alice` user that we just created. For that, click on the **Set password** button to fill in the **Password** and **Password confirmation** fields with any password you want, and turn off the **Temporary** setting. Do not worry about this setting for now as we are going to talk about it in the next chapter. Just keep in mind that by disabling it, we are creating a definitive password for the user. Click on the **Save** button to set the user's password:

Set password for alice

×

Password *

•••••

Password confirmation *

•••••

Temporary ?

Off

Save [Cancel](#)

The screenshot shows a password setting interface. At the top is a title 'Set password for alice' and a close button 'X'. Below the title are two password input fields, each showing five dots ('•••••') and a visibility icon. Underneath the first field is a 'Temporary' toggle switch, which is currently off. At the bottom are two buttons: a blue 'Save' button and a light blue 'Cancel' button.

Figure 10.2: Setting a new password for a user

After setting the password, you will see that the list of credentials has been updated with the new password we just set for the user. From that list, you can view information about any credential associated with a user – non-sensitive information – as well as delete it.

Now, let's test whether the user can authenticate in our realm. For that, we are going to try to access the account console, an application provided by Keycloak where users can manage their own information. We are going to discuss it in the following sections, but for now, just open your browser at

`http://localhost:8080/realm/myrealm/account` and click on the **Sign in** button in the top-right corner of the page. You should be redirected to the `myrealm` login page.

Type in the username and password for the user `alice` and click on the **Login** button. If everything is working properly, you should be able to access the account console as the user `alice`:

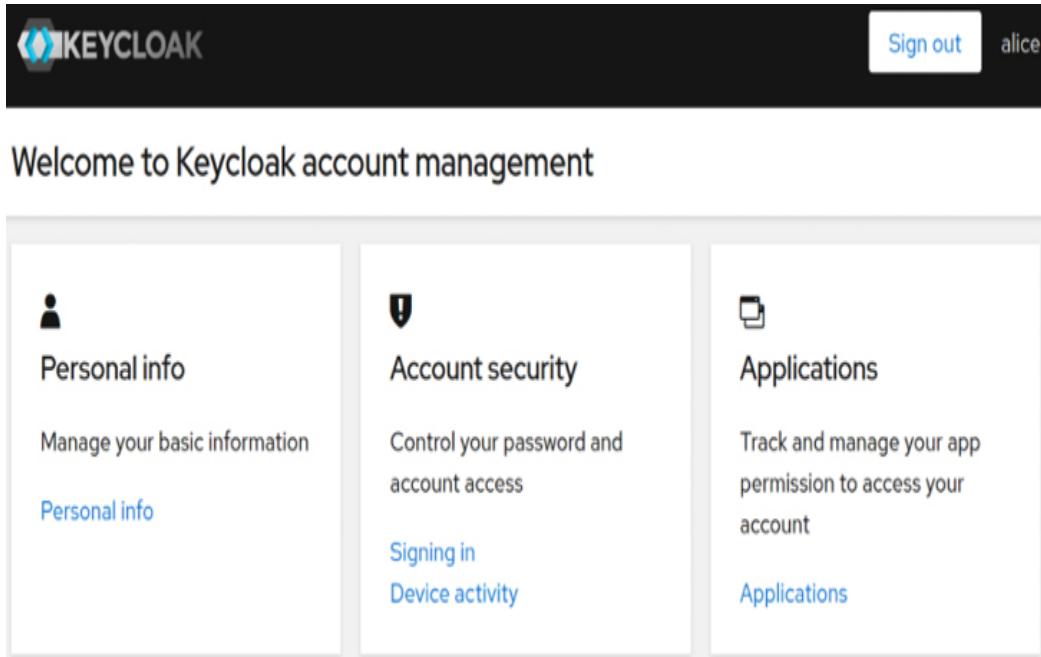


Figure 10.3: Authenticating as the newly created user

In this section, you learned how to manage user credentials through the administration console. You also learned that Keycloak allows you to manage different types of credentials, with shortcuts for managing the user password. You also learned that once created, all the credentials associated with a user – not only passwords – can be managed through the **Credentials** tab.

In the next section, we are going to look at how to interact with users during the authentication process to gather more information about them.

Obtaining and validating user information

In the previous sections, we created the user `alice` by providing only a **username**. We also set a password for this user to

authenticate in the realm. As you may have noticed, we are missing some important information about `alice`, and we want her to fill in that information to create her account.

Keycloak allows you to interact with users during the authentication process using a functionality called **Required user actions**. This setting is related to the actions that users should perform prior to successfully authenticating to a realm. Keycloak provides a good set of common actions covering different scenarios, such as the following:

- **Verify Email:** Send an email to the user – if one was set – to confirm it belongs to that user.
- **Update Password:** Ask the user to update their password.
- **Update Profile:** Ask the user to update their profile by providing their first name, last name, and email.

There are other options, but the preceding actions should give you an idea of how powerful this setting is and how you can interact with your users when they are authenticating.

Let's configure a user action to ask for more information about `alice`. For that, select `alice` from the user list and select the **Update Profile** action from the list of available actions in the **Required user actions** field. Then, click on the **Save** button:

alice

Enabled Action ▾

< Details Attributes Credentials Role mapping Groups Consents Identity provider link >

ID *	f7e948b5-01d3-4364-976d-1fd539421fca
Created at *	2/17/2023, 8:49:16 PM
Required user actions	Update Profile Select action X ▼
Username *	alice
Email	
Email verified ?	<input checked="" type="radio"/> No
First name	
Last name	

[Save](#) [Revert](#)

Figure 10.4: Forcing users to update their account with any missing information

Now, let's try to access the account console as `alice`. For that, open your browser at

`http://localhost:8080/realm/myrealm/account` and click on

the **Sign in** button in the top-right corner of the page. You should be redirected to the **myrealm** login page.

Type in the username and password for **alice** and click on the **Login** button. If everything is working properly, you should be redirected to a page asking for the information we are missing for that user.

Fill in all the fields and click on the **Submit** button:

Update Account Information

⚠ You need to update your user profile to activate your account.

Email

First name

Last name

Submit

Figure 10.5: Asking a user to update their account information

Once you submit the information, Keycloak is going to update the user with the information provided. The updated information should now be available when you access the account console.

The same idea applies to any other required action you set for a user, where each is related to specific steps that the user is required to complete before authenticating to a realm. For instance, if you set the **Update Password** action, the user is going to be asked to change their password, whereas the **Verify Email** action is going to make sure the email associated with the user is valid through an email verification process.

In this section, you learned how Keycloak allows you, as an administrator, to interact with your users to obtain and validate information about their accounts.

In the next section, we are going to look at how to allow users to self-register their accounts.

Enabling self-registration

Depending on your requirements, you might want to allow users to self-register in a realm and delegate to them the responsibility of filling in their information. Compared to manually creating a user, Keycloak is going to provide a link on the login page for user self-registration.

For that, click on **Realm Settings** in the left-hand side menu and then click on the **Login** tab. In this tab, enable the **User registration** option.

Now, let's create a new user by going through the self-registration process. Open your browser at

`http://localhost:8080/realm/myrealm/account` and click on the **Sign in** button in the top-right corner of the page. You should

be redirected to the `myrealm` login page. At the login page, click on the **Register** link:

The screenshot shows a login interface. At the top, the text "Sign in to your account" is displayed. Below it is a form with two input fields: "Username or email" and "Password". A large blue button labeled "Sign In" is centered below the inputs. At the bottom of the page, there is a light gray bar containing the text "New user? [Register](#)".

Figure 10.6: Allowing users to sign up to a realm

Once you've done that, you should be presented with a registration page, asking you to provide the same information that you did when you created users through the administration console. Fill in the fields with any information you want and click on the **Register** button to create a new user.

Now, go back to the Keycloak administration console and check whether the user you just created is shown in the user list. If everything is correct, you should be able to see the user you just created in that list.

Self-registration is a powerful feature and a must-have for certain use cases where users should be allowed to sign up to a realm. It also provides the necessary level of flexibility so that you can customize the registration page to obtain additional information about users, such as their mobile numbers or addresses, according to your needs. We are going to talk about customization in *Chapter 13, Extending Keycloak*.

In this section, you learned how to enable self-registration for a realm so that users can create their own account in a realm without any intervention from an administrator.

In the next section, we will look at how to manage additional information about users.

Managing user attributes

Keycloak allows you to manage additional metadata about users using attributes. As you learned in the previous sections, Keycloak relies on a basic set of information to identify and authenticate users. This information is also made available when you're introspecting tokens or accessing a user's profile. To manage the attributes of a user, select the respective user from the user list and click on the **Attributes** tab. Each attribute has a key – the name of the attribute – and a text value.

User attributes can solve different types of problems, from passing additional information about users to applications to enabling different forms of authorization, such as **attribute-based access control (ABAC)**.

When using attributes, you are probably going to need to create protocol mappers so that they can be mapped to tokens to make them available to applications, or even when querying the `introspection` token and `userinfo` endpoints.

When extending Keycloak, as we are going to see in *Chapter 13, Extending Keycloak*, you should also be able to extend the account console to populate user accounts with additional information using attributes. The same goes for customizing the update profile page, which is shown to users during the authentication process, as we learned in the previous sections. Here, you can store custom information that was gathered during this step using attributes.

In this section, you learned how to extend Keycloak's user model by using user attributes. You also learned that by leveraging user attributes, you can extend different parts of Keycloak to obtain additional information from users and store it as user attributes. You also learned that user attributes are commonly used to pass additional information about users to applications using protocol mappers.

In the upcoming sections, we are going to look at how to integrate with third-party identity providers and identity stores to manage users from sources other than a Keycloak database. We will start by learning how to fetch user information from LDAP directories.

Integrating with LDAP and Active Directory

Many organizations still use an LDAP directory as their single source of truth for digital identities. Keycloak allows you to integrate with different LDAP server implementations so that you can leverage your existing security infrastructure and use all the authentication and authorization capabilities provided by Keycloak.

Keycloak can integrate using LDAP in different ways; it can act as a stateful broker where data from your LDAP directory is imported into the Keycloak database, as well as kept in sync with your LDAP directory, or it can act as a stateless broker delegating credential verification to your LDAP directory. You should also be able to set up multiple LDAP directories within a single realm and configure a priority order that Keycloak should respect when authenticating users.

In Keycloak, the term “user federation” refers to the capability to integrate with external identity stores. LDAP is a form of user federation and, as such, can be configured by clicking on the **User Federation** link on the left-hand side menu of the administration console.

To configure a new LDAP server, click on the **Add Ldap providers** button:

User federation

User federation provides access to external databases and directories, such as LDAP and Active Directory. [Learn more](#)

To get started, select a provider from the list below.

Add providers



Add Kerberos providers



Add Ldap providers

Figure 10.7: Creating a new LDAP user federation provider

After clicking on the **Add Ldap providers** button, you will be presented with a page containing all the settings you'll need to integrate with an LDAP directory. Keycloak supports the most common LDAP vendors, including Active Directory. The **Vendor** field allows you to choose one of these vendors and Keycloak does its best to find the best default setting for the vendor you choose.

If you are interested in integrating Keycloak using LDAP, you should become familiar with most of the settings on this page, mainly those related to connection settings and the structure of the LDAP directory. Here, we will focus on the additional settings provided by Keycloak to customize the integration, starting with the **Import Users** setting.

The **Import Users** setting allows you to define whether Keycloak should import data from your LDAP server into the server

database. By default, this setting is enabled so that whenever users authenticate through an LDAP provider, the information about that user is locally persisted, just like if you were creating the user manually or through self-registration. By having users from LDAP created locally, you can leverage all of Keycloak's identity and access management capabilities; otherwise, you would only be able to use it as a broker to authenticate your users using the LDAP directory.

Basically, when a user tries to authenticate using a password, the password validation is performed against the LDAP directory. Upon successful authentication, Keycloak will create the user locally on its database. Once imported, the user is considered a federated user, and a link is created between the user and the LDAP provider.

This link between the user and the LDAP provider is a key aspect of user federation. By looking at the link the user has to a specific user federation provider, such as LDAP, Keycloak is able to differentiate whether a user is a **local user** or a **federated user**. In this context, the term “federated” refers to the trust that’s created between Keycloak and an external identity store – in this case, the LDAP directory – so that both can share identity and access management data. The same concept also applies when you are integrating Keycloak with an existing identity store from your organization, as we’ll see in the next chapters.

Keycloak provides some key synchronization settings for managing how data is read and written back to an LDAP directory. Before you do anything else, you should decide which synchronization strategy

you want through the **Edit Mode** setting. You can choose from three different strategies: **READ_ONLY**, **WRITABLE**, and **UNSYNCED**.

The **READ_ONLY** strategy allows you to use your LDAP directory in read-only mode, where changes to federated users are not possible and are not replicated back to the LDAP directory. On the other hand, the **WRITABLE** strategy is a powerful strategy that allows you to replicate any change that's made to federated users back to the LDAP directory. The **UNSYNCED** mode is similar to **READ_ONLY** but it allows changing federated users locally without replicating the changes to the **LDAP** directory. Whether you should use a **READ_ONLY** or **WRITABLE** strategy depends on your use case. Under some circumstances, LDAP is the single source of truth for identities in your organization, which you do not have much control over. If you are using Keycloak to modernize your security infrastructure while still centralizing identity management in your LDAP directory, then the **READ_ONLY** strategy makes sense.

However, if you have plans to migrate from LDAP and replace it with a more modern identity and access management solution, then the **WRITABLE** strategy should help you during that journey. It should also allow you to keep your LDAP active and in sync with the changes that are made through Keycloak.

Once you've decided which strategy works best for you, you can look at additional settings provided by Keycloak to control how synchronization should happen. Keycloak allows you to synchronize user information by manually triggering the synchronization process through the administration console, or by

scheduling a time when synchronization should happen automatically.

To trigger a manual synchronization for user information, you should click on the **Action** button in the top-right corner of the page to choose between synchronizing all users or only those that have changed since the last synchronization by clicking on the **Sync all users** or the **Sync changed users** buttons, respectively. In fact, it is a good practice to run a full synchronization right after creating your provider.

The reason for this is that this step is going to help you to avoid importing users when they authenticate for the very first time, saving some precious time during the authentication flow.

You are also given other useful actions such as removing users that have been imported from the LDAP provider, or just removing the link between users and the LDAP provider, effectively changing the user to a regular local user.

Passwords are not stored in the Keycloak database when importing users from the LDAP directory. When unlinking users, they are no longer able to authenticate to Keycloak using a password as their password is not yet set. In this case, users should choose to reset their password during login or administrators can set a temporary password. For more details, consider looking at *Chapter 11, Authenticating Users*.

Once your users have been imported, you can schedule periodic synchronization by choosing the strategy that best suits your needs.

A full sync means that Keycloak is going to check the LDAP directory for changes that need to be replicated to its local database, so that new users that are created in LDAP, as well as updated information about those users, are kept in sync.

A partial sync means that Keycloak is going to look up new users and make changes to existing users based on the last synchronization time, hence helping to keep Keycloak updated using a more efficient strategy.

In this section, you learned about user federation and how Keycloak can integrate with existing LDAP servers to authenticate users and synchronize information.

In the next section, you will learn about LDAP mappers and their importance in terms of fetching additional information from an LDAP directory, as well as the behavior of the integration.

Understanding LDAP mappers

Just like users, Keycloak can also fetch other types of information from LDAP. Different to how you fetch users from LDAP – which is part of the core functionality of an LDAP provider – this information is fetched using mappers.

An LDAP mapper is a special, and powerful, functionality in Keycloak for mapping information from LDAP into Keycloak, and vice versa.

It provides another extension point to LDAP's integration and fine-grained control over how to read and write LDAP data for users, groups, roles, certificates, or even information that is only available when you're using a specific LDAP vendor, such as Active Directory. Whenever you need to map a specific set of data from LDAP, you should go through the list of supported mappers and find one that suits your needs.

When creating a new provider, Keycloak automatically configures a set of mappers, depending on the configuration of the provider. For instance, depending on the import mode or the edit mode, a different set of mappers is created. That is why deciding on these two settings is important prior to creating the provider. Otherwise, you would need to change mappers accordingly when changing these settings once a provider has been created.

To manage the mappers associated with an LDAP provider, click on the **Mappers** tab of the provider you are managing. On this tab, you are presented with a list of all the mappers that are currently active for the provider. From this page, you can also associate new mappers with your provider. You can do this by clicking on the **Add mapper** button at the top of the list.

There are several types of mappers that you can use, with each being specific to a particular task. In the next few sections, we are going to look at how to create mappers to manage group and role data from LDAP.

Synchronizing groups

To manage group data from LDAP, click on the **Add mapper** button at the top of the mapper list.

Type in a name for the new mapper and select **group-ldap-mapper** from the **Mapper Type** field.

Some of the settings on this page are specific to how groups are organized in the LDAP tree. For instance, you must provide the **Distinguished Name (DN)** of where your groups are located, the attribute that will be used to fetch the group name, as well as how membership information is defined in your LDAP tree so that Keycloak can automatically discover the groups that users belong to.

The **group-ldap-mapper** type gives you several settings to configure how groups should be fetched from LDAP and how synchronization should happen. Some of the settings are specific to how groups are organized in the LDAP tree.

The first step when creating this mapper is to set the location of the groups in your LDAP tree. For that, you should fill in the **LDAP Groups DN** field with the base DN where all your groups are located.

You should also be able to provide an additional filter if you have a more complex LDAP tree so that groups are fetched based on some custom criteria. For that, you can set a filter using the **LDAP Filter** field.

Keycloak is going to look up group entries from the base DN based on the object classes you've defined for group entries. You can set the object classes using the **Group Object Classes** field.

The next step is to configure how Keycloak should map information from LDAP group entries to Keycloak.

The name of a group can be mapped from these entries using the **Group Name LDAP Attribute** field. You can change this field to whatever LDAP attribute you are using to store the group's name. Usually, the **Common Name (CN)** attribute is used for this purpose.

Now that you understand how to look up groups from LDAP and map their information to Keycloak, it is time to understand how to map group hierarchy and user membership.

In LDAP, groups are usually organized in a hierarchy to represent your organizational tree. Keycloak allows you to map and preserve the group hierarchy by automatically creating it when you're fetching groups from LDAP. The first step is to set the attribute that was used to infer the relationship between the groups in the hierarchy using the **Membership LDAP Attribute** field. Keycloak is going to look up the children of a group by looking at the value of this attribute. Its format is usually the **Fully Qualified Name (FQN)** of another group entry.

You should still be able to set a different format for the membership attribute if you are still relying on `memberUid` to reference another group in the LDAP tree. For that, choose **UID** in the **Membership Attribute Type** field.

Keycloak also allows you to map user membership from LDAP so that when you're importing users, they are automatically assigned to the groups they belong to. For that, you can set different strategies for how this relationship is created in Keycloak. The **User Groups Retrieve Strategy** field allows you to choose whether user membership should be fetched based on the `member` attribute of groups – similar to when fetching the group hierarchy – or whether membership should be fetched based on the presence of another attribute within the user entry in LDAP – usually, this is the `memberOf` attribute.

Regarding synchronization, the mapper allows you to have fine-grained control over how group information is kept in sync with your LDAP directory, as well as how groups should be imported into Keycloak.

If you are using a writable LDAP provider, the mapper defaults to writing back any changes you make to groups that have been imported from LDAP, including user membership. This behavior is managed through the **Mode** field, which provides different strategies on how group information should be imported and synced back to LDAP.

By default, groups that have been imported from LDAP are created as top-level groups in Keycloak. Sometimes, it might be useful to import groups into a specific group in Keycloak to differentiate them from local groups. For that, you can set the **Groups Path** field to any existing group you have in Keycloak.

In this section, you learned how to map group information using the `group-ldap-mapper` mapper. You also learned that Keycloak is

very flexible regarding how this data is fetched and kept in sync with LDAP.

In the next section, we will be looking at how to map roles from LDAP.

Synchronizing roles

Like groups, roles are also mapped from LDAP using a specific mapper. To import role data, click on the **Add mapper** button at the top of the mapper list.

Type in a name for the new mapper and select `role-ldap-mapper` from the **Mapper Type** field.

As you can see, the core settings for role mapping are pretty much the same ones that you learned about in the previous section.

Mainly, those related to configuring how Keycloak is going to look up entries in your LDAP tree.

In this section, we are going to focus on the behavior and the specific properties related to how Keycloak will map role information from LDAP.

Roles are automatically imported from Keycloak whenever the user authenticates in Keycloak. Keycloak also allows you to manually trigger a synchronization once you've created the mapper.

When importing roles, Keycloak defaults to creating these roles as realm roles, where the roles are automatically granted to users. This behavior is controlled by the **Use Realm Roles Mapping** field, which can also be disabled so that imported roles are created as client roles for a specific client in Keycloak.

In this section, you learned how to integrate Keycloak with LDAP and how users, groups, and role information can be obtained from it. You also learned that Keycloak is very flexible regarding mapping different types of information from LDAP through a functionality called LDAP mappers.

Lastly, you learned that Keycloak gives you fine-grained control over how data is imported, as well as how data is replicated back to LDAP whenever you make changes to the information that's imported from LDAP.

In the next section, we are going to look at how to integrate with third-party identity providers by leveraging Keycloak as an identity broker to authenticate and replicate their users.

Integrating with third-party identity providers

Keycloak can integrate with third-party identity providers using a set of open standard protocols.

In the previous section, you learned about user federation and how to easily integrate with LDAP. Now, you are going to learn about brokering and how to leverage user federation to create cross-domain trust between Keycloak and an identity provider using standard authentication protocols, where the identity data about users is shared and used by Keycloak to create, authenticate, and authorize users.

Integration with third-party identity providers is made possible by using Keycloak as an identity broker, where Keycloak acts as an

intermediary service for authenticating and replicating users from a targeted identity provider.

Identity brokering can solve different types of problems. As we will see in the next section, it can be used to integrate with social providers, as an integration point for a legacy identity and access management system, or to share identity data between a business partner and your organization.

In Keycloak, you can integrate with two main types of identity providers, depending on the security protocol they support:

- SAML v2
- OpenID Connect v1.0

Through identity brokering, you can provide a much better experience for users, where they can leverage an existing account to authenticate and sign up in your realm. Once these users have been created and their information has been imported from the third-party provider, they become users of your realm and can enjoy all the features provided by Keycloak and follow the security constraints imposed by your realm.

In this section, we are going to look at how to create an OpenID Connect v1.0 identity provider. For simplicity, we are going to use another realm in the same Keycloak server to represent the third-party identity provider we are trying to integrate with. However, the same concepts and steps you are about to learn should be valid for any other OpenID Connect-compliant identity provider.

Creating an OpenID Connect identity provider

Firstly, create a realm in Keycloak called **third-party-provider**. In this realm, create a client with the following settings:

- **Client ID:** broker-app
- **Client authentication:** ON
- **Root URL:**

`http://localhost:8080/realm/myrealm/broker/oidc/endpoint`

Make sure to keep a note of the client secret that's generated as we are going to use it later when we configure the identity provider.

Now, create a user called **third-party-user** in the **third-party-provider** realm and make sure to set a password for them.

Now, let's create a new identity provider in the **myrealm** realm. For that, click on the **Identity Providers** link in the left-hand side menu:

Identity providers

Identity providers are social networks or identity brokers that allow users to authenticate to Keycloak. [Learn more](#)

To get started, select a provider from the list below.

User-defined:



Keycloak OpenID Connect



OpenID Connect v1.0



SAML v2.0

Social:



BitBucket



Facebook



Github



GitLab



Google



Instagram



LinkedIn



Microsoft



OpenShift v3

Figure 10.8: Creating a new identity provider

The **Identity providers** page allows you to either create a new provider or list all the providers that have been configured for a realm. If the realm does not have an identity provider yet, you will be prompted to create one.

Keycloak allows you to choose from a vast list of built-in identity providers. You can choose to integrate with an identity provider

using standard protocols such as OpenID Connect v1 or SAML v2, or choose from the vast list of social providers such as LinkedIn, Instagram, GitHub, Google, and many others.

Let's select **OpenID Connect v1.0** from the list of providers. After that, you should be redirected to the **Provider settings** page.

On this page, you have a read-only **Redirect URI** field, whose value is the URL we set for the `broker-app` client as a **root URL**. This URL is the location where users are going to be redirected once they've authenticated through the identity provider. In our case, users are going to be redirected back to the `myrealm` realm after successfully authenticating through the identity provider.

As you learned in *Chapter 4, Authenticating Users with OpenID Connect*, an **OpenID Connect Provider (OP)** advertises the endpoints that can be used to interact with them through a document available from a specific endpoint. By using this endpoint, we can quickly configure our identity provider, since most of the settings on the **Provider settings** page are going to be filled in automatically with the information from the OP you are integrating with.

On the **Provider settings** page, there is a **Discovery endpoint** field that you should set with the location where the OP is exposing its discovery document. In our case, this document is located at

```
http://localhost:8080/realm/third-party-provider/.well-known/openid-configuration.
```

After setting the URL, click on the **Show metadata** link. Once you've done that, you should see that some of the other fields on this page were automatically populated with the information from the discovery document.

Now, let's fill in some additional information to finish configuring the provider. For that, fill in the following fields:

- **Display Name:** My Third-Party Provider
- **Client Authentication:** Client secret sent as post
- **Client ID:** broker-app
- **Client Secret:** <CLIENT_SECRET>

Finally, click on **Create** to create the identity provider.

Note that both the **Client ID** and **Client Secret** fields refer to the `broker-app` client in the `third-party-provider` realm. This is the client used by the identity provider to authenticate users in that realm.

Now, let's test whether new users can authenticate and sign up to our realm using the newly created provider. For that, open your browser at `http://localhost:8080/realm/myrealm/account` to access the account console and click on the **Sign in** button in the top-right corner of the page. You should be redirected to the `myrealm` login page:

Sign in to your account

Username or email

Password

Sign In

Or sign in with

My Third-Party Provider

New user? [Register](#)

Figure 10.9: Login page with an option to authenticate using an identity provider

Note that you will be presented with the option to authenticate with **My Third-Party Provider**, which is the provider we just created.

Click on the **My Third-Party Provider** button; you should be redirected to the `third-party-provider` realm to authenticate. On the login page, provide the username and password for the `third-party-user` user to log in.

If everything is working properly, you should be redirected to a page asking for additional information about the user. Fill in all the fields and click on the **Submit** button.

Once the user has been authenticated to the third-party provider, a set of tokens will be issued to Keycloak. These represent the user's identity and the permissions that have been granted to the user when authenticating to the `third-party-provider` realm. By looking at these tokens, Keycloak is capable of fetching user information and creating or updating the user in your realm.

Now, if you list the users that are available in the `myrealm` realm, you should see that the `third-party-user` user is among them. This means you can manage them just like any other user in your realm. However, unlike other users, they don't have a password yet, and can only authenticate through the third-party provider.

There are several settings you can choose from when configuring a provider. For instance, Keycloak can be used to store tokens that have been issued by an identity provider, a useful capability if you need to use these tokens to access APIs protected by the provider. Once stored, Keycloak allows you to obtain these tokens through another functionality called the **token exchange**.

Keycloak also allows you to define a specific authentication flow when the user is authenticating for the first time using an identity provider. This is a powerful feature that allows you to gather additional information about your users or even force them to set up credentials.

Depending on your requirements, you can also configure your realm to only allow users to authenticate and link their accounts with an identity provider through the account console. This is achieved by turning on the **Account Linking Only** field. This ensures that users can't select the identity provider on the login page, only when they're in the account console.

There are many other settings you can choose from, and you can find them in the Keycloak documentation at

https://www.keycloak.org/docs/latest/server_admin/#_identity_broker.

In this section, you learned how Keycloak can be used to integrate with third-party identity providers using open standard protocols. You also learned that Keycloak allows you to quickly integrate with any identity provider using either the OpenID Connect or SAML2 protocols. You also learned that this integration is possible since you can use Keycloak as an identity broker, where users are authenticated and created based on the information that's returned by these providers.

In the next section, you will learn about how to extend the concepts presented in this chapter to integrate your realm with different social providers.

Integrating with social identity providers

A common requirement for applications that use Keycloak is the possibility to authenticate users using different social providers,

such as Google, GitHub, Instagram, and Twitter.

Integration with social providers follows the same principles that you learned about in the previous section, where Keycloak acts as a broker to authenticate and exchange identity data about users using a well-known and open standard security protocol.

To integrate with a social provider, click on the **Identity Providers** link in the left-hand side menu.

Keycloak allows you to select from different social providers. To integrate with them, you only need to fill in some information, which you usually obtain from the social provider you are integrating with.

Let's configure GitHub as a social provider to allow users to authenticate using their GitHub account. Firstly, make sure you have a valid GitHub account. If not, you can create one at <https://github.com>.

Now, let's create a GitHub social provider in our realm by selecting **GitHub** from the list of available providers. Once you've selected GitHub, you should be presented with a page containing a few settings that you need to fill in to create the provider.

To use GitHub, we need to create an OAuth app at <https://github.com/settings/developers>. When creating the application, you will be asked to provide an authorization callback URL or redirect URL. This URL is the endpoint in Keycloak that is going to receive the response from GitHub once the user is successfully authenticated – or when an error occurs. When creating a social provider in Keycloak, you are given a **redirect URI**,

which you should use to configure the application in GitHub. This URI is available from the **Redirect URI** field. Copy and paste the value of this field and use it to create the app in GitHub.

Depending on the social provider you are integrating with, you are going to be asked for additional information to make the integration possible. For GitHub, we need a client ID and client secret, both of which are provided by GitHub once you've created your app. Fill in both the **Client ID** and **Client Secret** fields with the values you got from GitHub.

Now, click on the **Create** button to create the provider.

Now, let's test whether the users in our realm can authenticate and sign up to our realm using the newly created provider. For that, open your browser at

`http://localhost:8080/realm/myrealm/account` to access the account console and click on the **Sign in** button in the top-right corner of the page. You should be redirected to the `myrealm` login page.

You should now be at the login page of Keycloak. Note that you will be presented with the option to authenticate with GitHub, the provider we just created. Click on the **GitHub** button to be redirected to GitHub to authenticate.

After authenticating with GitHub, you might be presented with a consent page, asking you to grant permissions to your realm for accessing information about the user. After approving the consent, Keycloak is going to create a user based on the information that was obtained from GitHub. You should be automatically authenticated and redirected to the account console.

In this section, you learned how Keycloak makes it easier to integrate with different social providers. By integrating with GitHub, you learned about the basic steps and concepts around integrating with any other social provider that supports the OpenID Connect or OAuth2 protocols.

In the next section, we are going to learn about how users can manage their data using the Keycloak account console.

Allowing users to manage their data

In the previous sections, you learned how to manage users through the admin console as an administrator. You also learned that users can self-register in a realm. However, one of the main capabilities of Keycloak is to also allow users to manage their own accounts through a service called Keycloak account console.

The Keycloak account console is a regular application provided by Keycloak and is where users can manage their own accounts. They can also do the following:

- Update their user profile
- Update their password
- Enable second-factor authentication
- View applications, including what applications they have authenticated to
- View open sessions, including remotely signing out of other sessions

To access the account console, open <http://localhost:8080/realm/myrealm/account/> in a browser. You will be redirected to a welcome page, as follows:

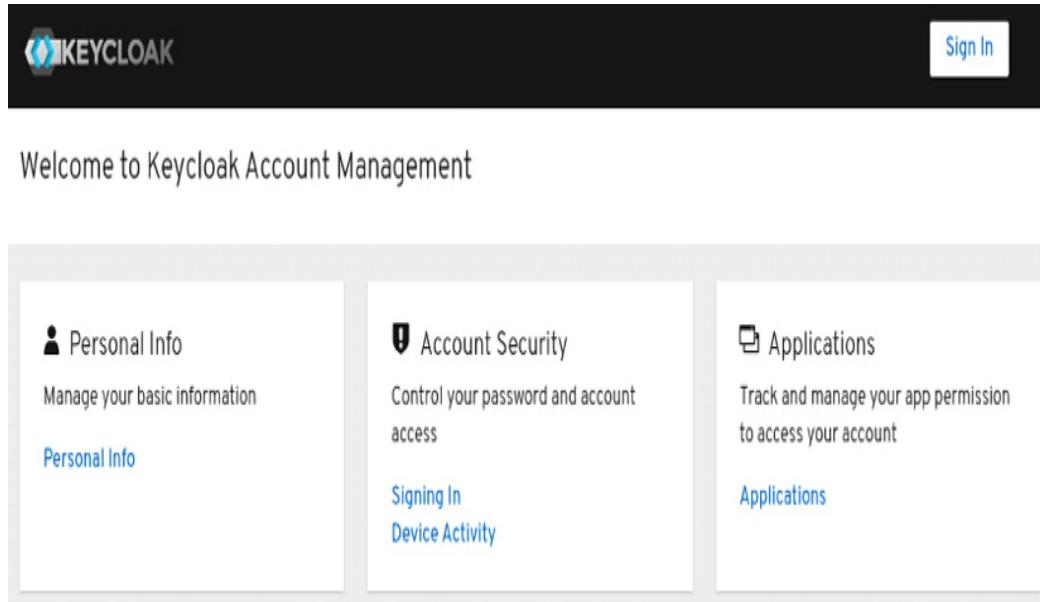


Figure 10.10: The Keycloak account console

To log in to the account console, you should either click on any of the links on that page or click on the **Sign In** button in the top-right corner of the page. By doing any of these things, you should be redirected to the login page and, after providing the user credentials, you should be redirected back to the account console:

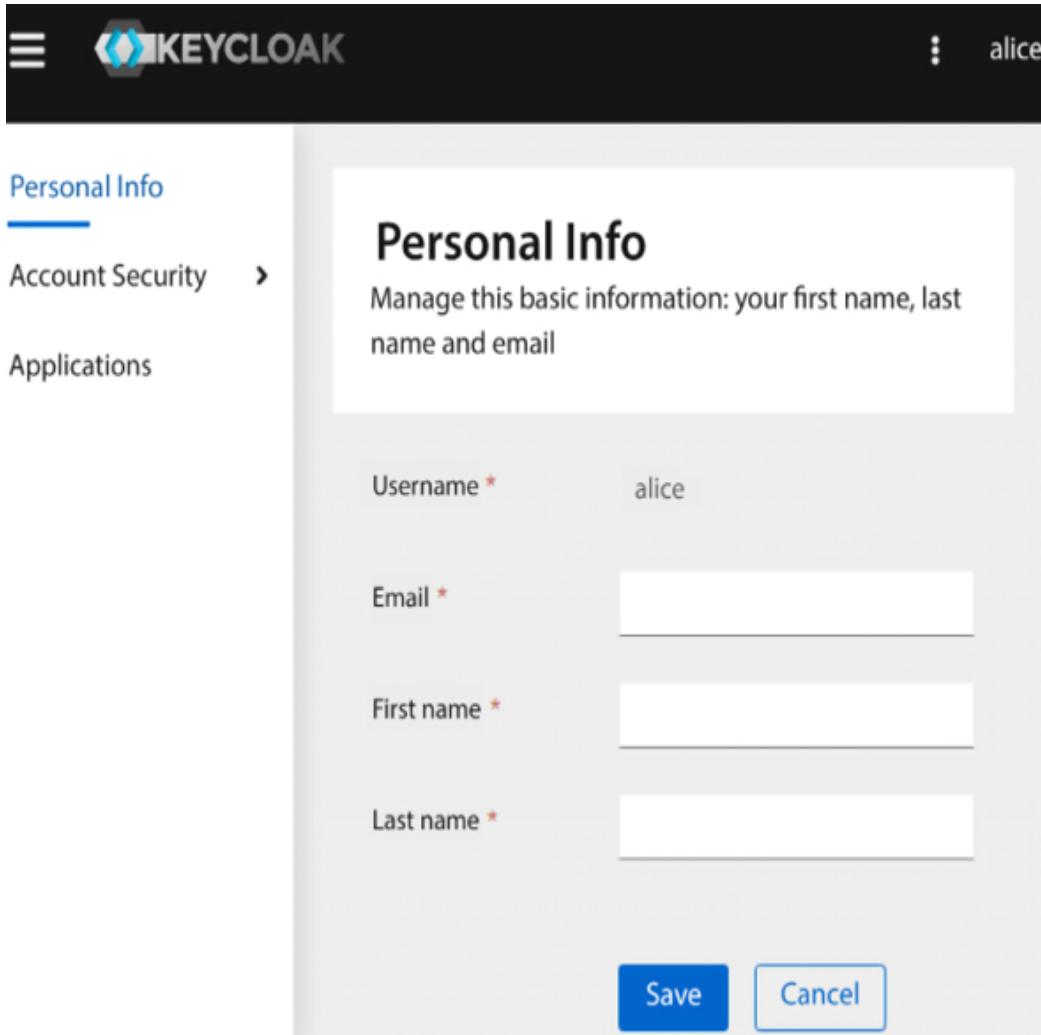


Figure 10.11: Authenticating to the account console

Once authenticated, users can view and manage different information about their accounts:

- **Personal Info:** Allows users to manage profile information such as email, first name, and last name.
- **Account Security:** Allows users to manage their credentials, as well as set up two-factor and multi-factor authentication using OTP and security devices, respectively. In this area, users should also be able to track their account activity.

- **Applications:** Allows users to manage the applications they have logged in and logged out from, as well as the permissions that have been granted to applications.

Keycloak automatically creates an **account-console** and **account** client in your realm to authenticate and authorize access to the user's account, respectively.

If you need to change how users authenticate to the account console, you can change the **account-console** client settings according to your needs. For instance, you might want to ask users to consent to the use of their data in order to comply with privacy requirements. This is the client that will be serving the user interfaces and driving users to manage their accounts.

On the other hand, the **account** client allows you to authorize access to the user data. Users created in a realm are automatically granted a **manage-account** client role. This role – and a few others – belongs to the **account** client and controls whether a user should have access to the data through the Account REST API. For instance, to disable access to this API, change the role mappings for the user to remove this role.

You can also find the URL of the account console through the Keycloak admin console. In the admin console, click on **Clients**. From there, you will find the URL of the account console next to the **account** client.

As we will see in the following chapters, Keycloak allows you to customize the look and feel of the account console, as well as how information is presented and managed through it, allowing you to obtain and store additional information from users.

In this section, you learned how users can manage their accounts through the account console. You also learned that Keycloak allows you to control what users can view and manage in their accounts, by either relying on your realm settings or by managing the roles that have been granted to the user.

Summary

In this chapter, you were presented with the main aspects of user management in Keycloak. You learned that users can be created directly in Keycloak or by integrating with different third-party identity providers and external identity stores. You also learned that Keycloak enables these integrations by leveraging open standard protocols such as OpenID Connect, SAML, and LDAP. You also learned that users are provided with capabilities to sign up for a realm, either by enabling self-registration to a realm or by integrating with a third-party provider. Finally, you learned that users can also manage their accounts through the Keycloak account console.

In the next chapter, we are going to look at how users can authenticate using different credentials, as well as how Keycloak is the perfect fit for strong authentication.

Questions

1. Can I integrate my own user database with Keycloak?
2. Does Keycloak query the LDAP directory every time the user authenticates?
3. How do I differentiate users that have been created using a third-party or social identity provider?

Further reading

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

- Keycloak user management:
https://www.keycloak.org/docs/latest/server_admin/#user-management
- Keycloak user federation:
https://www.keycloak.org/docs/latest/server_admin/#user-storage-federation
- Keycloak identity brokering:
https://www.keycloak.org/docs/latest/server_admin/#identity_broker
- Keycloak account console:
https://www.keycloak.org/docs/latest/server_admin/#account-service

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/SecNet>



11

Authenticating Users

In the previous chapters, you learned how to manage users. You also walked through examples that involved users authenticating in Keycloak. By now, you should be aware of how easy it is to set up Keycloak to promptly authenticate your users, but there is much more to authentication than just using a login page and asking users for passwords.

Keycloak has a set of well-defined flows representing how end users and clients – the actors – interact with the server when authenticating to a realm. For end users, these flows usually involve using the browser as an intermediary, and for clients, the steps are based on backchannel requests to the token endpoint.

As you learned in the previous chapters, the end users authenticating to a realm are presented with a login page. From this page, users can start different interactive flows with the server in order to:

- Self-register to the realm
- Authenticate
- Reset their password

For each of these flows, Keycloak provides a built-in and ready-to-use flow **definition** with the most common steps that users should

follow when interacting with a realm throughout any of these flows, while still enabling you to create your own definitions to better address your self-registration, authentication, and password recovery requirements.

In this chapter, we are going to take a close look at how to manage **authentication flows** and how to enable strong authentication to a realm by leveraging **two-factor authentication (2FA)** and **multi-factor authentication (MFA)**.

For that, you will learn about the different types of credentials you can choose from to authenticate users and how they work together to increase the overall security of your system.

In this chapter, we will cover the following topics:

- Understanding authentication flows
- Using passwords
- Using **one-time passwords (OTPs)**
- Using **Web Authentication (WebAuthn)**
- Using strong authentication

Technical requirements

Before we begin, create a `myrealm` realm and a user called `alice` in this realm.

In the next few sections, we will be using the Keycloak account console to authenticate `alice` using the different authentication strategies.

Check out the following link to see the **Code in Action** video:
<https://packt.link/iuHcj>.

Understanding authentication flows

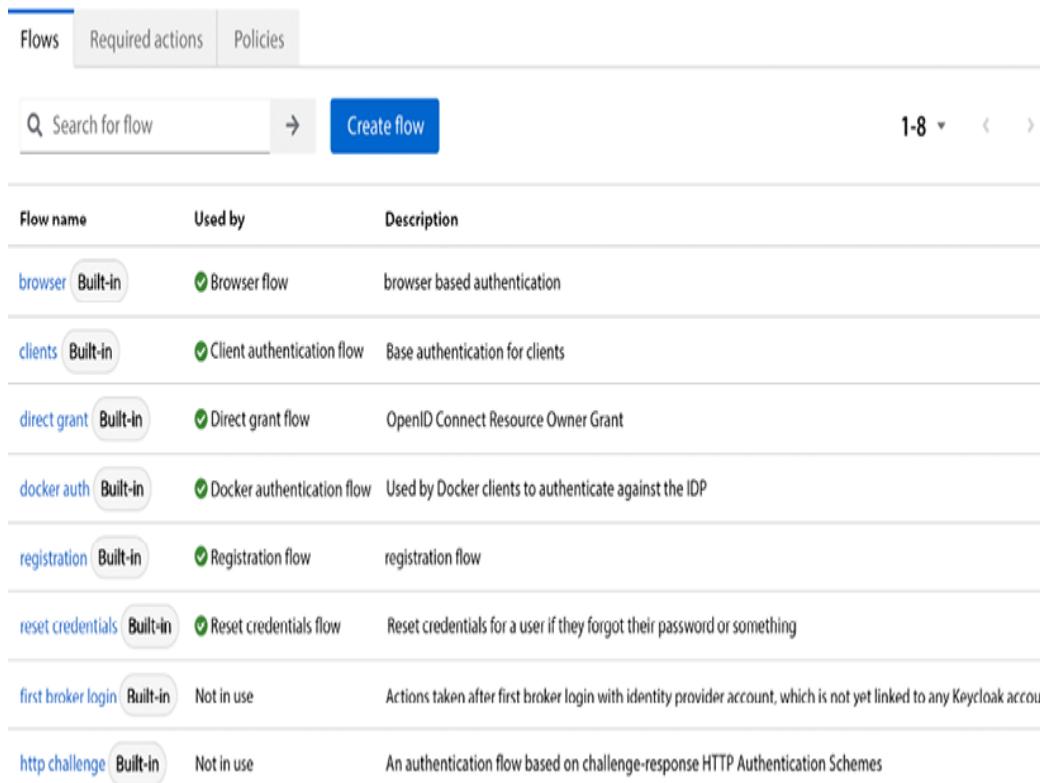
An authentication flow is driven by a set of sequential steps or executions that are grouped together to define how users and clients are authenticated.

Keycloak is very flexible in terms of how to arrange executions in an authentication flow definition. By default, realms are created with built-in definitions that cover the most common steps to securely authenticate end users and clients, which you can change or extend at any time to address your own authentication requirements.

To understand this better, let's look at the available authentication flow definitions for the `myrealm` realm. For that, open the administration console and click on the **Authentication** link in the left-hand side menu:

Authentication

Authentication is the area where you can configure and manage different credential types. [Learn more](#)



The screenshot shows the 'Flows' tab selected in the navigation bar. A search bar and a 'Create flow' button are at the top. Below is a table listing eight authentication flows:

Flow name	Used by	Description
browser <small>Built-in</small>	<input checked="" type="checkbox"/> Browser flow	browser based authentication
clients <small>Built-in</small>	<input checked="" type="checkbox"/> Client authentication flow	Base authentication for clients
direct grant <small>Built-in</small>	<input checked="" type="checkbox"/> Direct grant flow	OpenID Connect Resource Owner Grant
docker auth <small>Built-in</small>	<input checked="" type="checkbox"/> Docker authentication flow	Used by Docker clients to authenticate against the IDP
registration <small>Built-in</small>	<input checked="" type="checkbox"/> Registration flow	registration flow
reset credentials <small>Built-in</small>	<input checked="" type="checkbox"/> Reset credentials flow	Reset credentials for a user if they forgot their password or something
first broker login <small>Built-in</small>	Not in use	Actions taken after first broker login with identity provider account, which is not yet linked to any Keycloak account
http challenge <small>Built-in</small>	Not in use	An authentication flow based on challenge-response HTTP Authentication Schemes

Figure 11.1: Authentication flow definitions

On this page, you have a list of all the available flow definitions and how they are bound with the different flows supported by Keycloak. The flow to which a definition is bound is indicated by the **Used by** column on that list.

In the context of authentication, you can configure the following flows:

- Browser flow
- Direct grant flow
- Client authentication

Browser flow is related to how end users authenticate using a browser. Every time the end user authenticates in Keycloak, the steps from the definition associated with this flow are executed. As shown in the preceding screenshot, all the steps from the **browser** authentication flow definition are going to be executed when you're authenticating users through the browser.

The same goes for **Direct grant flow** and **Client authentication flow**. However, these two flows are related to how clients authenticate in a realm – **Client authentication flow** – or when clients are authenticating users – **Direct grant flow** – using backchannel requests to obtain tokens using the token endpoint.

Although this chapter is focused on the flows involved during user and client authentication, the same concepts should apply when configuring the self-registration (**Registration flow**) or resetting password (**Reset credentials flow**) flow.

Configuring an authentication flow

Keycloak allows you to configure authentication flows by changing their corresponding definitions directly or creating your own definitions using an existing one as a template.

The easiest – and recommended – way to create a flow definition is to use an existing one as a template by duplicating it. The reason for this is that you can easily roll back your changes and switch to

the definition you used as a template, in case the flow is broken by your changes.

Let's have a quick look at how to customize **browser** flow and see how it affects end users authenticating in a realm using a browser. For that, open the **browser** flow and then make a new flow by selecting **Duplicate** from the actions menu in the top right corner. You should be prompted to choose a name for the new flow. Let's name it **My Browser** and click on the **OK** button to create a new flow.

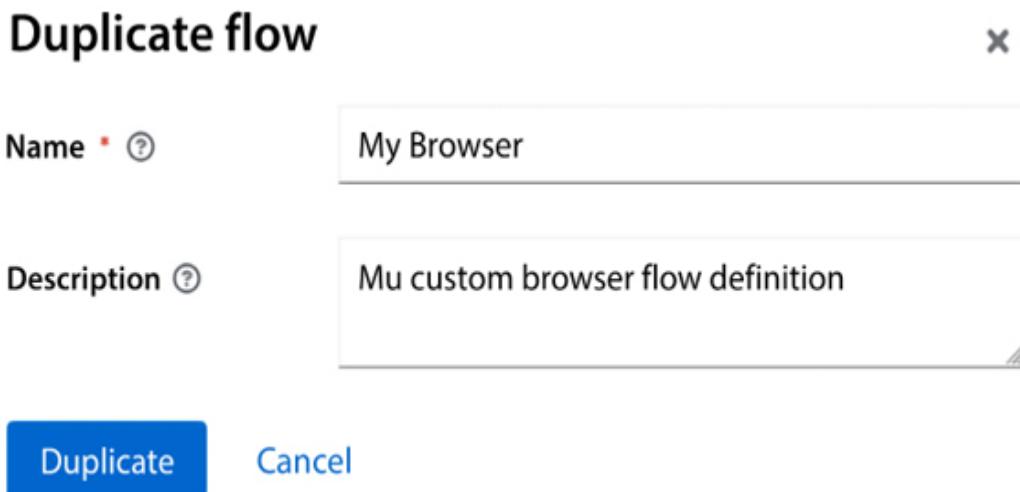


Figure 11.2: Creating a new flow definition from the built-in browser flow

An authentication flow definition is a hierarchical tree containing different authentication executions – the authentication steps – as well as other authentication flow definitions – also known as **subflows**.

Authentication executions are the actual steps that perform some action when you're authenticating an actor. These actions can be related to obtaining some input from the actor – such as asking end users for only their username when using the browser – or to

authenticate the actor using a specific authentication mechanism and credential, such as a password.

The elements in an authentication flow definition are executed sequentially, from top to bottom. The decision regarding whether the next step in the flow should be executed depends on the outcome of the current step and its settings. When a step is marked as **REQUIRED**, it must complete successfully prior to moving to the next one. If a required step completes successfully, the flow stops if there are no other required steps in the flow. On the other hand, when a step is marked as **ALTERNATIVE**, the flow can continue, even though the step was not completed successfully, so that other steps have a chance to successfully perform their actions.

For subflows, the **REQUIRED** and **ALTERNATIVE** settings are related to whether all the required executions or any of the executions within the flow completed successfully, respectively.

Taking the **My Browser** flow definition as an example, authentication is defined as follows:

1. First, the **Cookie** execution tries to seamlessly reauthenticate the user if there is a cookie in the request that maps to a valid user session. In other words, do not try to authenticate users if they previously authenticated to the realm. This step is marked as **Alternative**, indicating that if the user could not be authenticated at this step, the flow will continue.
2. If **Kerberos** execution is enabled, try to authenticate the user using any Kerberos credentials. Note that by default, this execution is disabled.

3. **Identity Provider Redirector** checks whether the realm has been configured to automatically redirect the user to a predefined identity provider. It is also marked as **Alternative**; not completing this step will continue the flow.
4. **My Browser Forms** is a subflow that groups specific steps to authenticate the user using password-based authentication and possibly 2FA using a **OTP**. Note that this step is marked as **Alternative**, so if any of the previous steps complete successfully, it will not be executed. Otherwise, users will be forced to provide their credentials again, even though they have already been authenticated.
5. The first step in the subflow is to authenticate the user using a username and password in a single step using the **Username Password Form** execution. This is the login page you saw when you authenticated to a realm. Note that this step must complete successfully as it is marked as **Required**.
6. If the previous step was successful – the user was authenticated initially – then there is another subflow called **My Browser Browser – Conditional OTP** to check whether 2FA using an OTP should be performed. In this subflow, the execution **Condition – User Configured** checks whether the user has an OTP credential set, and if so, executes the **OTP Form** step to authenticate the user using their **OTP**.

Now, let's change how users authenticate in the realm by using an *Identifier First Login* flow. In this case, the username and password are gathered in different steps, instead of asking for both using a single login page. For that, click on the trash icon on the right-hand

side of the **Username Password Form** to delete the execution. At the moment, your flow should look as follows:

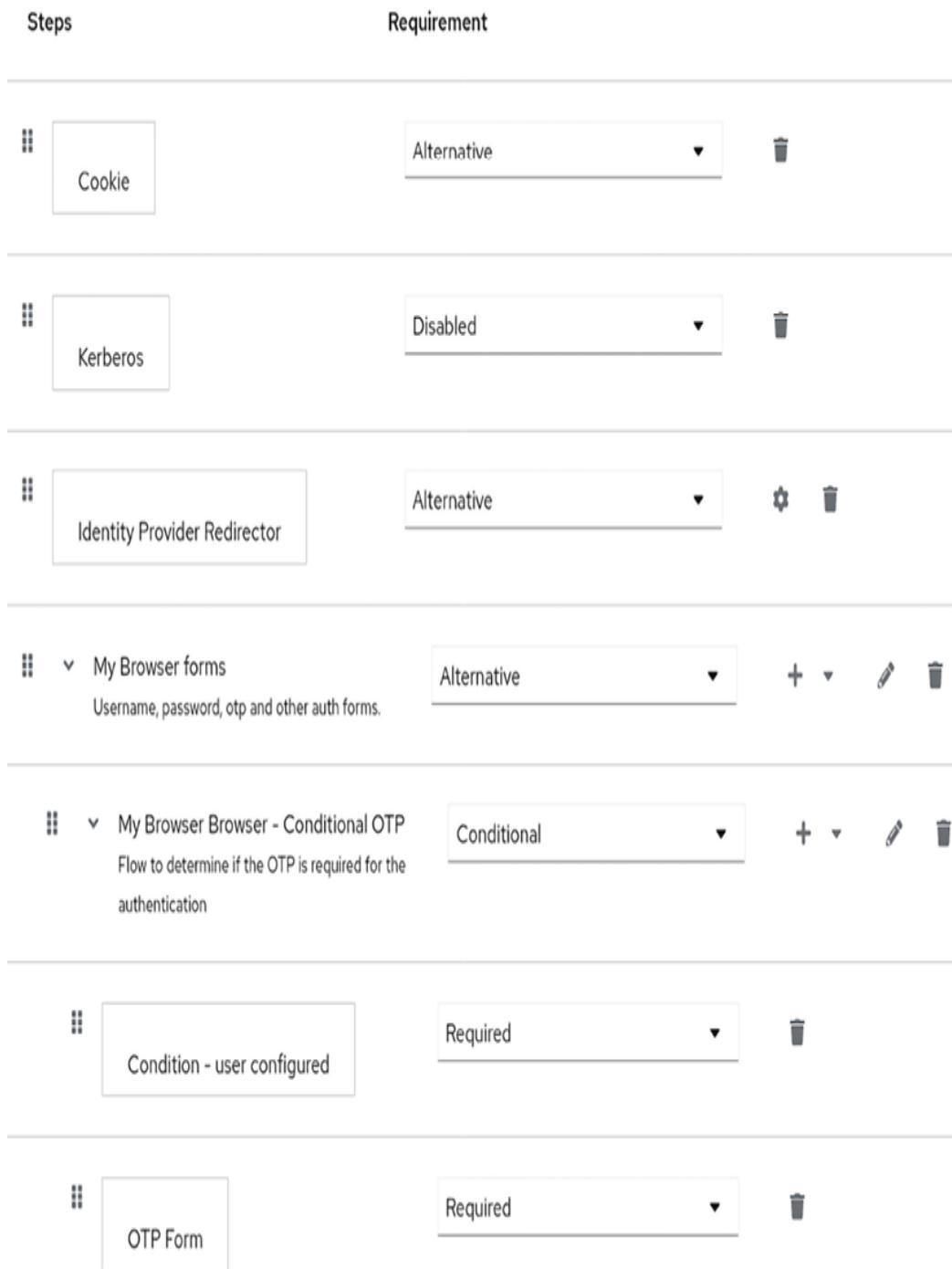


Figure 11.3: Removing the Username Password Form execution from the flow

Now, let's add two steps to this flow to ask the user for their username and then ask for their password. For that, click on the plus icon on the right-hand side of the **My Browser Forms** subflow and click on **Add step**.

Once you've done that, you should be redirected to a page where you can choose the authentication execution to include as a step of the subflow:

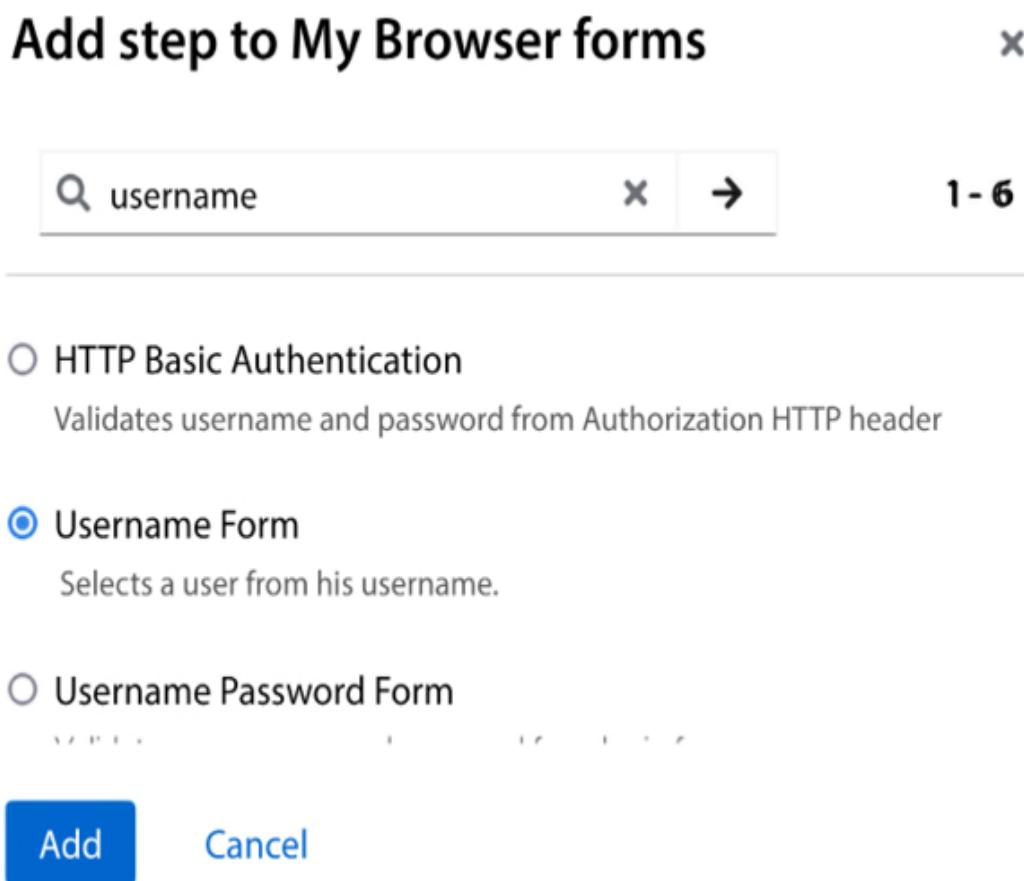


Figure 11.4: Choosing an authentication execution

On this page, you should be able to select from a vast list of authentication executions. In our case, we are going to select

Username Form from the list and click on the **Add** button to add the execution to the **My Browser forms** subflow.

Once the execution has been added to the flow, you should see it within the subflow. By default, executions are added to the bottom of the flow, but in our case, we want this execution at the top of the subflow so that we can obtain the username first. For that, click on the icon on the left-hand side of **Username Form** and drag the step up until it becomes the first execution in the subflow.

Perform the same steps you did previously to add the **Password Form** authentication execution to the subflow to obtain the password and authenticate the user. Make sure **Password Form** is the second execution in the subflow.

Let's make sure that both the **Username Form** and **Password Form** executions are marked as **Required**. For that, click on the **Required** setting for each authentication execution. This is an important step as it forces our end users to provide both pieces of information when they're logging in to the realm.

Now, the **My Browser** authentication flow should look like this:

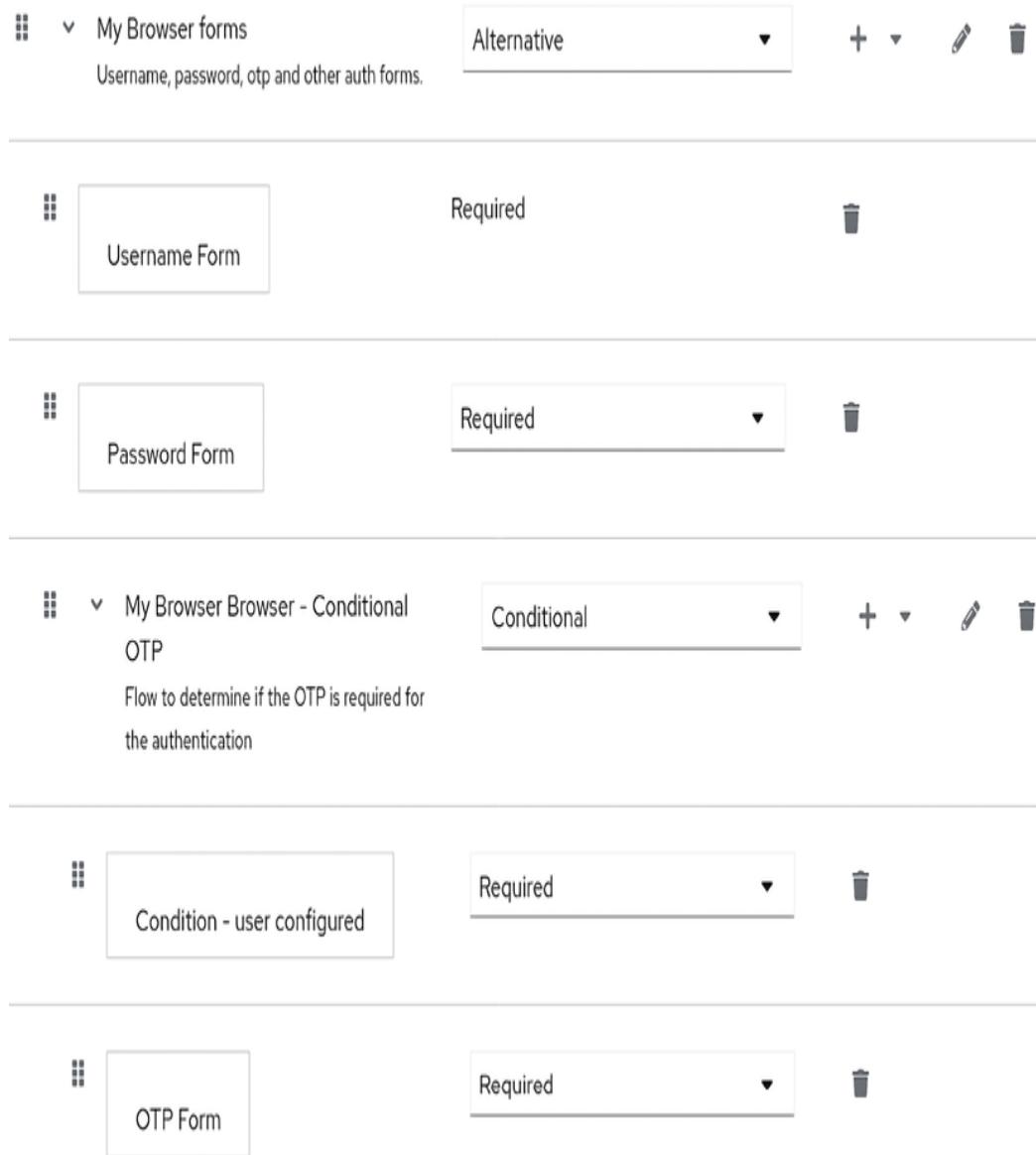


Figure 11.5: The final configuration for the My Browser authentication flow

Finally, you need to make sure the **My Browser** flow definition is assigned to the **Browser** flow.

For that, click on the **Action** button on the top right of the page and click on **Bind flow**. A modal dialog will be shown to choose the flow you want to bind the **My Browser** flow definition to. Make sure to select **Browser flow** and click on the **Save** button.



Figure 11.6: Binding the My Browser authentication flow definition to Browser flow

Now, let's try to log in to the Keycloak account console as `alice`.

For that, open your browser at

`http://localhost:8080/realm/myrealm/account` and log in using your user credentials. When authenticating to the realm, you should notice that the username and password of the user are obtained and validated in multiple steps.

In this section, you learned about the main aspects of authentication flows. You learned that Keycloak allows you to customize how users and clients authenticate by creating or changing authentication flow definitions. You also learned that by leveraging authentication flows, you can adapt Keycloak so that it fits into your authentication requirements.

In the next few sections, we will look at the different authentication methods supported by Keycloak.

Using passwords

In the previous chapters, you were basically using passwords to authenticate users. You were also quickly introduced to how to set up passwords when managing users. In this section, we are going

to look closer at how password-based authentication works and how passwords are managed.

We are not going to cover how users authenticate using passwords here because you are already familiar with that, but we will cover additional details around this form of authentication.

Password-based authentication is probably one of the most popular methods for authenticating users. It is easy to implement and is what most end users are used to when they need to authenticate to a system. However, the simplicity of this credential type has some disadvantages and weaknesses, all of which we will cover later in this section.

To help us overcome some of the disadvantages of password-based authentication, Keycloak relies on common best practices to make sure passwords are secure in transit and at rest. It also allows you to define policies so that you can govern some key aspects of password management, such as expiry, password format, and policies around reusing previous passwords.

Passwords are the simplest type of credential that you can set for your users and are used by Keycloak to authenticate users by default. Passwords are managed on a per-user basis, as you learned in the previous chapter. On the **User details** page, there is a **Credentials** tab, which provides everything you need to reset and delete the user's password:

The screenshot shows the Keycloak user management interface for a user named 'alice'. At the top, there is a status indicator 'Enabled' and an 'Action' dropdown. Below the header, there is a navigation bar with tabs: 'Credentials' (which is highlighted in blue), 'Role mapping', 'Groups', 'Consents', 'Identity provider links', and 'Sessions'. The main content area is titled 'Credentials' and contains a table with columns: 'Type', 'User label', and 'Data'. There is one row in the table where 'Type' is 'Password' and 'User label' is 'My password'. To the right of this row, there is a 'Show data' link and a prominent blue-bordered 'Reset password' button. A vertical ellipsis icon is also present on the far right.

Figure 11.7: Managing user passwords

Keycloak uses a strong password **hashing algorithm** to prevent brute-force attacks, as well as to securely store passwords. The default hashing algorithm used by Keycloak is **PBKDF2**, a well-known and widely used algorithm to keep passwords secure at rest.

Whenever you set a password for a user, its value is going to be combined with a secure random number, also known as a **salt**, and later hashed multiple times – the number of **iterations** – to create a derived key that is hard to crack. When stored, the password is never in plaintext. Instead, the derived key is stored together with the necessary metadata to validate the password afterward.

Keycloak is preconfigured for hashing passwords using **HMAC-SHA-256** and has an iteration count of **27,500**. You should be able to use a stronger hashing algorithm such as **HMAC-SHA-512** or change the number of iterations when you're configuring password policies, as we are going to see later.

The PBKDF2 algorithm is costly in terms of CPU. Depending on the CPU you have available for

Keycloak, it might impact its performance and overall response time. Make sure to change the number of iterations depending on your requirements to have a balance between performance and security. At the time this book is being written, OWASP recommends at least 600,000 iterations depending on the hashing algorithm in use.

Password-based authentication is not the most secure method of authenticating users since there is a long list of weaknesses associated with it. To name a few, passwords are usually stolen or leaked, they are susceptible to phishing attacks, and some users just do not care about how strong their passwords are, making your system as secure as how your users define, keep, and use their passwords. The fact that users usually use the same password across different systems also makes your system as secure as the weakest system in this chain. In terms of usability, when using policies to force users to use strong passwords, they become larger and more complicated, which makes them hard to remember or even type when the user is authenticating to a system, hence impacting the overall end user experience.

Keycloak helps you improve the overall security of password-based authentication, but it does not solve all such problems. Passwords, when used alone, are just a single factor for authenticating users, so you should consider using additional factors to improve the overall security of your system. As we are going to see in the upcoming sections, password-based authentication is not the only option you have for authenticating users in Keycloak, allowing you to employ

strong authentication to your system by combining other forms of authentication or even removing passwords completely.

In this section, you learned about the key aspects of how passwords are managed in Keycloak. You also learned that Keycloak relies on common best practices to keep passwords secure and that you can also use policies to control different aspects of password management.

In the next section, you will learn how Keycloak allows you to configure password policies to employ stronger passwords.

Changing password policies

Keycloak allows you to define different types of policies for passwords. These policies can be created by clicking on the **Authentication** link on the left-hand side menu, and then clicking on the **Policies** tab:

Authentication

Authentication is the area where you can configure and manage different credential types. [Learn more](#)

The screenshot shows a user interface for managing password policies. At the top, there are three tabs: 'Flows', 'Required actions', and 'Policies'. The 'Policies' tab is selected and highlighted with a blue underline. Below the tabs, there is a horizontal navigation bar with five items: 'Password policy' (underlined), 'OTP Policy', 'Webauthn Policy', 'Webauthn Passwordless Policy', and 'CIBA Policy'. In the center of the page is a large, dark gray circle containing a white plus sign (+). Below this icon, the text 'No password policies' is displayed in a bold, black font. Underneath this text, a message reads: 'You haven't added any password policies to this realm. Add a policy to get started.' At the bottom of the main content area, there is a button labeled 'Add policy' with a dropdown arrow icon to its right.

Figure 11.8: Password policy settings

In this tab, you can choose from different policies and manage specific aspects of password management, such as the following:

- Enforce the number of special characters, digits, and lowercase or uppercase characters in passwords.
- Define a minimum length.
- Define an expiration time.
- Avoid having the user's username in passwords.
- Define a blacklist dictionary.
- Avoid reusing previous passwords.

You can easily create any of these policies by clicking on the **Add policy** select box and then selecting the policy you want to create.

For a detailed description of each policy available in Keycloak, look at the documentation at https://www.keycloak.org/docs/latest/server-admin/#_password-policies.

By leveraging password policies, you should be able to overcome some of the weaknesses of password-based authentication by enforcing stronger passwords and controlling the frequency at which they should be updated. You should be able to easily define rules for passwords, such as avoiding having their username in passwords, forcing a specific number of special, lowercase, and uppercase characters, forcing a minimum length, and so forth.

In this section, you learned that Keycloak allows you to control different aspects of password management by using different types of policies.

In the next section, we will look at the different options we can use to reset user passwords.

Resetting user passwords

Keycloak allows you to reset users' passwords using different strategies. As an administrator, you can use the administration console to choose a password for a user or force the user to update their password when they log in. Users should also be able to reset their passwords when they're on the login page or when they're managing their account through the Keycloak account console.

When you change the password of a user through the administration console, Keycloak defaults to marking the new password as temporary. A temporary password means that the next time the user tries to log in to a realm, they must provide a new password.

You can control whether a password that's been set by an administrator is temporary by turning on the **Temporary** switch when you're creating or resetting the user password. If you turn off this setting, the user will not be asked to change their password when they log in.

A temporary password is nothing but a shortcut for setting a required action for a user to force them to update their password. The **Update Password** required action can be set at any time by an administrator to force a specific user to update their password:

alice

Enabled Action ▾

Details Attributes Credentials Role mapping Groups Consents Identity pro ▾

ID * e550c67f-66d7-410f-990b-8a5ff14a91eb

Created at * 4/3/2023, 8:29:32 AM

Required user actions Update Password X Select action X ▾

Username * alice

Email

Email verified ⓘ No

First name

Last name

Save Revert

The screenshot shows a user profile for 'alice' in the Keycloak administration interface. The top navigation bar includes tabs for Details, Attributes, Credentials, Role mapping, Groups, Consents, and Identity pro (partially visible). The 'Enabled' status is set to 'Enabled'. Below the tabs, there are several input fields: 'ID' (e550c67f-66d7-410f-990b-8a5ff14a91eb), 'Created at' (4/3/2023, 8:29:32 AM), and a 'Required user actions' section containing an 'Update Password' button. Other fields like 'Username' (alice), 'Email', 'First name', and 'Last name' are present but empty. At the bottom, there are 'Save' and 'Revert' buttons.

Figure 11.9: Using the Update Password required action to force users to update their password

From a user's perspective, passwords can be updated either by going to the Keycloak account console or by starting a specific flow

on the login page.

From the account console, users can change their passwords by clicking on the **Update** button when they're on the **Signing In** page:

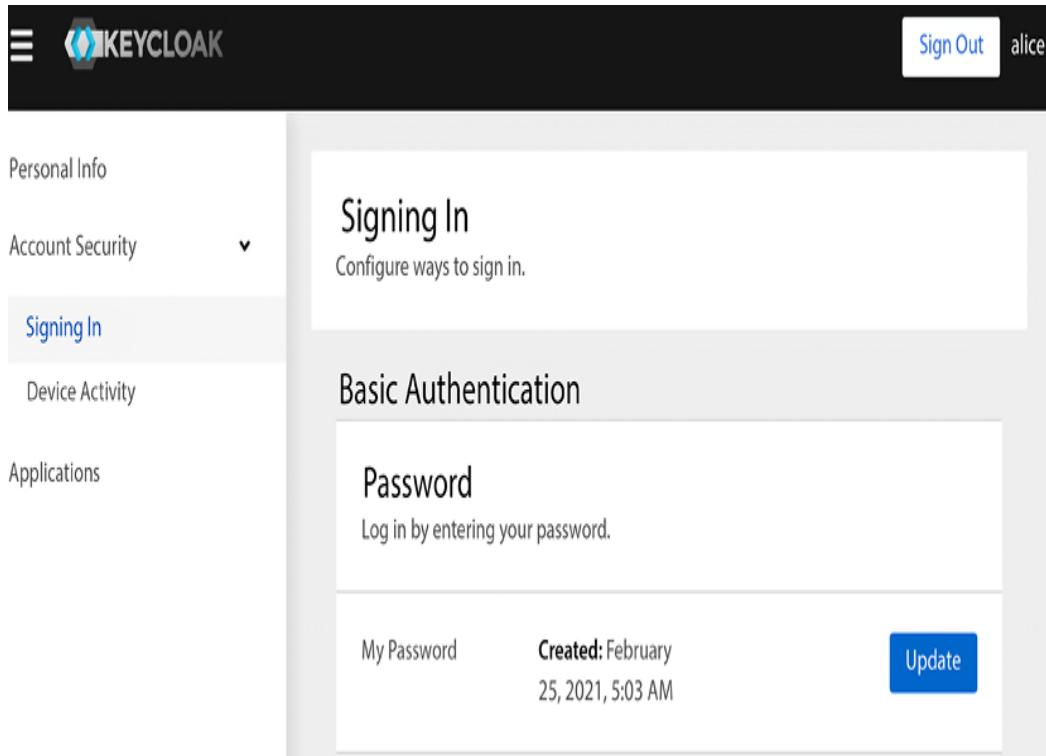


Figure 11.10: Updating the password using the account console

If a user has forgotten or lost their password, they can start a specific flow to reset their password on the login page. This flow is disabled by default. To enable it, you click on the **Realm Settings** link on the left-hand side menu of the administration console and click on the **Login** tab. On this tab, turn on the **Forgot password** setting. A link will appear on the login page that users can click on to reset their passwords:

myrealm

Enabled Action ▾

Realm settings are settings that control the options for users, applications, roles, and groups in the current realm. [Learn more](#)

< General Login Email Themes Keys Events Localization Security defen >

The screenshot shows the 'myrealm' realm settings interface. At the top, there's a status bar with 'Enabled' and an 'Action' dropdown. Below it, a sub-header says 'Realm settings are settings that control the options for users, applications, roles, and groups in the current realm.' with a 'Learn more' link. A navigation bar below has tabs: General, Login (which is selected and highlighted in blue), Email, Themes, Keys, Events, Localization, and Security defen. Under the 'Login' tab, there are three settings: 'User registration' (Off), 'Forgot password' (On), and 'Remember me' (Off). Each setting has a toggle switch icon.

Login screen customization

User registration ⓘ

Forgot password ⓘ

Remember me ⓘ

Figure 11.11: Changing the realm settings to allow users to reset their passwords

When this setting is enabled, users should be presented with a **Forgot Password?** link on the login page:

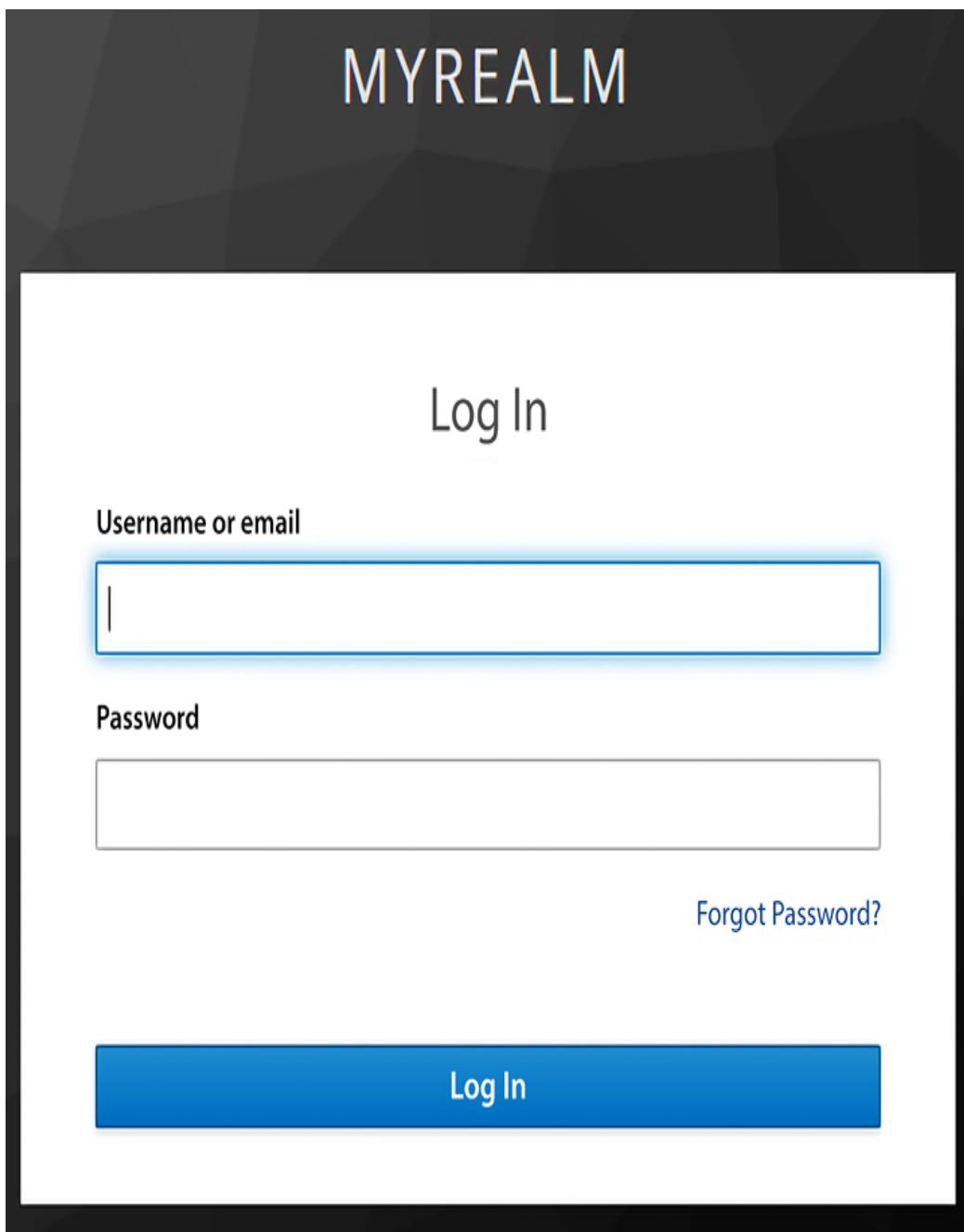


Figure 11.12: The **Forgot Password?** link on the login page

After clicking on the **Forgot Password?** link, users will be asked to provide their username or email so that they can receive a link by email to reset their password.

Note that this flow is based on email verification, where users should have a valid email address associated with their account. Your realm should also be configured to send emails using your preferred SMTP server. For more details on how to set up an SMTP server, look at the documentation at https://www.keycloak.org/docs/latest/server-admin/#_email.

In this section, we covered the different ways we can manage passwords. You learned that passwords can be set by an administrator and users can be forced to update their password when they're authenticating to a realm. You also learned that users can change or reset passwords through the account console or the login page, respectively.

You were also presented with more details on how password-based authentication works in Keycloak. You learned how Keycloak keeps passwords secure at rest, as well as how it helps you employ strong passwords and control the different aspects of password management by leveraging password policies. Finally, you learned that password-based authentication is only one, and not the most secure, of the available options you can use to authenticate users.

In the next section, we are going to look at how to authenticate users more securely by combining passwords and **OTPs** to enable **2FA**.

Using OTPs

As an additional layer of security, Keycloak allows you to use a second factor – or evidence – when authenticating users. In addition to providing a password – something users know – users are obligated to provide secondary evidence about their identity – something they have – which can be a code or a security key in their possession.

An OTP is probably one of the most common ways to enable 2FA for user accounts. It is relatively easy to use and adds an additional layer of security when you're authenticating users.

Although it's a useful method for 2FA, OTPs have some disadvantages. They rely on a shared key between the server and users and do not provide the best usability for end users, while still being open to common attacks such as phishing or scams. As we are going to see later, Keycloak helps you overcome these limitations by using a security device as a second factor using WebAuthn.

Keycloak makes it easy to configure and authenticate users using OTPs, where realms are automatically configured to support 2FA using OTPs. Users are also allowed to easily set up 2FA for their accounts by registering their devices to generate OTP codes.

In the next few sections, we will look at how to configure and authenticate users using OTPs.

Changing OTP policies

Keycloak allows you to define different policies for OTPs. These policies can be changed by clicking on the **Authentication** link on the left-hand side menu and then clicking on the **OTP Policy** tab:

Authentication

Authentication is the area where you can configure and manage different credential types. [Learn more](#)

The screenshot shows the 'OTP Policy' configuration page within the Keycloak 'Authentication' section. The top navigation bar includes tabs for 'Flows', 'Required actions', and 'Policies', with 'Policies' being the active tab. Below the tabs, there are five policy categories: 'Password policy', 'OTP Policy' (which is selected and highlighted in blue), 'Webauthn Policy', 'Webauthn Passwordless Policy', and 'CIBA Policy'. The main configuration area starts with 'OTP type' set to 'Time based' (radio button selected). Below it is 'OTP hash algorithm' set to 'SHA1'. Under 'Number of digits', '6' is selected (radio button). There is a 'Look ahead window' input field with a value of '1'. The 'OTP Token period' is set to '30 Seconds'. Under 'Supported applications', three options are listed: 'FreeOTP', 'Google Authenticator', and 'Microsoft Authenticat...'. At the bottom, there is a 'Reusable token' switch set to 'Off'. Finally, there are 'Save' and 'Reload' buttons.

Flows Required actions Policies

OTP Policy

OTP type ② Time based Counter based

OTP hash algorithm ② SHA1

Number of digits ② 6 8

Look ahead window ② - 1 +

OTP Token period ② 30 Seconds

Supported applications ② FreeOTP Google Authenticator Microsoft Authenticat...

Reusable token ② Off

Save Reload

Figure 11.13: OTP Policy tab

An OTP is a code based on a secret key – hashed using a specific algorithm – and a moving factor that can be either the current time or a counter, where this code can only be used once to authenticate a user. Keycloak can authenticate and allow users to generate codes using two main algorithms:

- **Time-Based One-Time Password (TOTP)**
- **HMAC-Based One-Time Password (HOTP)**

By default, realms created in Keycloak are configured to use TOTPs. They are also configured with only six digits and have a validity window of 30 seconds. You can change these settings at any time, based on your requirements.

When you change the details in this tab – mainly the OTP type and the hash algorithm – make sure the apps that are being used by your users to generate codes support the configuration.

The difference between the two algorithms is the moving factor that's used to generate the code and how they are validated. As the name implies, a TOTP is based on time, so the code is only valid for a certain period – usually 30 seconds. On the other hand, an HOTP is based on a counter. The validity of the code is infinite until the code is validated and the counter is increased.

The decision of which algorithm to use is use case-specific. However, TOTPs provide better security than HOTPs because if the code is lost or leaked, its validity window is reduced, hence reducing the attack surface when it comes to using OTPs.

Due to TOTPs being based on time, you should be aware that the host where Keycloak is deployed should have the clock in sync with the devices being used by your users to generate codes. If the clock is not in sync, users might be unable to authenticate using a TOTP. You should be able to define a clock skew compensation to reduce the difference between clocks. For that, set **Look Ahead Window** with the number of seconds to compensate for the time difference.

From a user's perspective, Keycloak allows them to obtain OTP codes from their own personal devices – such as their smartphones or tablets – using two main mobile applications available from the Android and iOS app stores:

- **FreeOTP**
- **Google Authenticator**

As you will learn in the next few sections, by using either of these apps, your users can easily enable 2FA for their accounts and authenticate in a realm using OTPs.

In this section, you learned about OTPs and how to change their settings. You also learned that users can use their preferred devices to generate codes using the FreeOTP and Google Authenticator apps.

Now, let's look at the different strategies we can use to authenticate users using OTPs.

Allowing users to choose whether they want to use OTPs

Once your users have been successfully authenticated using their passwords, Keycloak is going to check whether they have any OTP credentials associated with their account. If no OTP credentials have been set, Keycloak authenticates the user and redirects the user back to the application. That is the behavior you have seen so far when a user has been authenticating to a realm.

However, if the user has an OTP credential set, Keycloak is going to perform an additional step during the authentication flow to obtain the OTP from the user and validate it, prior to authenticating the user.

Let's see how this works by logging in to the account console using the user **alice**. For that, open your browser to

`http://localhost:8080/realm/myrealm/account` and log in using the appropriate user credentials. Note that at the moment, **alice** is only using her password to authenticate.

In the account console, users can set up 2FA using an OTP by clicking on the **Set up authenticator application** button when they're on the **Signing in** page:

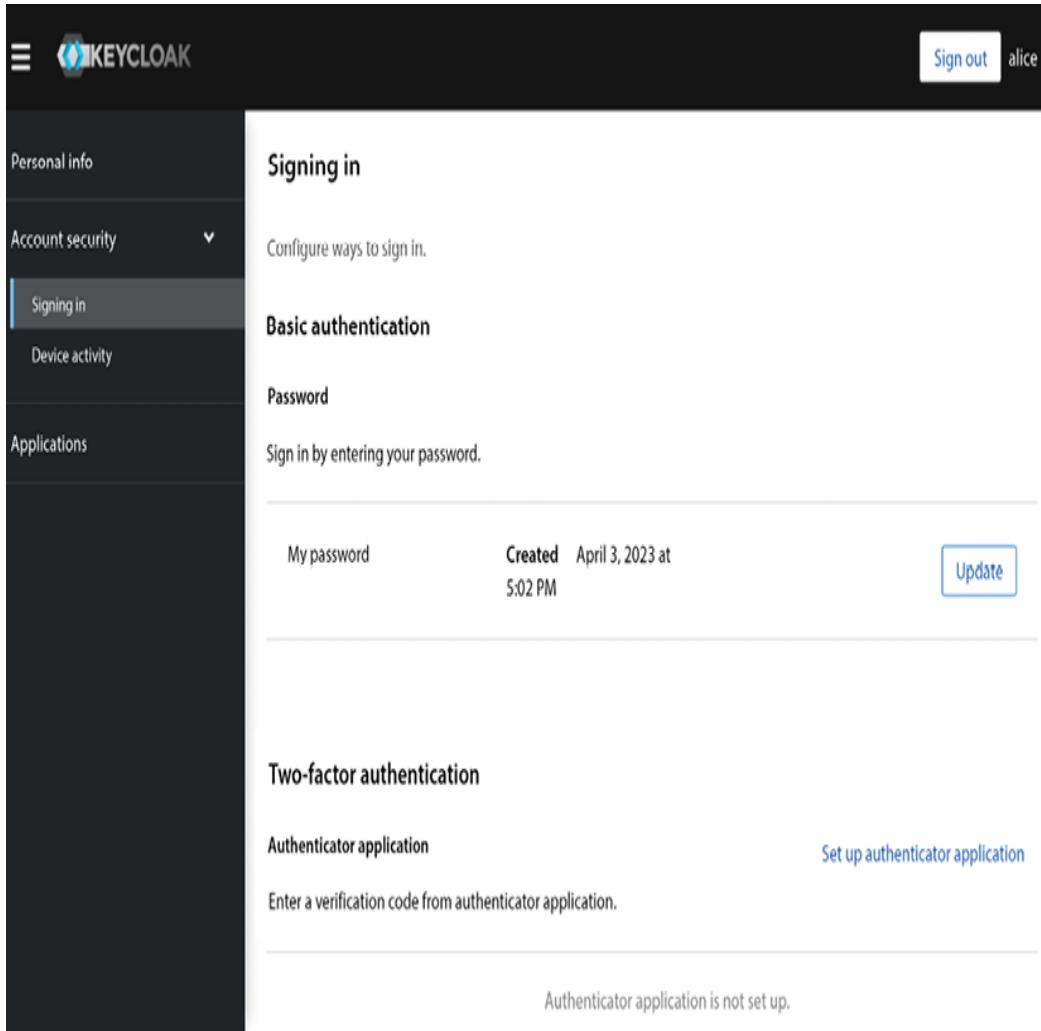


Figure 11.14: Configuring 2FA using an OTP

After choosing to set up a new authenticator, your users will be presented with a QR code representing the shared key that will be used to generate the codes.

By using your smartphone, you should be able to scan this QR code using either the FreeOTP or Google Authenticator mobile application:

Mobile Authenticator Setup

1. Install one of the following applications on your mobile:

Microsoft Authenticator

Google Authenticator

FreeOTP

2. Open the application and scan the barcode:



[Unable to scan?](#)

3. Enter the one-time code provided by the application and click Submit to finish the setup.

Provide a Device Name to help you manage your OTP devices.

One-time code *

Device Name

Submit

Cancel

Figure 11.15: Configuring a new OTP

After scanning the QR code using any of these applications, they are going to start generating the codes that we will be using to complete the OTP credential registration process, as well as to authenticate the user later, once we've finished this step. Note that codes are generated every 30 seconds since we are using a TOTP.

To complete the OTP credential registration process, set the **One-time code** field with any code from the mobile application you are using and click on the **Submit** button. Optionally, users can also define an alias for the OTP credential to associate it with a specific device or application.

Now, let's try to authenticate again as `alice`. To do that, click on the **Sign Out** link in the top-left corner of the page to log out from the account console, and then authenticate again using the username and password of that user:

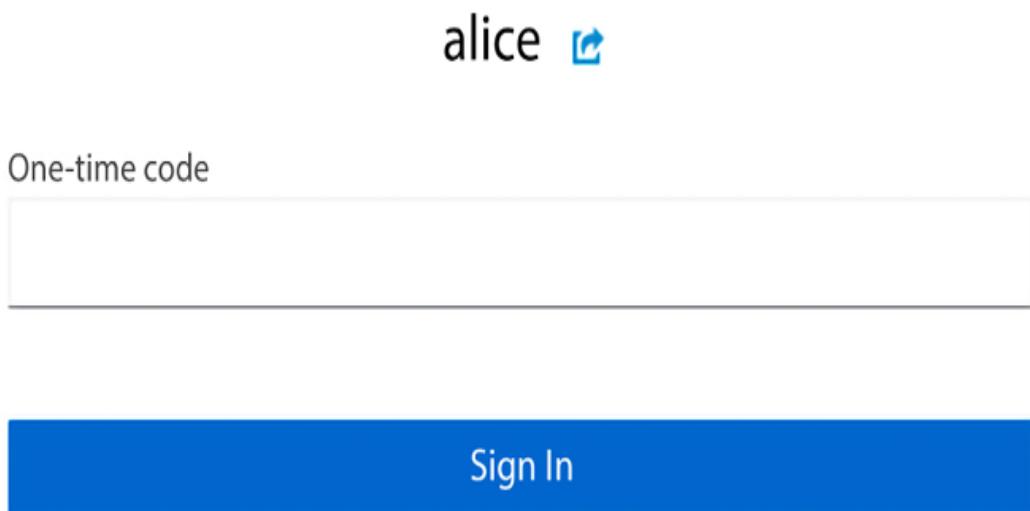


Figure 11.16: User is prompted to provide the code when logging in

Compared to what happened previously, now, you will be presented with a page asking you to provide a code. Use your smartphone to obtain a code and fill in the **One-time code** field. By clicking the **Sign In** button, you should be able to access the account console if Keycloak was able to successfully validate the code you provided.

In this section, you learned that Keycloak defaults to requiring 2FA, but only if the user is associated with an OTP credential. You also learned that by leveraging the account console, users can easily set up 2FA for their account.

In the next section, you are going to learn how to enforce 2FA for all users in a realm.

Forcing users to authenticate using OTPs

For some use cases, the decision of whether to authenticate using an OTP is not up to users but based on the security constraints that have been defined for a realm. Keycloak allows you to change the default behavior of OTP authentication to force users to either set up an OTP credential prior to authenticating or use an existing one to successfully authenticate to a realm.

To enable this behavior, click on the **Authentication** link on the left-hand side menu and then click on the **My Browser** authentication flow definition from the list.

The **My Browser** flow definition was created at the beginning of the chapter, and it should be bound to the **Browser flow**.

On this page, you are going to change the requirements for the **My Browser Browser - Conditional OTP** step and mark it as **Required**:

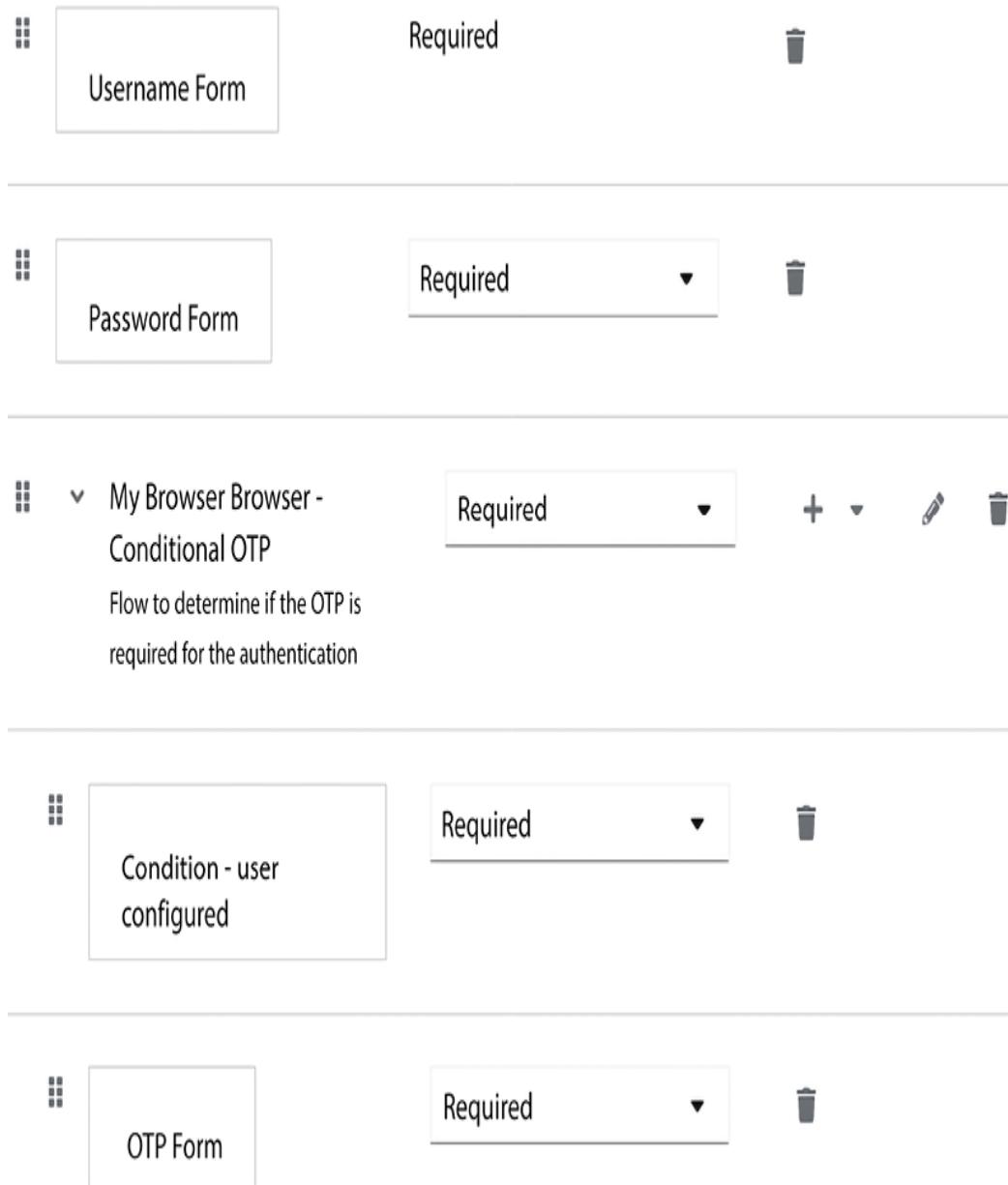


Figure 11.17: Enforcing 2FA for a realm

Now, go to the user settings for `alice` and remove the OTP credential associated with the account.

Note that similar to passwords, administrators can set a required action to force users to configure an OTP when they are logging in. For OTPs, the name of the required action is **Configure OTP**.

Now, let's log in to the account console using `alice`. For that, open your browser to

`http://localhost:8080/realm/myrealm/account` and log in using the user's credentials.

Compared to what happened previously, the user is now forced to set up an OTP credential. The steps to do so are the same as when using the account console, as you learned in the previous section. The main difference here is that users are obligated to set up an OTP credential if they do not have one. Only after that can they authenticate in a realm.

In this section, you learned how to use an OTP to enable 2FA to a realm. You learned that 2FA provides stronger authentication than only using passwords to authenticate users. You also learned that users can easily enable 2FA to their accounts by using the FreeOTP or Google Authenticator app.

We are going to extend the concepts we've presented in this chapter to set up a stronger authentication using WebAuthn for 2FA and MFA.

Using Web Authentication (WebAuthn)

The WebAuthn protocol aims to improve the security and usability of authenticating users over the internet. For that, it provides additional capabilities for server and security devices to communicate with each other – using the browser as an intermediary – to authenticate users using a cryptography protocol.

WebAuthn is based on asymmetric keys – a private-public key pair – to securely register users' devices and authenticate them in a system. There is no shared key between devices and the server, only a public key. By acting as an intermediary between security devices and the server, WebAuthn makes it possible to use these devices for 2FA or MFA using biometrics, or to seamlessly authenticate users without any explicit credentials other than their security devices: a concept also known as username-less and password-less authentication.

When used for 2FA, WebAuthn is a more secure method than OTPs because there is no shared key between Keycloak and the third-party applications used to generate codes. Instead, users are granted a security device that relies on strong cryptography to communicate the second factor without exposing any sensitive data.

For users, WebAuthn improves their experience when they're authenticating to a system by completely eliminating the need to deal with passwords or OTP codes. Instead, they can use their devices to seamlessly authenticate themselves.

A security device – or authenticator – can be anything as long as it complies with a set of requirements from FIDO2. It can be a smartphone with support for fingerprints, a security key that's attached through a USB, or a **Near-Field Communication (NFC)** device.

WebAuthn gives you fine-grained control over the different aspects of how users register and authenticate through these devices. It allows you to control the requirements of how to verify the identity of the user in possession of a device or whether credentials should be stored in the device, hence eliminating the need for storing credentials on the server.

In Keycloak, you can use WebAuthn to solve different use cases:

- Allow users to register devices either during authentication or using the account console.
- Use security devices for 2FA as a more secure alternative to OTP.
- Use security devices for MFA using any form of biometric authorization supported by these devices.
- Use security devices for username-less or password-less authentication.

You should also be able to allow your users to choose from multiple authentication methods when they're on the login page. For instance, you can allow users to choose whether they want to use password-less authentication using WebAuthn or password-based authentication with an OTP as a second factor.

For more details about how to use WebAuthn in Keycloak, look at the documentation available at https://www.keycloak.org/docs/latest/server-admin/#_webauthn.

In this section, you learned about some of the key concepts surrounding WebAuthn and how it helps to employ strong authentication. You also learned that WebAuthn improves the user experience when security devices are used as a second factor or by removing the need to type in any credentials.

In the next section, we are going to learn how to define an authentication flow to authenticate users using WebAuthn.

Enabling WebAuthn for an authentication flow

To allow users to authenticate using their devices, we are going to need to create an authentication flow definition that supports WebAuthn.

Based on what you learned in the *Understanding authentication flows* section, create a new flow by performing the following steps:

1. Create a new flow by using the **My Browser** flow – you created it in the first section – as a template. Name the new flow **My WebAuthn**.
2. Delete the **OTP Form** execution from the **My WebAuthn My Browser Browser – Conditional OTP** subflow.

3. Add a new step to the **My WebAuthn My Browser Browser – Conditional OTP** subflow by selecting the **WebAuthn Authenticator** execution.
4. Mark **My WebAuthn My Browser Browser – Conditional OTP** as a **Conditional** flow.

At this point, you should have a flow definition that looks as follows:

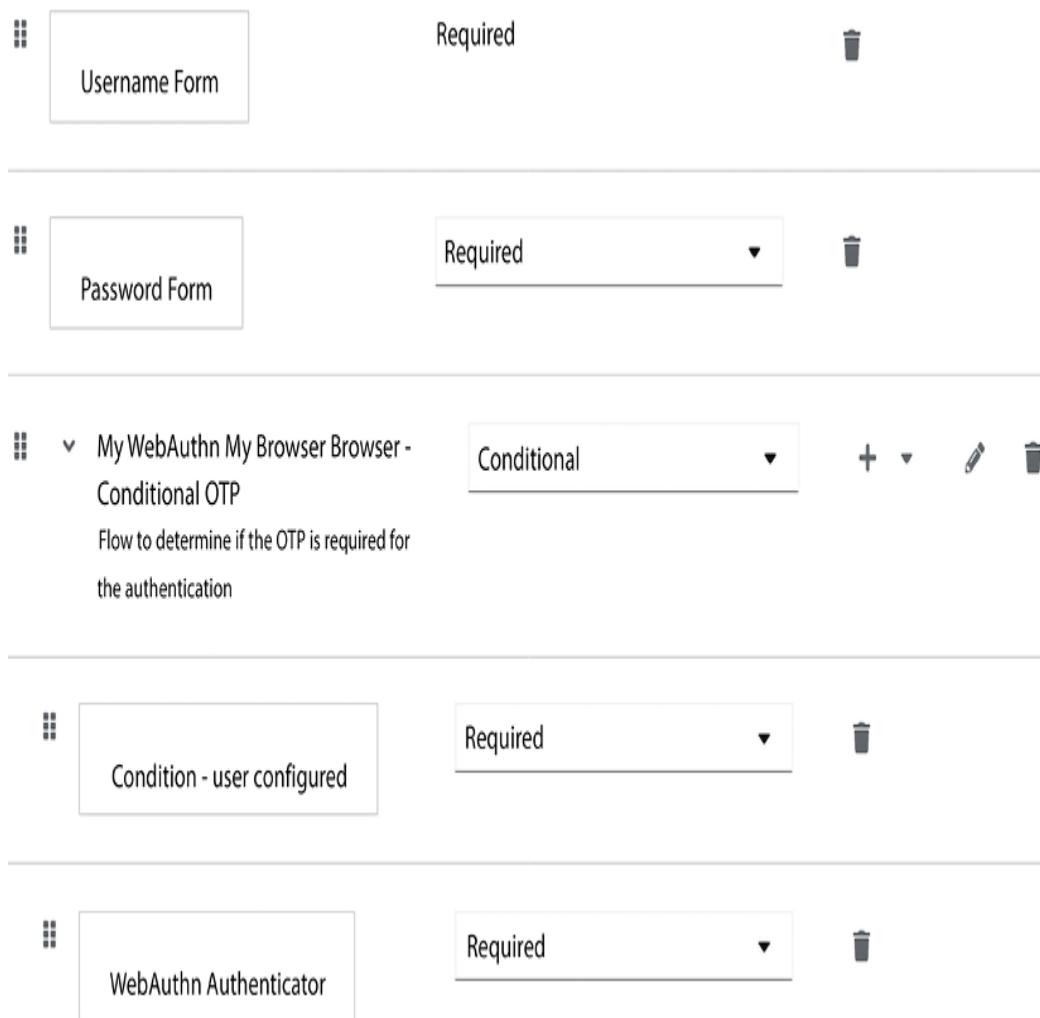


Figure 11.18: Creating an authentication flow definition to authenticate using a WebAuthn-compliant security device

Finally, associate this authentication flow definition with **Browser flow**.

As you may have noticed, we are basically replacing the OTP with WebAuthn for 2FA. You did not have to do much except replace the **OTP Form** execution with the **WebAuthn Authenticator** execution.

If you try to log in, you will still only be able to log in using a password because the user hasn't been configured with a security device yet.

In the next section, you will learn how users can register security devices using the account console.

Registering a security device and authenticating

Now that the realm is configured to support security devices as a second authentication factor using WebAuthn, let's register the security device through the account console. Open your browser at <http://localhost:8080/realm/myrealm/account> and log in using the required user credentials. In the account console, click on the **Signing in** link on the left-hand side menu:

The screenshot shows the Keycloak account settings interface for a user named 'alice'. The left sidebar includes 'Personal info', 'Account security' (selected), 'Device activity', and 'Applications'. The main content area displays 'My password' (Created April 3, 2023 at 5:02 PM) with an 'Update' button. Under 'Two-factor authentication', it says 'Authenticator application' and provides a placeholder 'Enter a verification code from authenticator application.' A blue link 'Set up authenticator application' is present. Below this, a message states 'Authenticator application is not set up.' Under 'Security key', it says 'Use your security key to sign in.' and features a blue link 'Set up Security key'. A message below indicates 'Security key is not set up.'

Figure 11.19: List of available devices

On this page, look at the **Security key** section and click on **Set up Security key** to register a security device:

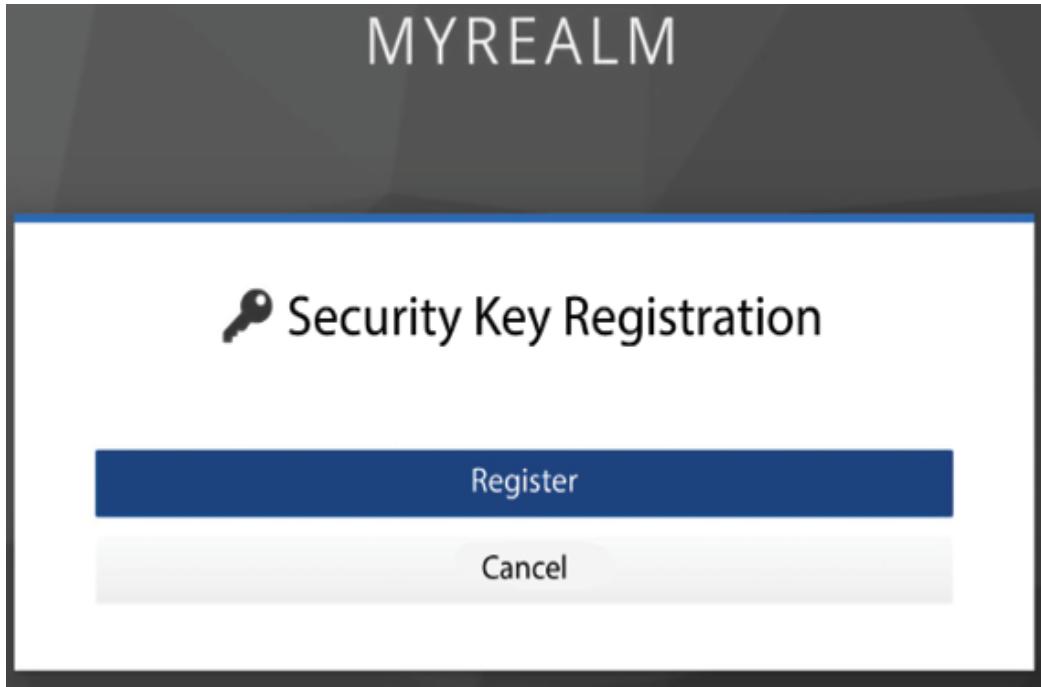


Figure 11.20: Registering the device

To register the new device, click on the **Register** button. The browser should ask you to use your security device to complete the registration – such as by touching it – and ask what you want to name the device. You can use any name you want.

After successfully registering the device, it should be listed among the available options for 2FA:

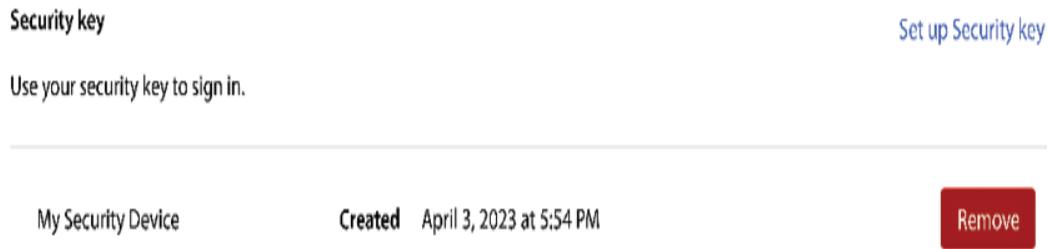


Figure 11.21: Security device successfully registered

Now, let's try to authenticate again as `alice`. For that, click on the **Sign Out** link in the top-left corner of the page to log out from the

account console, and authenticate again using the username and password of the user:

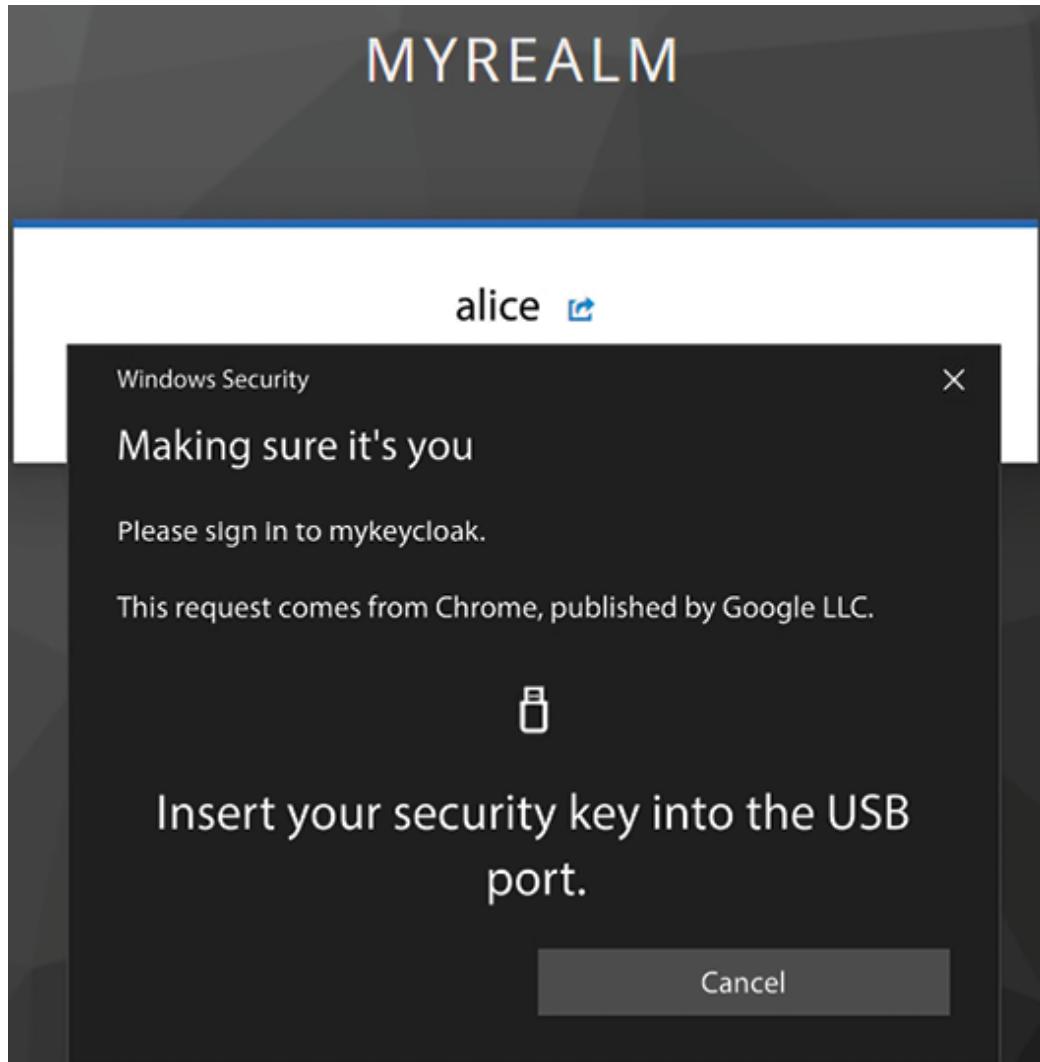


Figure 11.22: User is requested to interact with the security device to complete the authentication process

When authenticating as `alice`, you should be prompted to interact with the security device to complete the authentication. Note that this behavior is quite similar to using an OTP as a second factor, but the user does not need to type in any code, nor are any codes sent over the wire.

In this section, you learned how to register security devices using the account console. You also learned how easy it is to use WebAuthn and security devices for 2FA. You then learned that Keycloak allows you to use WebAuthn for MFA by relying on any biometrics authorization provided by a security device.

In the next section, you will be presented with some key concepts around strong authentication.

Using strong authentication

Strong authentication is a term that's widely used nowadays. What it means depends on the context where it is used. In general, strong authentication is about employing either 2FA or MFA to authenticate users.

As you learned in the previous sections, Keycloak provides the necessary capabilities to enable either 2FA or MFA for a realm. If your requirements for strong authentication only include the use of 2FA, you are good to go with either an OTP or a security device when you're using WebAuthn.

However, MFA is probably the strongest form of authentication you can get, where biometric authorization is a key aspect of securely identifying and authenticating the user. In this case, you should consider using WebAuthn and setting up security devices to verify the identity of the user – using fingerprint scanning, for instance – to make sure the user using the device is indeed the user trying to authenticate.

Strong authentication can also involve leveraging 2FA or MFA to enable other authentication factors, such as the history of IP addresses and devices that users are using to authenticate. In this case, you might want to select the best factor to authenticate a user, depending on a risk score or based on the user's behavior.

Alternatively, you may just wish to force the user to reauthenticate when they're accessing sensitive data or performing a critical action on your system.

At the time of writing this book, Keycloak does not provide built-in support for some common authentication paradigms such as adaptive authentication or risk-based authentication.

However, it is possible to enable step-up authentication to a realm so that different and stronger credentials are requested until a pre-determined level of trust is achieved during the user session lifetime. For more details about how to enable step-up authentication, look at the documentation at

https://www.keycloak.org/docs/latest/server_admin/#_step-up-flow.

As you learned in this chapter, Keycloak is very flexible in terms of how you can configure as well as implement new forms of authentication for a realm. As you will see in *Chapter 13, Extending Keycloak*, Keycloak provides a set of **Service Provider Interfaces (SPIs)** that developers can use to extend its core capabilities.

In this section, you had a quick overview of strong authentication and how Keycloak can help you to achieve it. Now, let's take a look at this chapter's summary.

Summary

In this chapter, you were provided with more details on how to authenticate users in Keycloak. First, you were introduced to authentication flows and how they play an important role in defining how users – as well as clients – authenticate to a realm. You were presented with the main authentication methods supported by Keycloak and how to configure them to promptly authenticate users, as well as how to combine them to support 2FA and MFA. Finally, you were briefly introduced to strong authentication and how Keycloak can help you employ secure authentication methods for a realm.

By leveraging the information in this chapter, you should now be able to customize Keycloak to authenticate users according to your needs and use different authentication methods.

In the next chapter, you are going to look at session management and how it correlates with authentication.

Questions

1. How do I change the look and feel of the pages shown in this chapter?
2. I cannot follow the WebAuthn examples and register a security device. What am I missing?

Further reading

- Keycloak authentication documentation:
https://www.keycloak.org/docs/latest/server_admin/#configuring-authentication_server_administration_guide
- Keycloak required actions:
https://www.keycloak.org/docs/latest/server_admin/#con-required-actions_server_administration_guide
- Keycloak login page settings:
https://www.keycloak.org/docs/latest/server_admin/#controlling-login-options
- Keycloak account console:
https://www.keycloak.org/docs/latest/server_admin/#account-service
- Keycloak step-up authentication documentation:
https://www.keycloak.org/docs/latest/server_admin/#step-up-flow
- WebAuthn: <https://webauthn.io/>

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/SecNet>



12

Managing Tokens and Sessions

In addition to acting as a centralized authentication and authorization service, Keycloak is, at its core, a session and token management system.

As part of the authentication process, Keycloak may create server-side sessions and correlate them with tokens. By relying on these sessions, Keycloak is able to keep the state of the authentication context where sessions originated, track users' and clients' activity, check the validity of tokens, and decide when users and clients should re-authenticate.

In this chapter, we are going to look at how Keycloak allows you to manage tokens and their underlying sessions, as well as understanding the different aspects that you should be aware of when doing so. For that, we are going to cover the following topics:

- Managing sessions
- Managing tokens

Technical requirements

During this chapter, you are going to use the Keycloak administration console to follow some of the examples provided

herein; therefore, make sure you have Keycloak up and running as per what you learned from *Chapter 1, Getting Started with Keycloak*.

Managing sessions

Session management has a direct impact on some key aspects such as user experience, security, high availability, and performance.

From a user experience perspective, Keycloak relies on sessions to determine whether users and clients are authenticated, for how long they should be authenticated, and when it is time to re-authenticate them. This characteristic of sessions is basically what gives users the **single sign-on (SSO)** experience when authenticating to different clients within the same realm, and what makes a unified authentication experience possible.

From a security perspective, sessions provide a security layer for tracking and controlling user activity and making sure that tokens issued to clients are still valid passports to act on behalf of users. They are also important for limiting and controlling the amount of time for which users can stay connected to a realm and its clients, helping to reduce the attack surface when sessions or tokens are leaked or stolen. As we are going to see later in this topic, sessions can be invalidated prematurely by administrators, users, and clients as a reaction to, or in the prevention of, unauthorized access from malicious actors.

From a performance perspective, sessions are kept in memory and distributed across the different nodes in a cluster, having a direct impact on the server runtime. As you learned from *Chapter 9*,

Configuring Keycloak for Production, Keycloak stores sessions in shared caches where the number of active sessions, how long they are kept alive for, and their node affinity are key factors that need to be balanced to optimize network, memory, and CPU resources.

With all that in mind, Keycloak provides you with flexible session and token management to balance all three aspects mentioned herein. Administrators should be able to track the active sessions for users and clients, check which clients users are authenticated to, force a single or global logout for invalidating sessions, revoke tokens, and control the different aspects of sessions' and tokens' lifetimes.

In the next topic, we are going to start looking at how to manage sessions' lifetimes.

Managing session lifetimes

One of the first questions you need to answer before going to production with Keycloak is how often users and clients should re-authenticate. To help you with this question, you should be aware of how Keycloak creates sessions and how to define their lifetimes.

The session lifetime determines when sessions should expire and be destroyed. Once expired, the users and clients associated with these sessions are no longer authenticated and are forced to re-authenticate to establish a new session.

Keycloak expires sessions using a background task that runs from time to time, checking for expired

sessions. By default, the task runs every 15 minutes. You are free to change this value if you really need to. The default setting should be enough for most deployments.

Keycloak creates sessions at different levels when authenticating users. Firstly, a user session is created to track the user activity regardless of the client. This first level is what is called the SSO session, also referred to as a user session. At the second level, Keycloak creates a client session to track the user activity for each client the user is authenticated to in the user session. Client sessions are strictly related to the validity of tokens and how they are used by applications.

As a top-level session, the SSO session lifetime is a global setting used to control how often users and clients need to re-authenticate. Keycloak allows you to configure the maximum amount of time for which SSO sessions should be kept alive and the idle period after which to expire sessions prematurely. When an SSO session expires, all client sessions associated with it also expire.

The SSO session is like an HTTP session. Both are used to track and maintain the state across multiple requests from the same agent.

To configure these settings, you should click on **Realm Settings** on the left-side panel and then click on the **Sessions** tab:

The screenshot shows the 'Sessions' configuration page for the realm 'myrealm'. At the top right, there is a blue circular icon with a white circle inside, labeled 'Enabled'. To its right is a link 'Learn more' with a blue arrow icon. Below the header is a navigation bar with tabs: General, Login, Email, Themes, Keys, Events, Localization, Security defenses, Sessions (which is highlighted in blue), and Tokens. The main content area is titled 'SSO Session Settings'. It contains four sets of input fields for session parameters:

- SSO Session Idle**: Set to 30 Minutes.
- SSO Session Max**: Set to 10 Hours.
- Remember Me**: Set to 0 Minutes.
- SSO Session Max**: Set to 0 Minutes.

Figure 12.1: Configuring SSO session lifetime

From this tab, you can set both maximum and idle times for SSO sessions by setting the **SSO Session Max** and **SSO Session Idle** settings, respectively. These two settings together effectively tell Keycloak that sessions should be kept alive for a certain amount of time and no longer than that, and that in the meantime, Keycloak should check user activity within a certain period – the idle period – to decide whether sessions should expire prematurely.

Let's understand these two settings by example. By default, Keycloak defines a 10-hour lifetime for SSO sessions. This time effectively means that sessions can live up to 10 hours and no longer than that.

However, the idle timeout is set to 30 minutes by default, and that effectively means that if Keycloak does not see any user activity within a 30-minute period, sessions are going to be destroyed,

regardless of the maximum time set. The idle timeout is bumped every time users interact with Keycloak, either directly through the authorization endpoint – when using a browser – or indirectly when tokens are refreshed by clients.

If a user authenticates and moves away from the keyboard and the client does not refresh its tokens during this period, the user session will be destroyed in 30 minutes. However, if the user is constantly interacting with Keycloak using the browser, or the client is constantly refreshing its tokens, the user session can last up to 10 hours.

Like SSO sessions, administrators can set the **Client Session Idle** and **Client Session Max** settings to set the idle and maximum time for client sessions, respectively:

Client session settings

The image shows two input fields for configuring client session lifetime. The top field is labeled "Client Session Idle" with a value of "0" and a unit of "Minutes". The bottom field is labeled "Client Session Max" with a value of "0" and a unit of "Minutes". Both fields have up and down arrow buttons for adjusting the value.

Client Session Idle	0	Minutes
Client Session Max	0	Minutes

Figure 12.2: Configuring the client session lifetime

These two settings provide administrators with more fine-grained control over the session lifetime of clients, making it possible to define hard limits for how long tokens are valid and force clients to re-authenticate whenever they try to refresh tokens. In other words, tokens issued to any client in a realm are only valid up to the

maximum time you set, with the possibility to prematurely expire client sessions and invalidate tokens if the client is not refreshing its tokens within the idle period.

However, and differently from SSO sessions, when a client session is invalidated, users are not necessarily forced to re-authenticate if their SSO sessions did not expire, but it will force clients to re-authenticate to obtain a new set of tokens. Note that when a client session expires, users might be redirected to Keycloak as a consequence of forcing the client to re-authenticate, potentially causing some impact on the user experience when users are using a browser.

By default, Keycloak defines the same configuration set for SSO sessions to control the client session lifetime. By changing the value of the **Client Session Idle** and **Client Session Max** settings to any other value than 0, you should be able to define a different lifetime for client sessions.

As you will learn in the *Managing tokens* section, Keycloak also allows administrators to override both the **Client Session Max** and **Client Session Idle** settings on a per-client basis.

As a rule of thumb, the session lifetime should be as short as possible considering security, performance, and user experience aspects. By using a short lifetime, you can reduce the impact of session hijacking attacks or tokens being leaked or stolen. It also avoids overloading the server with sessions that do not show any

user activity and therefore helps to save server resources such as memory and CPU. However, a short session lifetime has a direct impact on user experience and how often users need to re-authenticate. In a user-first approach, you will probably start with what is the best for your users and then adjust the session lifetime according to your security requirements and the constraints you have on hardware and infrastructure resources.

In this topic, you learned about how to manage session lifetime and its impact on user experience, security, and performance. You also learned that during the authentication and token issuance processes, Keycloak may create an SSO session on a per-user basis and client sessions for each client the user is authenticated to.

In the next topic, you are going to learn how to track and manage user and client sessions.

Managing active sessions

Keycloak gives administrators great traceability and visibility of sessions at different levels:

- Per-realm
- Per-client
- Per-user

At the realm level, administrators can look at the statistics on the number of active sessions on a per-client basis. For that, click on the **Sessions** link on the left-side panel:

Sessions						Action ▾
Sessions are sessions of users in this realm and the clients that they access within the session.						Learn more ⓘ
All session types		Search session				1-1 ▾
User	Type	Started	Last access	IP addr...	Clients	⋮
admin	REGULAR	5/2/2023, 12:44:37 PM	5/2/2023, 12:52:52 PM	127.0.0.1	security-admin-console	⋮

Figure 12.3: Managing active sessions in a realm

From this page, you can click on any client to get more details about its active sessions. By selecting a client, you should be redirected to the **Sessions** tab on the client details page. From this page, you can see the active sessions in the client on a per-user basis with additional information about the IP address of the user and when the session was started.

The screenshot shows the 'Sessions' tab selected in the client settings. At the top, there is a status bar with 'security-admin-console' (client name), 'OpenID Connect' (provider), an 'Enabled' toggle switch, a help icon, and an 'Action' dropdown menu. Below the status bar, a message states: 'Clients are applications and services that can request authentication of a user.' A navigation bar below the message includes tabs for 'Settings', 'Roles', 'Client scopes', 'Sessions' (which is highlighted in blue), and 'Advanced'. A search bar labeled 'Search session' is followed by a sorting dropdown showing '1-1'. The main content area displays a table of active sessions:

User	Type	Started	Last access	IP address
admin	Regular SSO	5/2/2023, 12:44:37 PM	5/2/2023, 12:56:43 PM	127.0.0.1

Below the table, another sorting dropdown shows '1-1'.

Figure 12.4: Managing active sessions in a client

By clicking on any **user** on this page, you are redirected to the **user details page**, the third and the last level of visibility for active sessions:

The screenshot shows the 'Details' tab selected for the 'admin' user. At the top, there is a status bar with the user name 'admin', an 'Enabled' toggle switch, and an 'Action' dropdown menu. Below the status bar, a navigation bar includes tabs for 'Details' (which is highlighted in blue), 'Attributes', 'Credentials', 'Role mapping', 'Groups', 'Consents', and 'Identity provider'. A search bar labeled 'Search session' is followed by a 'Logout all sessions' button and a sorting dropdown showing '1-1'. The main content area displays a table of session details:

Started	Last access	IP address	Clients
5/2/2023, 12:44:37 PM	5/2/2023, 1:00:42 PM	127.0.0.1	security-admin-console

Below the table, another sorting dropdown shows '1-1'.

Figure 12.5: Managing sessions for a user

On the user details page, you should click on the **Sessions** tab to look at all active sessions for the user. On this tab, you are given more details about sessions, such as when the session started, the last time Keycloak recorded activity from the **user**, and **all clients – and client sessions – associated with a user session**.

Under normal circumstances, you should expect users to have a single session and many clients. This is especially true in a typical SSO scenario when users are using a browser to authenticate, and the same session is reused to authenticate to different clients.

However, it might happen that users close their browsers, clear their cookies, or just use different devices to authenticate. Under these circumstances, you might have multiple user sessions for a single user depending on how they use your applications.

In this section, you learned how to get more visibility into the sessions active in a realm. You also learned that Keycloak gives you different levels of visibility on a per-realm, per-client, and per-user basis. Finally, you learned that at each level you are given additional information about sessions.

In the next topic, you are going to look at these different levels of visibility in more detail and how to expire sessions prematurely by forcing a single or global logout.

Expiring user sessions prematurely

In addition to providing visibility on active sessions, Keycloak also provides mechanisms for expiring sessions prematurely when at

the different levels:

- Realm
- Client
- User

When looking at the active sessions at the realm level, you can expire all active sessions in a realm by clicking on the **Sign out all active sessions** action from the **Action** selection box:

The screenshot shows the 'Sessions' page in the Keycloak admin console. At the top right, there is a dropdown menu labeled 'Action'. A sub-menu is open, showing 'Revocation' and 'Sign out all active sessions'. Below the header, there is a search bar with 'All session types' and a 'Search session' input field. The main table lists a single session for 'admin'.

User	Type	Started	Last access	IP addr...	Clients	⋮
admin	REGULAR	5/2/2023,12:44:37 PM	5/2/2023,1:02:09 PM	127.0.0.1	security-admin-console	⋮

At the bottom right of the table, there are navigation icons for '1-1' and arrows.

Figure 12.6: Forcing session expiration at the realm level

Upon clicking on the **Sign out all active sessions** action, Keycloak is going to promptly expire all sessions by iterating over all of them and removing their references. On this page, you should also be able to reactively revoke tokens by clicking on the **Revocation** action, as you will learn in the next section.

When at the user level, you can expire individual sessions by clicking on the **Sign out** link or simply expire all sessions by

clicking on the **Logout all sessions** button:

The screenshot shows the Keycloak Admin Console interface for a user named 'admin'. At the top, there is a status indicator 'Enabled' with a blue circle icon. Below it is a navigation bar with tabs: Details, Attributes, Credentials, Role mapping, Groups, Consents, and Identity provider. The 'Details' tab is selected. A search bar labeled 'Search session' is followed by a blue button labeled 'Logout all sessions'. To the right of the search bar, there is a dropdown menu showing '1-1' and arrows. The main table displays user session information with columns: Started, Last access, IP address, and Clients. One row is shown with the following data: Started: 5/2/2023,12:44:37 PM; Last access: 5/2/2023,1:06:11 PM; IP address: 127.0.0.1; Clients: security-admin-console. To the right of this row is a three-dot menu icon. At the bottom right of the table area is a 'Sign out' button.

Figure 12.7: Expiring user sessions

When expiring sessions at the user level, Keycloak might send notifications through the backchannel to applications so that they can also invalidate their local sessions. For more details, look at the documentation at

<https://www.keycloak.org/docs/latest/server-admin/#backchannel-logout> on how to handle logout events in your application.

Note that when using any of the preceding methods to log out all sessions either at the realm or user level, Keycloak iterates over sessions and expires them one by one. At the user level, you should not expect many active user sessions, but probably many client

sessions depending on the number of clients that users are authenticated to. However, expiring sessions at the realm level for every user authenticated in a realm might be an expensive operation.

In this topic, you learned how to prematurely expire user sessions using the Keycloak administration console. You were also presented with some considerations on how to expire sessions using the different options presented herein.

In the next section, you are going to look at how Keycloak uses cookies to track user and client sessions.

Understanding cookies and their relation to sessions

As you know, HTTP is a stateless protocol where cookies are often used to share the state between browsers and servers. Keycloak heavily relies on HTTP cookies to track user sessions when users are interacting with it using a browser.

After successfully authenticating a user, Keycloak sets a `KEYCLOAK_IDENTITY` cookie to correlate a browser session with the corresponding user session on the server. If this cookie is leaked or stolen, users might have their sessions compromised.

The `KEYCLOAK_IDENTITY` cookie is set by default as an `HttpOnly` cookie to prevent **cross-site scripting (XSS)** and session hijacking attacks. Its expiration is based on the value set to the maximum time set for user sessions and its value has enough entropy to prevent guessing attacks.

You can add more security barriers to this cookie, and the most effective is to make sure Keycloak is only accessed through a secure channel using HTTP over TLS (HTTPS). When using HTTPS, the `secure` attribute is set to the cookie to prevent it from being transmitted in cleartext, as well as the `SameSite=none` attribute to make sure the cookie is only sent in cross-site requests through a secure connection. For more details on how to enable TLS, check *Chapter 9, Configuring Keycloak for Production*.

Regarding session expiration, the `KEYCLOAK_IDENTITY` cookie does not automatically expire when using any of the methods presented in the previous topic. Therefore, browsers may still send this cookie, but they are no longer referencing an active session. Upon receiving an invalid cookie, Keycloak is going to invalidate it and force the user to re-authenticate.

In this topic, you got a brief introduction to how Keycloak tracks user sessions using cookies. You also learned that protecting cookies is crucial to keep user sessions secure and how Keycloak helps to enforce strict policies on them.

In this section, you also learned about some key aspects of session management in Keycloak. You learned that Keycloak is constantly creating server-side sessions to track authenticated users and the clients they are authenticated to. You also learned about the importance of properly configuring a session's lifetime and its impact on the user experience, security, and performance of applications and Keycloak. Lastly, you were presented with different options for expiring sessions through the administration

console and how Keycloak leverages cookies to track sessions when users are using a browser.

In the next section, you are going to look at how to manage tokens and their correlation with sessions.

Managing tokens

As you learned from the previous section, tokens are usually bound to sessions. Therefore, the token validity – not necessarily their lifetimes – depends on sessions.

Tokens have their own lifetime and how long they are considered valid depends on how they are validated. By leveraging **JSON Web Token (JWT)** as a format for tokens, Keycloak enables applications to validate and inspect tokens locally without any additional roundtrip to the server. However, this capability has a consequence where tokens, although within their lifetime, might not be valid anymore if their sessions have expired.

Without taking this into account, you might end up in a situation where tokens are no longer valid (the sessions they are bound to have expired) but are still accepted by applications because they are within their lifetime, therefore increasing the attack surface if tokens are leaked. As you are going to learn in this section, you should always consider a clear strategy for token expiration and revocation.

Depending on how applications are requesting tokens from Keycloak, they are given a set of tokens:

- An ID token

- An access token
- A refresh token

As you learned in previous chapters, depending on the authorization grant type the client is using, Keycloak might issue all those tokens or only a subset of them. Each token has its own lifetime.

Unlike refresh tokens (which we'll cover shortly), ID tokens and access tokens share the same lifetime. Both tokens are short-lived and commonly used by public clients (for example, single-page applications) where token storage is not the most secure. In the case of an access token, it is frequently sent over the wire to access protected resources from resource servers and is susceptible to interception. Its lifetime and validity are key factors to reduce the impact when it is leaked or revoked.

On the other hand, refresh tokens have a longer lifetime, and their validity depends on the lifetime set for user and client sessions. This characteristic is what makes it possible to have short lifetimes for ID tokens and access tokens and what makes it possible to refresh these tokens when they expire. By living longer, refresh tokens are the perfect target for attackers, and they also need a clear strategy for expiration and revocation.

As you will see in *Chapter 14, Securing Keycloak and Applications*, you can use additional layers of security to protect tokens from being misused when they leak, such as key rotation. Yet, tokens' lifetimes are crucial

to determine the overall security of your clients and their behavior.

In this section, you were briefly introduced to some basic concepts regarding tokens, such as their lifetime and validity and how they might impact the overall security of applications.

In the next topic, you are going to look at how to manage tokens' lifetimes.

Managing ID tokens' and access tokens' lifetimes

Keycloak allows you to set tokens' lifetimes similarly to sessions. For that, click on the **Tokens** tab when on the **Realm Settings** page. On the **Tokens** tab, you should be able to limit the lifetime for all three tokens mentioned in the previous topic, with the possibility to define specific settings for refresh tokens.

For ID tokens and access tokens, their lifetime can be set by changing the **Access Token Lifespan** setting to any value you want.

By default, Keycloak sets the lifespan for these tokens to only 5 minutes:



Figure 12.8: Setting the ID token and access token lifespan

The name is confusing, but behind the scenes, Keycloak uses the **Access Token Lifespan** setting to also calculate the lifetime for ID tokens.

Keycloak also allows you to override **Access Token Lifespan** on a per-client basis. For that, navigate to the details page of a client and click on the **Advanced** tab:

Advanced Settings

This section is used to configure advanced settings of this client related to OpenID Connect protocol

Access Token Lifespan	Inherits from realm settings	5	Minutes
Client Token Idle	Inherits from realm settings	0	Minutes
Client Token Max	Inherits from realm settings	0	Minutes

Figure 12.9: Overriding the ID token and access token lifetime on a per-client basis

In **Advanced Settings**, set **Access Token Lifespan** to override the lifetime for ID tokens and access tokens for a particular client.

The value you set should be as short as possible to reduce the impact when tokens leak, therefore forcing clients to refresh their tokens. However, a too-short value might also impact the performance of your application and Keycloak itself as you will have more frequent refresh token requests. The default value

should be enough for most use cases, but you can adjust the value according to your needs.

This setting is especially important for access tokens as they are frequently transmitted over the wire as a bearer token for accessing backend services (e.g., resource servers), as per *RFC 6750 – Bearer Token Usage*.

As mentioned before, tokens issued by Keycloak are based on the JWT format, enabling applications to validate tokens without an additional round trip to introspect the token using the token introspection endpoint of Keycloak, but validating the token signature and some standard claims related to the lifetime.

Therefore, and depending on your security requirements, you might not tolerate a situation where the token is within its lifetime, but the refresh tokens are no longer backed by an active session in Keycloak. Under these circumstances, you might want to accept the additional overhead of using the token introspection endpoint in favor of security for critical operations.

Here, the lifetime also has a direct impact on the user experience and on the complexity of clients. Short-lived tokens are usually used together with long-lived refresh tokens to avoid users eventually re-authenticating during a token refresh. Clients using refresh tokens are more complex to implement though due to the additional logic needed to deal with refresh tokens. On the other hand, long-lived tokens help to make clients simpler by removing the need for frequent refreshes with additional risks in case tokens happen to leak. It is up to you to find the right balance that fits your needs.

In this topic, you learned about how to set the lifetime for ID and access tokens. You learned that as a best practice, the lifetime should be as short as possible and what the correlation between short-lived tokens, refresh tokens, client complexity, security, and performance is. Finally, you learned that the validity of tokens can be out of sync with the state of their sessions and how you can avoid that by using the token introspection endpoint instead of performing local validations of tokens.

In the next topic, you will learn how to manage the lifetime of refresh tokens.

Managing refresh tokens' lifetimes

The lifetime of refresh tokens is defined by the idle timeout that you learned about in the previous section to set the lifetimes of user sessions and client sessions, respectively. The same rule applies here to their maximum lifetime so that they are only valid up to the maximum time that you set for user and client sessions.

Firstly, refresh tokens' lifetimes are calculated based on the time set for client sessions, either by setting the **Client Session Idle** setting at the realm level or overriding the same setting on a per-client basis. If a lifetime is not explicitly set for client sessions, then Keycloak will fall back to the value you set for user sessions, through the **SSO Session Idle** setting.

To override the refresh token lifetime on a per-client basis, navigate to the details page of a client and click on the **Advanced** tab:

Advanced Settings

This section is used to configure advanced settings of this client related to OpenID Connect protocol

The screenshot shows the 'Advanced Settings' section of a Keycloak client configuration. It contains three main settings:

- Access Token Lifespan:** Set to "Inherits from realm settings" with a dropdown menu open, showing "5 Minutes".
- Client Token Idle:** Set to "Inherits from realm settings" with a dropdown menu open, showing "0 Minutes".
- Client Token Max:** Set to "Inherits from realm settings" with a dropdown menu open, showing "0 Minutes".

Figure 12.10: Overriding the refresh token lifetime on a per-client basis

In **Advanced Settings**, you can override the refresh token lifetime by setting the **Client Session Max** and **Client Session Idle** settings. Note that by default Keycloak does not define an explicit value for these settings at the client level so that they are implicitly set with the values set at the realm level, as you learned in the previous section.

With regard to refresh tokens, you should consider the following:

- Refresh tokens are always bound to a client session after authenticating users in Keycloak using a specific authorization grant type such as the authorization code grant type.
- Refresh tokens are considered valid if the user and client sessions they are bound with have not expired.
- Clients should be able to use refresh tokens to obtain new tokens only if their respective client sessions are still active.

By taking these three considerations into account, you should be able to realize how refresh tokens are crucial to getting short-lived ID tokens and access tokens, as well as how much they are critical to the overall security of your applications by allowing you to define more strict policies regarding token lifetime on a per-client basis.

Refresh token lifetimes can be adjusted depending on how secure clients can keep their tokens. For instance, a confidential client could have refresh tokens that live longer, whereas for public clients, you might want a smaller window.

Note however that as soon as a refresh token expires, users will be forced to re-authenticate to the client, therefore impacting the user experience, if using a browser.

One of the worst things that might happen is when a refresh token is leaked. This will enable attackers to obtain tokens from Keycloak and gain access to applications by impersonating the client to which the tokens were issued. There are different barriers that you can impose to avoid or reduce the impact when that happens. One of them is refreshing token rotation.

In this topic, you learned about how to manage “refresh token” lifetimes and its importance for enabling short-lived tokens and the overall security of applications.

In the next section, we are going to look at how to enable refreshing token rotation to prevent attackers from reusing refresh tokens.

Enabling refreshing token rotation

As a protection measure for when refresh tokens are leaked, Keycloak allows you to enable refreshing token rotation.

Refreshing token rotation is a strategy to reduce the impact of a refresh token being leaked by invalidating it prior to issuing a new one when a legitimate client is making refresh token requests.

When enabled, rotation helps to quickly identify when the refresh token leaked and forces either the attacker or the legitimate client to re-authenticate to obtain a fresh set of tokens, including a new and valid refresh token. Considering that only the legitimate client can authenticate to the token endpoint, only it will be able to obtain a new refresh token.

To enable refresh token rotation, navigate to the **Tokens** tab on the **Realm Settings** page and enable the **Revoke Refresh Token** setting:

Refresh tokens

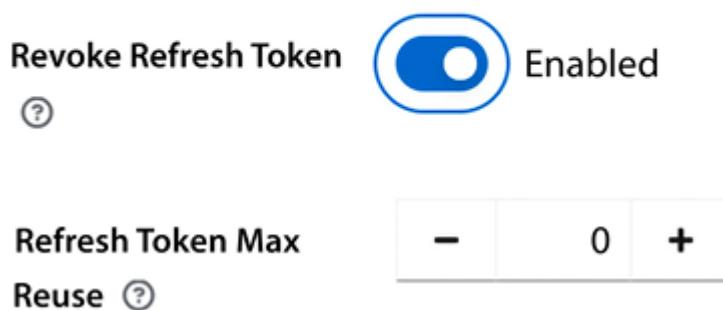


Figure 12.11: Enabling refresh token rotation

By enabling that setting, you are presented with an additional **Refresh Token Max Reuse** setting to define how many times a refresh token can be used by a client until a new one is issued, thus invalidating the previous refresh token.

By default, the **Refresh Token Max Reuse** setting is set to **0** and that effectively means that a refresh token can only be used once. If a client tries to reuse the same refresh token, Keycloak is going to deny the request and force the client to re-authenticate the user. If you increase the value by 1, for instance, it will allow clients to use the same refresh token only twice.

In practice, refreshing token rotation is usually a good practice for reducing the attack surface when refresh tokens are leaked. It is also useful to quickly identify when that happens and react to possible exploits. However, it is not the only security measure you should consider, especially when dealing with public clients.

Public clients are inherently insecure as they do not need to provide any credentials to authenticate to the token endpoint. As such, you should consider using Mutual TLS Client Authentication to enforce the usage of sender-constrained tokens where the client certificate is used to bind the tokens to the client they were issued for, therefore preventing an attacker from using potentially leaked refresh tokens when presenting them to the token endpoint. For more details, check *Chapter 14, Securing Keycloak and Applications*.

In the next topic, you will learn about how to revoke tokens to either simply expire tokens when they are no longer necessary or as a reaction to possible exploits.

Revoking tokens

Keycloak allows you to revoke tokens using different methods. As you learned in the previous section, *Managing sessions*, tokens are

bound with sessions, and when sessions expire, tokens are no longer considered valid by Keycloak.

One of the easiest ways to globally invalidate tokens, regardless of the user and client, is to use a not-before-revocation policy to force tokens to expire based on a set time.

As you also learned in the previous section, Keycloak allows you to manage the active sessions in a realm by clicking on the **Sessions** item in the left-side pane. When on the **Sessions** page, you can click on the **Revocation** action from the **Action** select box to revoke tokens created before a given time:

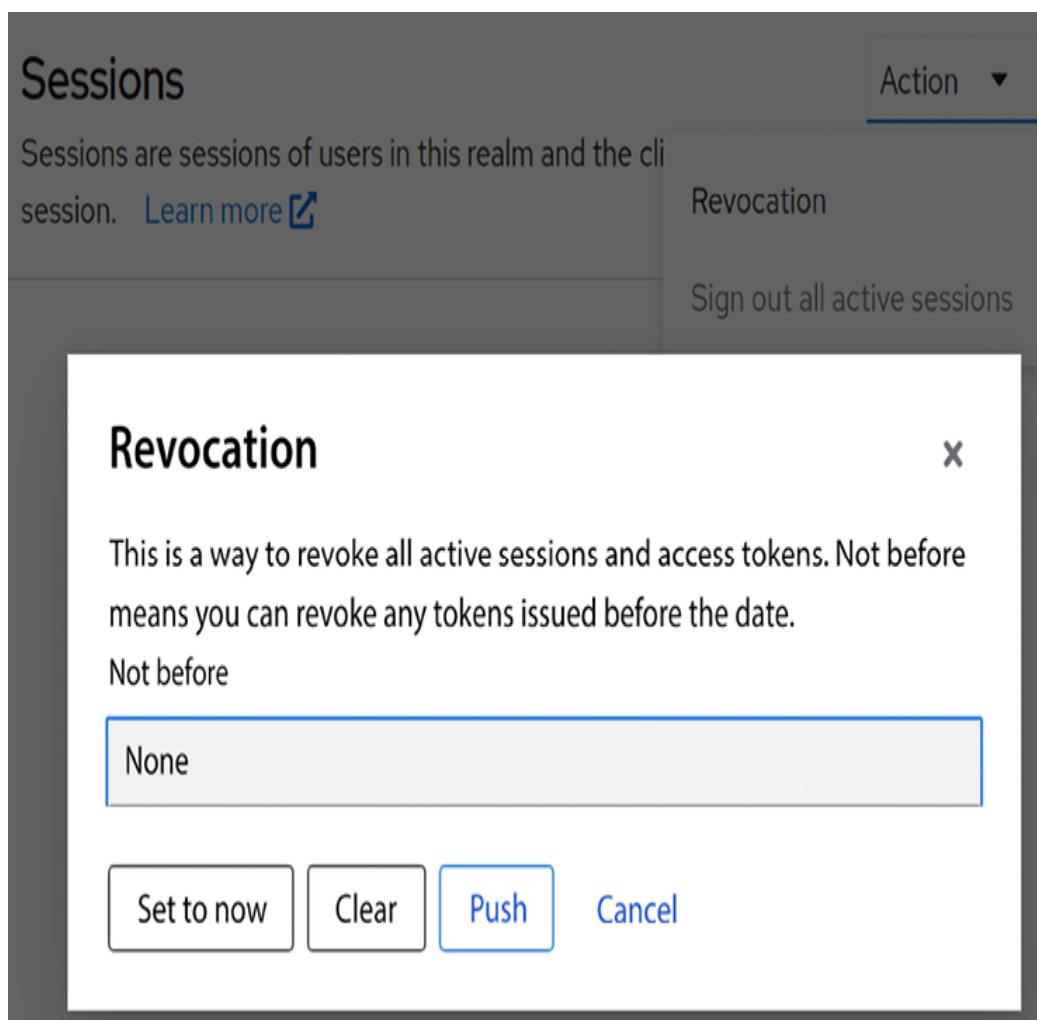


Figure 12.12: Expiring sessions using a time-based revocation policy

By clicking on the **Set to now** button, Keycloak will automatically fill the **Not before** field with the current time and update the realm settings to fail token validation whenever a token was created before the time set.

Note that when revoking tokens using a not-before policy, applications are not immediately informed about the revocation status.

Keycloak also allows you to revoke tokens by either expiring their underlying sessions – user or client sessions – or by using a revocation endpoint as defined per *RFC 7009*.

By revoking tokens using session expiration, administrators should be able to leverage what you learned in a previous section, *Managing sessions*, to automatically invalidate any token associated with a session.

On the other hand, Keycloak also allows clients to revoke their tokens using a specific endpoint, the token revocation endpoint, based on *RFC 7009*. By using this approach, clients can help Keycloak to track unused tokens, reduce the amount of time for which tokens are susceptible to leaking, and clean up any data associated with them to save memory and CPU resources. For more details on how to revoke tokens using the token revocation endpoint, look at the documentation at

https://www.keycloak.org/docs/latest/server_admin/#_oidc-endpoints

Except for a not-before revocation policy, all other methods imply expiring the user and client sessions by either expiring all active sessions in a realm or only client sessions when using the token revocation endpoint. The not-before policy only impacts how tokens are validated; their corresponding sessions are kept alive until they eventually expire.

In this section, you learned about some key aspects of token management and how tokens correlate with session management. Firstly, you were presented with the key concepts and considerations around token lifetime, as well as the different settings to configure it. Then, you learned how refresh tokens enable short-lived access tokens and their importance to the overall security and performance of applications. Lastly, you learned about how refresh token rotation can help to reduce the attack surface when refresh tokens are leaked, as well as about the different methods for revoking tokens.

Summary

In this chapter, you learned about some key aspects of token and session management. By leveraging what you learned from this chapter, you should be able to define clear policies for session expiration and token revocation considering their impacts on security, user experience, and the performance of applications and Keycloak.

In the next chapter, you are going to look at one of the main aspects of Keycloak – extensibility – and how to extend it to adapt and fulfill unmet needs.

Questions

1. Does Keycloak store sessions in the database?
2. What is the difference between user and client sessions?
3. How do you proactively revoke tokens and expire sessions?
4. How can applications validate tokens?
5. How does the access token lifetime impact clients?

Further reading

Refer to the following links for more information on the topics covered in this chapter:

- Keycloak user and session management:
https://www.keycloak.org/docs/latest/server_admin/#user-session-management
- Mutual TLS client authentication:
https://www.keycloak.org/docs/latest/server_admin/#advanced-settings
- Token revocation endpoint:
<https://tools.ietf.org/html/rfc7009>
- Keycloak threat model mitigation:
https://www.keycloak.org/docs/latest/server_admin/#compromised-access-and-refresh-tokens
- OAuth 2.0 threat model and security considerations:
<https://tools.ietf.org/html/rfc6819>
- OAuth 2.0 security best current practice:
https://www.keycloak.org/docs/latest/server_admin/#account-service

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/SecNet>



13

Extending Keycloak

At this point, you should have a good idea of what Keycloak has to offer as an **Identity and Access Management (IAM)** solution. You may also be trying to correlate what you have learned so far with the use cases you need to solve and how to leverage Keycloak capabilities to fit into your requirements.

Although Keycloak offers a rich configuration model that allows you to easily adapt its capabilities according to your needs, it is expected that the standard configuration is not enough to sort out all of them.

Among other questions, you are probably asking yourself how to change Keycloak pages to comply with your own **User Interface (UI)** and **User Experience (UX)** patterns. Or perhaps how Keycloak can leverage and integrate into a legacy database identity store to fetch identity-related data for existing users. Or maybe – and I promise this is my last example – you want to send audit events to a fraud detection system and integrate with it for risk-based authentication.

In this chapter, you will learn how to go beyond any limitation in configuration by extending Keycloak to either customize existing capabilities or add new ones entirely. For that, you will receive an

overview of the design of Keycloak and see why it is the perfect choice to not only quickly deploy IAM to your ecosystem, but also to easily adapt IAM to fit your needs.

For that, we are going to cover the following topics in this chapter:

- Understanding service provider interfaces
- Changing the look and feel
- Customizing authentication flows
- Looking at other customization points

By the end of this chapter, you should be aware of how to leverage customization hooks to change the look and feel of Keycloak, according to your UI and UX requirements, understand the concept of a **Service Provider Interface (SPI)** and the role it plays when it comes to customization, and finally, look at some references and code examples regarding how customizations are implemented and installed.

As a widely used open source project, Keycloak was designed with extensibility in mind, where contributions are made daily to extend or add new capabilities. With the basic learning from this chapter, it is also expected that you will be able to help us to constantly improve and enrich the set of functionalities of Keycloak. For more information about how to contribute to Keycloak, see the guidelines at

<https://github.com/keycloak/keycloak/blob/main/CONTRIBUTING.md>.

Technical requirements

During this chapter, you are going to need a development environment with the **Java Development Kit (JDK) 17** specifications.

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/Keyc:  
▶
```

Alternatively, you can download a ZIP of the repository from <https://github.com/PacktPublishing/Keycloak---Identity-and-Access-Management-for-Modern-Applications-2nd-Edition/archive/main.zip>.

The examples you are going to follow along with in this chapter are available from the following directory within the repository:

```
$ cd Keycloak---Identity-and-Access-Management-for-  
▶
```

For this chapter, you also need to create a `myrealm` realm to follow some examples. You will also need to create an `alice` user to authenticate to the account console when running examples.

Let's start our journey by first looking at what an SPI is and why it is a key concept when extending Keycloak.

Check out the following link to see the **Code in Action** video:
<https://packt.link/WH4wI>.

Understanding service provider interfaces

If you are already familiar with the Java language, you probably know what an SPI is. If not, think about it as a pluggable mechanism to add or change behavior to an extensible Java application without changing its code base.

Keycloak is built with extensibility in mind, where features are implemented using a set of well-defined interfaces. Features such as the ability to authenticate users using different authentication mechanisms, auditing, integration with legacy systems to fetch identity data, mapping claims into tokens, registering new users and updating their profiles, and integrating with third-party identity providers are all backed by a set of service interfaces and a corresponding SPI. The same is also true for core features, such as caching, storage, or the different security protocols supported by Keycloak, although for those, you would hardly have a need to customize:

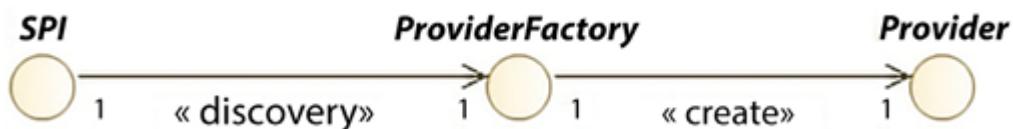


Figure 13.1: Keycloak Service Provider Interface (SPI)

In Keycloak, a feature is defined based on three main interfaces:

- SPI
- ProviderFactory
- Provider

For simplicity purposes, we will now use the term *provider* whenever referring to a combination of a **ProviderFactory** and **Provider** implementation, where a *custom provider* refers to customizations for an existing SPI or feature.

The SPI – according to Java terminology – is a top-level interface to load and describe the different implementations of a feature.

The **ProviderFactory** is a service factory interface and, as the name suggests, defines a contract to manage the life cycle of a particular implementation and create **Provider** instances. A factory is also responsible for defining a unique identifier for itself in the scope of the SPI, to not clash with other provider implementations.

A **Provider** is the actual service interface that you implement to realize a feature. The following is the main interface you will implement to customize existing features or add new ones:

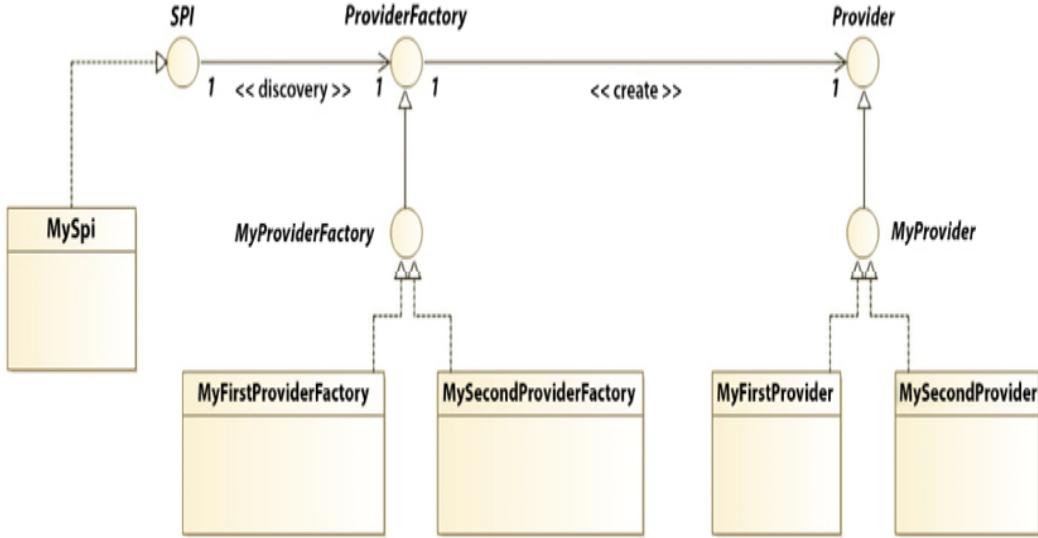


Figure 13.2: Realization of the three main interfaces

By allowing multiple implementations of a feature or **SPI**, Keycloak allows you to create your own implementations and enrich them, by either adding new features or changing their behavior. Let's understand how this translates into practice by looking at the list of **SPIs** and their respective providers. Open the administration console and select the **master realm**. Then, click on the user icon in the top-right corner of the page. Once you click the icon, you will be presented with a sub-menu with an option called **Realm info**:

The screenshot shows the Keycloak administration interface for the 'master realm'. At the top, there's a navigation bar with the Keycloak logo, a help icon, the user name 'admin', and a profile icon. A dropdown menu is open next to the user name, showing options: 'Manage account', 'Realm info', and 'Sign out'. Below the navigation bar, the page title is 'master realm'. There are two tabs: 'Server info' (disabled) and 'Provider info' (selected). A search bar with a magnifying glass icon and a right arrow is positioned above the provider list. The main content area is titled 'SPI' and 'Providers'. It lists providers categorized by SPI:

SPI	Providers
account	freemarker
actionTokenHandler	verify-email execute-actions reset-credentials idp-verify-account-via-email update-email

Figure 13.3: Accessing server runtime information

After clicking on the **Realm info** option, you will be presented with a page with information about the server runtime and a second tab called **Provider info**. Click on the **Provider info** tab to list all the providers installed on the server.

By looking at the list of providers, you can clearly see the realization of the diagrams presented earlier. As an example, type **social** in the input box at the top of the list to filter the results to only those related to integrating with social identity providers:



Figure 13.4: List of the different implementations of social identity providers

As you can see from that list, the **social** SPI has different providers for each social identity provider that you learned from *Chapter 10, Managing Users*. The same applies to any other **SPI**, such as **required-action**, **protocol-mapper**, and so forth.

In this topic, you had a quick overview of Keycloak design, with a focus on extensibility. You learned that Keycloak relies on a set of well-defined interfaces to plug additional features into the server or

change their behavior, and how to obtain information about the available SPIs and their respective providers.

In the next topic, you will look at how to package and deploy your custom providers to the server.

Packaging and deploying a custom provider

Keycloak expects custom providers to be packaged in a **Java Archive (JAR)**. In addition to their classes – the **ProviderFactory** and **Provider** implementations – you need to also include a service descriptor file to allow Keycloak to discover and initialize your custom provider at runtime.

The service descriptor is a regular file placed in the **META-INF/services** directory within the JAR file, and its name is the fully qualified name of the **ProviderFactory** type you implement.

Taking as an example the fictitious **com.acme.MyProviderFactory** factory (yes, we added a package specifically to make a point about using a fully qualified name) used in the diagram in the previous topic, the JAR file should look like this: **mycustomprovider.jar**.

```
META-INF/services/com.acme.MyProviderFactory  
com.acme.MyFirstProviderFactory.class
```

Here, the **META-INF/services/com.acme.MyProviderFactory** file should contain a reference to your implementation of **com.acme.MyProviderFactory**:

```
com.acme.MyFirstProviderFactory
```

Once your provider is packaged in a JAR file, the next step is to install it on the server. For that, you only need to copy the JAR file to the `$KC_HOME/providers` directory and restart the server.

In this topic, you learned how providers are packaged into a JAR and how to deploy them to the server.

Now that you are aware of the structure of the JAR expected by Keycloak, and how to deploy your own providers, we can start looking at the provider's life cycle.

Understanding the KeycloakSessionFactory and KeycloakSession components

Keycloak relies on two main components to manage providers: **KeycloakSessionFactory** and **KeycloakSession**.

We will not dive into detail about their purposes and how they are used, but instead provide you with an introductory understanding before exploring the life cycle of providers, and then move on to this chapter's code examples.

At Keycloak's core is **KeycloakSessionFactory**, which serves as a registry for all providers installed on the server and is responsible

for managing their life cycle. When Keycloak starts, a **KeycloakSessionFactory** is created to initialize and register any provider factories installed on the server. The other way around is also true, where, when Keycloak gracefully shuts down, **KeycloakSessionFactory** gives a last chance to factories to close any resource created during the initialization phase.

On the other hand, when Keycloak is up and running and processing requests, a **KeycloakSession** is created and bound to each request. A **KeycloakSession** is created from the **KeycloakSessionFactory** and serves as an entry point to manage realms, users, clients, and sessions, access contextual information about the current realm and the request, and obtain provider instances. The provider instances obtained from a session are created only once and cached during the lifetime of **KeycloakSession**. **KeycloakSession** is the component that you, as a developer, will use most when implementing providers.

By understanding these two components and what they are, even superficially, you should now be able to follow the next topics and sections whenever we mention these components. Do not worry about trying to understand these components in more detail right now because that will come naturally when you start implementing your own providers, as well as when looking at how other providers are implemented in Keycloak.

In the next section, you will look at the life cycle of providers.

Understanding the life cycle of a provider

Providers have a well-defined life cycle for initialization and deinitialization.

During the installation of providers, Keycloak calls specific methods on the **ProviderFactory** implementations for initialization purposes. The same applies when the server gracefully shuts down, where method calls are made to release resources created during the initialization phase.

That said, the life cycle of a **ProviderFactory** is bound to the server lifetime, where initialization and deinitialization are performed only once, when **KeycloakSessionFactory** is initialized or closed:

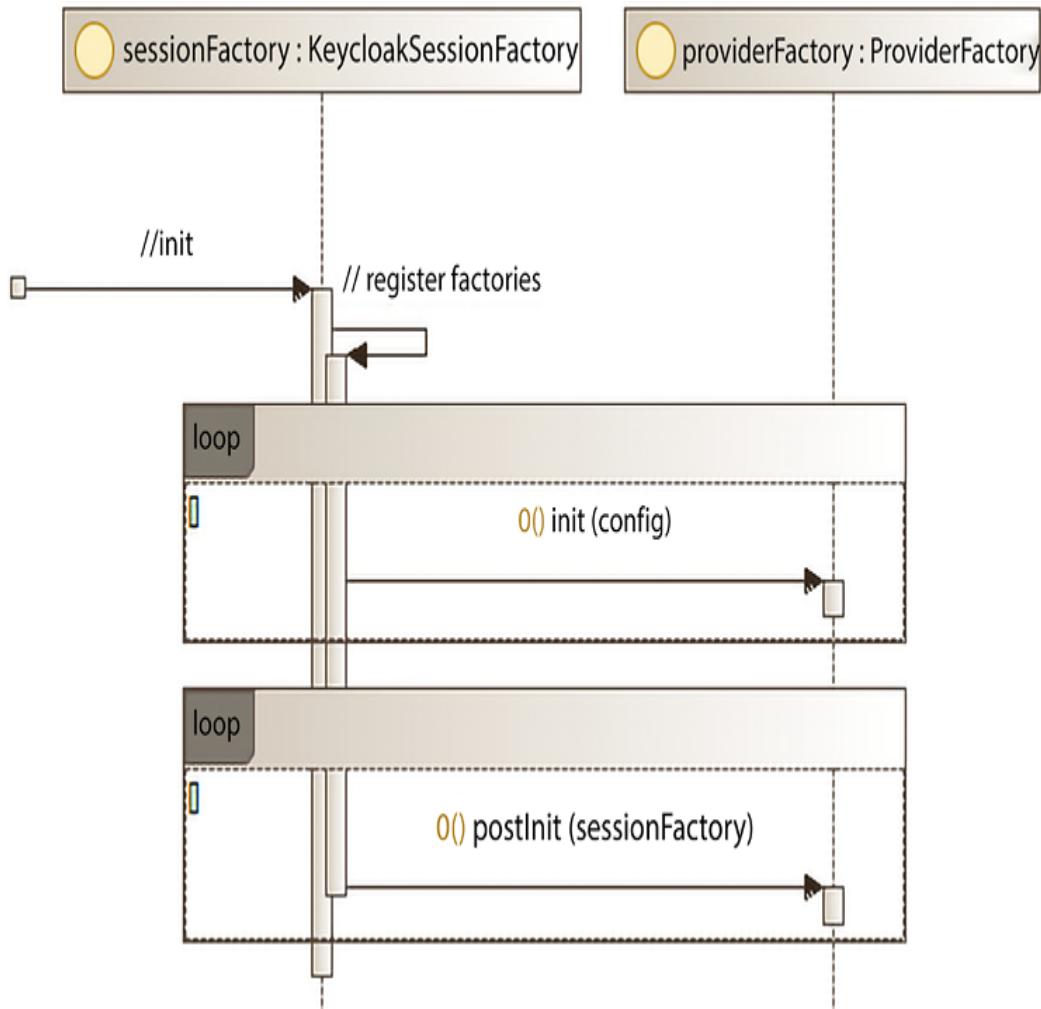


Figure 13.5: Initialization of factories during server startup

The following steps occur during the initialization phase of providers:

1. Keycloak loads all the available factories for each SPI.
2. For each factory, the `ProviderFactory#init` method is called with the provider configuration.
3. Once all factories have been initialized and registered, the `ProviderFactory#postInit` method is called on each factory to perform additional initialization steps based on `KeycloakSessionFactory`.

Regarding deinitialization, the steps are similar, but during this phase, only the **close** method is invoked for each factory.

The `ProviderFactory#init` method is called early when factories are still being registered to initialize them, based on any configuration set to a particular provider. If the factory neither depends on other factories nor **KeycloakSession** to initialize itself, this step should be enough to initialize a factory.

However, the `ProviderFactory#postInit` method is only called once all the factories have been registered, thereby enabling factories to perform additional steps during initialization using other providers and the **KeycloakSession** itself.

On the other hand, the life cycle of a provider is bound to a request. As mentioned earlier, a provider is created from its corresponding **ProviderFactory**, and that happens only once during the request life cycle. Regarding deinitialization, the **close** method on a provider is called at the end of the request:

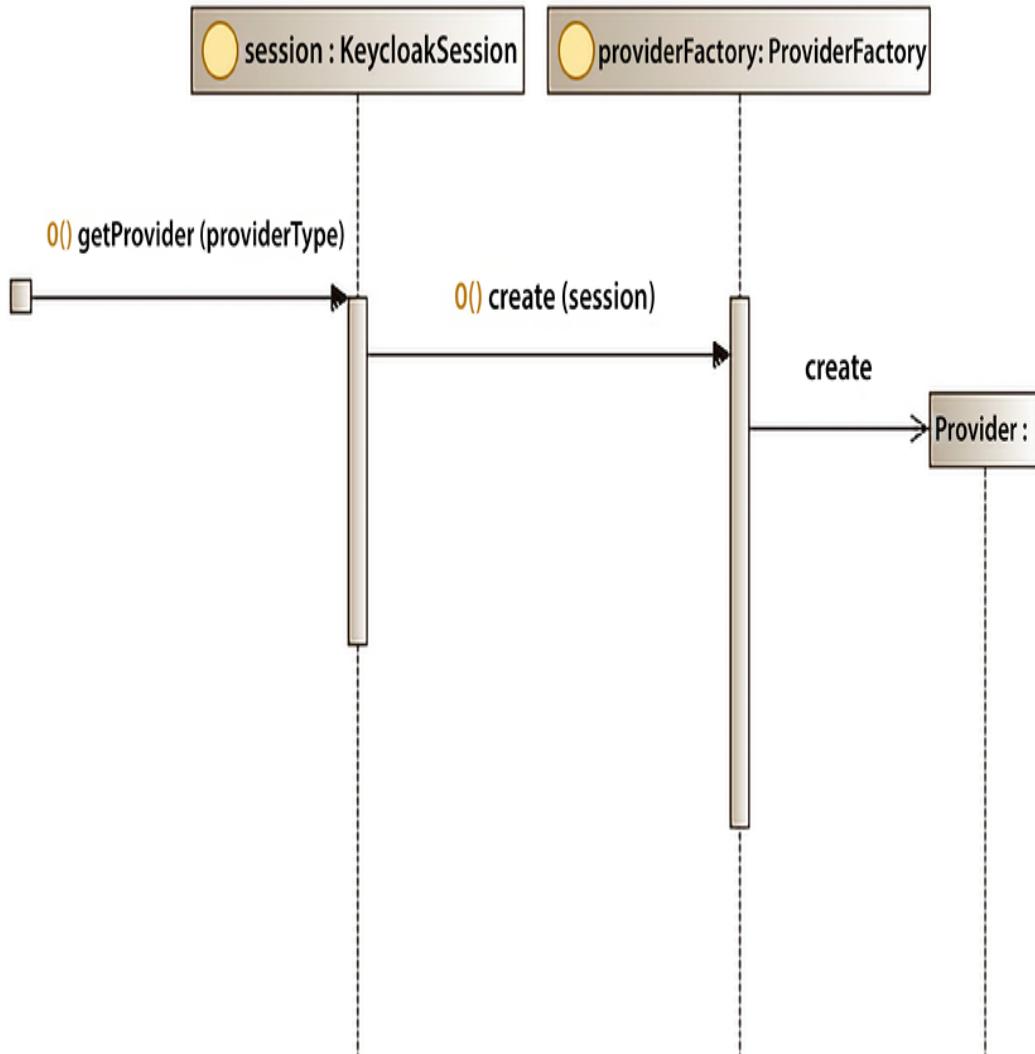


Figure 13.6: Creating a provider instance

In this topic, you learned about the life cycle of providers and how they are initialized either at server boot time or when deploying a provider. You also learned that at runtime, provider instances are created in the scope of a request and bound to a **KeycloakSession**.

In the next sections, you will walk through the steps to choose a capability to extend, implement a provider using an SPI, and install the provider to the server. The examples you are about to see are related to the most common types of customizations that you might

need when changing Keycloak to fit your use cases and requirements.

Changing the look and feel

One of the main customization hooks – and probably what people use most – is changing Keycloak's built-in themes to fit with your branding and respect your UI and UX requirements.

Keycloak provides an amazingly simple experience to change themes and allows you to change most – if not all – of its UI, from the end user-facing pages to the administration console itself.

In this topic, you will learn about the basics of theming by going through examples of how to change the look and feel of the login page. By understanding these basics, you should then be able to apply the same concepts to any other UI you want to customize.

This is probably one of the most documented features of Keycloak. Consider looking at the documentation available at

https://www.keycloak.org/docs/latest/server_development/#_themes

Understanding themes

Just like any other feature in Keycloak, themes are backed by their own SPI. However, instead of having to implement Java code to change themes, Keycloak offers a simple and elegant way of doing

so solely based on plain CSS classes, JavaScript, and any other HTML construct. In fact, depending on your needs, changing themes should be just a matter of defining a new CSS stylesheet.

Another important aspect of themes is internationalization. As an open source project used by organizations all around the world, Keycloak has contributions to support different languages, and it is most likely that users from your country can already benefit from messages in their native language, without any additional effort from your side.

In Keycloak, the built-in themes are included as part of the distribution and loaded from a specific server module. You can install your own themes by either:

- Deploying your themes as a JAR file, similar to how you install a provider
- Copying your theme to the **themes** directory

Regardless of how you install your theme, you must use a specific directory structure so that Keycloak can recognize your theme and install it accordingly.

For example purposes, we are going to extract the contents of the JAR file from where the built-in themes are loaded to understand what the directory structure looks like. To do that, extract the `KC_HOME/lib/lib/main/org.keycloak.keycloak-themes-22.0.0.jar` file into the `KC_HOME/themes` directory:

```
$ cd $KC_HOME
$ unzip lib/lib/main/org.keycloak.keycloak-themes-22.0.0.jar
```

```
$ mv themes/theme/* themes/ && rm -rf themes/theme
```

If you have successfully extracted the contents from the module, you should have a directory structure inside the **themes** directory similar to this:

```
$ cd $KC_HOME\themes
$ ls
├── base
├── keycloak
└── keycloak.v2
```

Each directory in the **themes** directory is a built-in theme with all the necessary configuration and resources to render the pages you have seen so far.

The **keycloak** theme, for instance, is the default theme if no theme is set to a realm. This is the theme you have used in this book when running examples.

The **base** theme is not really a theme but a skeleton used by other themes, where page templates, message bundles for internationalization, and common resources are located. Other themes, such as the **keycloak** theme, extend the **base** theme to define the layout by using specific CSS stylesheets, JavaScript, images, and so on.

Within each **theme** directory, you have sub-directories for the different sets of UIs in Keycloak that are open for customization. Each of these sub-directories represents a *theme type*:

```
$ cd keycloak
$ ls

.
├── account
├── admin
├── common
├── email
├── login
└── welcome
```

By looking at the different theme types and their names, you should have an idea of what you can change in a theme. Let's understand what they are:

- **account**: UI definitions for the account console
- **admin**: UI definitions for the administration console
- **common**: Common resources used across theme types
- **email**: UI definitions for emails
- **login**: UI definitions for login-related pages, including pages for registration, updating profiles, and resetting passwords
- **welcome**: UI definitions for the welcome page

Within each theme type, you have a mandatory file called **theme.properties** in which you define the configuration for a given theme type, for instance:

- Inheriting the configuration from another theme

- Importing resources from another theme
- Custom CSS styles
- Custom JavaScript resources
- Mapping your CSS styles to those used by Keycloak components, such as input boxes and buttons

One important aspect of the configuration is that you are not forced to create one from scratch when customizing a theme type, but you can instead leverage the configuration from another theme and change only what you need to. This is very handy if you just want to make punctual changes to an existing theme.

Now that you know what themes are, where they are located, and how they are structured, let's understand how you set a theme for a realm and a client.

Let's look first at how to define a theme for a realm. To do so, open the administration console and click on the **Realm Settings** item on the left-side panel. Once on this page, click on the **Themes** tab:

The screenshot shows the 'Themes' tab selected in the navigation bar. The page displays four dropdown menus for defining themes:

- Login theme**: Placeholder: Select a theme
- Account theme**: Placeholder: Select a theme
- Admin UI theme**: Placeholder: Select a theme
- Email theme**: Placeholder: Select a theme

At the bottom are two buttons: **Save** (highlighted in blue) and **Revert**.

Figure 13.7: Defining themes for a realm

From the **Themes** page, you can set a theme for any of the available theme types. As you can see, from the select box for each theme type you have a list containing the built-in themes, plus any other custom theme available from a provider JAR file, or from the **themes** directory. Only the themes that define a particular theme type are available from their corresponding select box. By default, if no theme is set to a realm the server will fall back to the **keycloak** theme.

The only exception is the **welcome** theme type, which is defined differently because it is neither specific to a realm nor a client. For more details, look at the documentation at

https://www.keycloak.org/docs/latest/server_development/#configure-theme.

On this page, you also have additional options to manage internationalization. By enabling the **Internationalization Enabled** setting, you are presented with additional options to define which localizations or languages you want to support, as well as a default localization if none can be inferred from the request or based on the authenticated user preferences.

Localization support is feature-rich. By default, messages are loaded from properties files within the **messages** directory for each theme type. As an alternative to message resource bundles in your themes, administrators can also customize messages via the localization support in the administration console. For more details, look at the **base** theme directory as well as the documentation at https://www.keycloak.org/docs/latest/server_admin/#_themes.

Keycloak also allows you to define a theme on a per-client basis, but only for login-related pages. By allowing that, you should be able to provide a customized end user experience, depending on the client they are authenticating to. To define a **login** theme for a client, select a client, and on the client details page, choose a theme from the **Login theme** setting:

Login settings

The screenshot shows the 'Login settings' section of the Keycloak configuration interface. It includes fields for 'Login theme' (with a 'Choose...' button and a dropdown arrow), 'Consent required' (set to 'Off'), 'Display client on screen' (set to 'Off'), and a large text area for 'Client consent screen text' which is currently empty.

Login theme	Choose...	▼
Consent required	Off	<input type="checkbox"/>
Display client on screen	Off	<input type="checkbox"/>
Client consent screen text		

Figure 13.8: Defining a client-specific login theme

In terms of look and feel, Keycloak gives you several options to adapt the login theme accordingly to your needs.

In this topic, you had an overview of themes, how they are configured, and how to set a theme for the different theme types to change their look and feel. In the next topic, we will create a theme from scratch so that you can quickly start creating your own themes according to your needs.

Creating and deploying a new theme

Creating a new theme involves creating a directory, configuring the theme using a `theme.properties` file, and adding any static resource you may require, including CSS stylesheets, JavaScript libraries, and message bundles.

For this topic, you should be able to use a pre-defined theme, available from the GitHub repository in the following directory:

```
$ cd ch13/themes/mytheme/src/main/resources/theme/my
```

In the **mytheme** directory, you have the following structure:

```
login/
└── resources
    ├── css
    └── img
```

The **mytheme** theme was built to change only the **login** theme type. In this example, we will customize only the login page.

When defining a theme type, you should have a standard directory structure as follows:

- The **resources** directory is the place from where Keycloak will look up static resources used by your theme.
- The **messages** directory is the place from where message bundles will be loaded.

Within the **resources** directory, it is best practice to have specific directories for each type of resource you need in your theme, such as CSS stylesheets, JavaScript, and image files.

In our example, the **messages** directory is empty because you will not need to define message bundles

and instead rely on those already available from the **base** theme.

As mentioned earlier, theme types must have a `theme.properties` file to define their configuration. In our example, this file contains only the basic settings to change the login page using custom CSS styles. Let's understand how this file is defined by opening the file that is located in the GitHub repository at `ch13/themes/mytheme/src/main/resources/theme/mytheme/login/theme.properties`:

```
# Inherit resources and messages from the keycloak theme
parent=keycloak
# Define the CSS styles
styles=css/login.css css/bootstrap.min.css css/signin.css
# Mapping CSS classes from Keycloak to custom CSS classes
kcHtmlClass=login-page
kcLoginClass=form-signin
```

From the preceding snippet, you can see that `mytheme` extends the `keycloak` theme and defines some additional CSS stylesheets. In this file, there is also a mapping between the Keycloak CSS classes to those defined in a custom CSS stylesheet at

`ch13/themes/mytheme/src/main/resources/theme/mytheme/login/css/signin.css`.

Now, let's use this theme when we authenticate to the account console. For that, you need to deploy the theme to the server by

building the example project and deploying the JAR:

```
$ cd ch13/themes/mytheme  
$ ./mvnw clean package  
$ cp target/mytheme.jar $KC_HOME/providers
```

When creating themes, you usually want to disable caching so that changes you make to your theme are automatically reflected at runtime. By default, Keycloak caches templates and theme configuration for performance reasons.

To disable caching, you can start the server in development mode as follows:

```
$ cd $KC_HOME/bin  
$ ./kc.sh start-dev
```

After performing these steps, log in to the administration console as the administrator user. Once you are in the console, select the **account-console** client from the list of clients, and then, on the client details page, select the **mytheme** theme from the list of options for the **Login Theme** setting. At the end, the **account-console** client settings should look as follows:

Login settings

The screenshot shows a configuration interface for 'Login settings'. At the top, there is a dropdown menu labeled 'mytheme' with a downward arrow icon. Below it, there are two toggle switches: 'Consent required' (set to 'Off') and 'Display client on screen' (set to 'Off'). At the bottom, there is a section titled 'Client consent screen text' which contains a large, empty text area.

Login theme	mytheme
Consent required	Off
Display client on screen	Off
Client consent screen text	(Large empty text area)

Figure 13.9: Defining the mytheme theme as the login theme for the account console

Now, log out from the administration console and try to log in to the **account console** by opening

`http://localhost:8080/realm/myrealm/account`. If

everything is properly configured, the login page should have a different look and feel, as follows:

MYREALM

English v

Sign in to your account

Username or email

Password

[Forgot Password?](#)

[Sign In](#)

New user? [Register](#)

Figure 13.10: The new layout for the login page when logging in to the account console

In this topic, you learned about how to create and deploy a theme to Keycloak. For that, you were provided with an example from the GitHub repository that changes the layout of the login page. You also learned that when creating a new theme, you usually want to disable caching so that any change you make to a theme is reflected when reloading the server pages.

In the next topic, you will understand how to go even further with your customizations by changing the built-in page templates.

Extending templates

Sometimes, extending a theme using only CSS styles is not enough, and you want to change the disposal of components in the page templates from the **base** theme.

Keycloak relies on **Apache FreeMarker**, a well-known and widely used template engine, to render pages based on templates. By leveraging the templates from the **base** theme, you have a powerful tool to completely change Keycloak pages.

For that, you only need to copy a template from one of the theme types in the **base** theme and include it in your own theme type.

However, this flexibility has a cost, where any change to the built-in templates between Keycloak releases should be manually applied to your own custom templates. This approach to customizing themes is very handy, but it would require a bit more knowledge from you about how Keycloak defines these templates.

For more details about how to extend a built-in template, look at the documentation at

https://www.keycloak.org/docs/latest/server_development/#html-templates.

Extending theme-related SPIs

Included in this section are code examples to customize how themes are selected, as well as how to add custom templates and resources using an SPI.

The code is located in the `ch13/themes/mytheme` directory in the GitHub repository. By looking at the following two classes:

- ch13/themes/mytheme/src/main/java/org/keycloak/book/ch13/theme/MyThemeSelectorProvider
- ch13/themes/mytheme/src/main/java/org/keycloak/book/ch13/theme/MyThemeResourceProvider

You should be able to see the realization of what you learned in the *Understanding service provider interfaces* section, but now applied to themes. These two providers are related to the `themeSelector` and `themeResource` SPIs, respectively.

`MyThemeSelectorProvider` is an example of how to dynamically select a theme at runtime where, depending on your requirements, you may need to choose a theme based on information pertaining to the request, the client, or the user authenticating to Keycloak. This provider has a simple logic to select a theme depending on the value of a request query parameter.

The steps to deploy the provider are the same as when you deployed the `mytheme` theme:

```
$ cd ch13/themes/mytheme  
$ ./mvnw clean package  
$ cp target/mytheme.jar $KC_HOME/providers
```

After updating the provider JAR file, make sure to restart the server.

To see it in action, remove any theme you have set for the realm or client and then try to log in to the administration console again.

At the login page, append the `&theme=mytheme` query parameter to the URL and reload the page. After doing that, you should be able to see the login page layout from the `mytheme` theme, even though no theme was set to the realm or a client. Also, note that the theme is applied regardless of the realm.

On the other hand, `MyThemeResourceProvider` is an example of how to load additional templates and resources to any theme. For this provider, we will not resolve any additional templates or resources, but just give you the baseline if you require this level of customization. This provider can be handy when you have customizations that require additional pages, such as when creating custom authenticators or required actions.

In this topic, you were provided with some code examples of how to customize theme selection and add additional pages by leveraging the `themeSelector` and `themeResource` SPIs, respectively.

In this section, you learned about Keycloak themes and how to use them to change the look and feel of the different server pages. For that, you learned about how themes are created and configured, how they are deployed, and how to go even further with customizations by changing the built-in page templates. Finally, you were given some code to demonstrate how to dynamically select themes at runtime, as well as how to add additional pages that can be used from custom authenticators or required actions.

In the next section, you will look at an example of how to leverage the Authentication SPI to customize how users are authenticated using a second factor.

Customizing authentication flows

As you learned from *Chapter 11, Authenticating Users*, Keycloak allows you to easily customize user authentication by changing authentication flows through the administration console.

Eventually, the built-in authentication executions might not be enough to address your authentication requirements, and, in this case, you can leverage the Authentication SPI to implement your own authentication executions.

We are not going to cover in this section all the details pertaining to the Authentication SPI, but instead give you a code example to help you understand the steps and mechanics when you create your own authenticators. The code example for this topic is available from the GitHub repository at [ch13/simple-risk-based-authenticator](#).

The example here is about a simple authenticator that relies on a risk score to determine whether the user should provide a second factor when authenticating. The risk score is calculated based only on the number of failed login attempts, where, if a user fails to log in three times in a row, next time they will be forced to provide a **One-Time Password (OTP)** as a second factor. However, you could leverage this example for something more complex where the risk analysis could also consider other factors, such as the device the user uses, the location, or even the score from an external fraud detection system.

To install the custom authenticator, you need to deploy the provider's JAR file as follows:

```
$ cd ch13/simple-risk-based-authenticator  
$ ./mvnw clean package  
$ cp target/simple-risk-based-authenticator.jar $KC_
```

After deploying the provider, make sure to restart the server.

You will now use what you learned in *Chapter 11, Authenticating Users*, to configure a new authentication flow as follows:

1. Create a copy of the **Browser** flow and name it **My Risk-Based Browser Flow**.
2. Delete the **OTP Form** execution from the **My Risk-Based Browser Flow Browser - Conditional OTP** sub-flow. Make sure that the sub-flow is marked as **REQUIRED**.
3. Add the **My Simple Risk-Based Authenticator** execution to the **My Risk-Based Browser Flow Browser - Conditional OTP** sub-flow.
4. Add the **Conditional OTP Form** execution to the **My Risk-Based Browser Flow Browser - Conditional OTP** sub-flow. Make sure that this execution is marked as **REQUIRED**.
5. Click on the **Action** select box and then click on the **Bind flow** action to associate the **My Risk-Based Browser** flow with the **Browser** flow.
6. Now, click on the **configuration icon** for the **Conditional OTP Form** execution:

Conditional OTP Form config

X

Alias * ⓘ

conditional-otp

OTP control User Attribute ⓘ

my.risk.based.auth.2fa.required

Skip OTP for Role ⓘ

Select Role

Force OTP for Role ⓘ

Select Role

Skip OTP for Header ⓘ

Force OTP for Header ⓘ

Fallback OTP handling ⓘ

force



Save

Cancel

Figure 13.11: Configuring the Conditional OTP Form execution

On this page, you should provide the following configuration:

- **Alias:** conditional-otp
- **OTP control User Attribute:**
my.risk.based.auth.2fa.required
- **Fallback OTP handling:** force

Once you are done, click on the **Save** button.

At the end, the new **My Risk-Based Browser Flow** should look like this:

My Risk-Based Browser Flow Not in use



Figure 13.12: The final configuration for My Risk-Based Browser Flow

Finally, let's enable the **Brute Force Detection** feature for the realm. This feature is responsible for tracking failed login attempts and avoiding brute-force attacks when an attacker tries to guess users' passwords. The custom authenticator we just configured relies on this feature to track the number of failed login attempts. To enable the feature, click on the **Realm Settings** item on the left-side panel. Once on this page, click on the **Security Defenses** tab and then on the **Brute force detection** sub-tab. Once at this tab, turn on the **Enabled** setting:

myrealm

Realm settings are settings that control the options for users, applications, roles, and groups.

◀ Email Themes Keys Events Localization Security defenses

Headers Brute force detection

Enabled On

Max login failures ?

Permanent lockout Off

Wait increment ? Minutes

Max wait ? Minutes

Failure reset time ? Hours

Quick login check milliseconds ?

Minimum quick login wait ? Minutes

Figure 13.13: Enabling brute-force detection for the realm

For more details about the brute-force detection feature, take a look at the documentation at [https://www.keycloak.org/docs/latest/server-admin/#password-guess-brute-force-attacks.](https://www.keycloak.org/docs/latest/server-admin/#password-guess-brute-force-attacks)

Let's now log in to the account console using the `alice` user. To do so, open your browser at

`http://localhost:8080/realm/myrealm/account` and log in using the user's credentials.

At this moment, you should be able to authenticate to the account console by providing only a password.

Now, log out from the account console, and when on the login page, try to log in again but with an invalid password. Repeat this step three times.

On the fourth occasion, provide the correct password of the user. If everything is correct, you should then be asked to configure an OTP to authenticate and access the account console. Next time you try to authenticate after failing to log in three times, you should also be asked for the OTP.

The Authentication SPI gives you the main customization hooks to adapt Keycloak to better fit it into your authentication requirements. Based on what you learned from *Chapter 11, Authenticating Users*, you should be able to add your own authentication executions and required actions, add additional steps during user authentication, and so on. For more details,

consider looking through the documentation available at
https://www.keycloak.org/docs/latest/server_development/#_auth_spi.

In this section, you were provided with an example of how to leverage the Authentication SPI to create your own authenticator providers.

In the next section, you will be provided with additional references to other customization points provided by Keycloak.

Looking at other customization points

In the previous sections, you learned about just a subset of the extension points that you have available in Keycloak. As mentioned earlier, Keycloak is built around the concept of SPIs and there are many other customization points that you might find useful.

The best source for querying the available SPIs is the documentation available at

https://www.keycloak.org/docs/latest/server_development.

Some key SPIs are also covered by examples in the Keycloak Quickstart repository available at

<https://github.com/keycloak/keycloak-quickstarts/>.

From the documentation, you may be interested in looking at the following SPIs:

- User Storage
- Event Listener

The User Storage SPI allows you to integrate Keycloak with any external identity store. A common use case for it is to fetch identity data from an existing database:

- Documentation:
https://www.keycloak.org/docs/latest/server_development/#_user-storage-spi
- Quickstart: <https://github.com/keycloak/keycloak-quickstarts/tree/latest/extension/user-storage-jpa>

The Event Listener SPI allows you to customize how to handle events fired by Keycloak so that you can integrate it with your audit or fraud detection systems:

- Documentation:
https://www.keycloak.org/docs/latest/server_development/#_events
- Quickstart:
 - <https://github.com/keycloak/keycloak-quickstarts/tree/latest/extension/event-listener-sysout>
 - <https://github.com/keycloak/keycloak-quickstarts/tree/latest/extension/event-store-mem>

As you learned at the beginning of the chapter, a list of all the available SPIs can be queried from the administration console on the **Provider Info** page. You should be able to implement customizations for any SPI listed there.

However, most of them are still considered internal SPIs and lack documentation. For these SPIs, the best you can do is look at Keycloak's code base to understand how they are implemented.

In this section, you were provided with some final references and considerations regarding other customization points not covered by this chapter.

Summary

In this chapter, you learned about one of the main aspects of Keycloak: extensibility. You learned that Keycloak not only helps you to deploy IAM to your ecosystem but also adapts IAM to your needs.

To understand this, you were taught the basics of how to change the look and feel of a server using themes and how to implement custom providers using some of the available SPIs. Although you were presented with only a few examples of how to extend Keycloak, you should be able to leverage what you learned from this chapter to extend Keycloak using any SPI.

In the next chapter, you will look at some security best practices and considerations when using Keycloak.

Questions

1. What is a public and private SPI?
2. How do I deploy my extensions?
3. Can I change the look and feel of Keycloak

4. Is it possible to customize how users authenticate to Keycloak?
5. Do I need to be a Java developer to extend Keycloak?

Further reading

For more information on the topics covered in this chapter, you can visit the following links:

- Configuring providers:
<https://www.keycloak.org/server/configuration-provider>
- Keycloak Server Developer Guide:
https://www.keycloak.org/docs/latest/server_development
- Apache FreeMarker: <https://freemarker.apache.org/>
- Java service provider interface:
<https://docs.oracle.com/javase/tutorial/ext/basics/spi.html>
- Keycloak GitHub repository:
<https://github.com/keycloak/keycloak>

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/SecNet>



14

Securing Keycloak and Applications

In this chapter, we will look at how to secure Keycloak for production environments. Then, we will look at how to secure the database, as well as how to secure cluster communication between Keycloak nodes. Finally, we will touch on some topics regarding how you can protect your own applications against threats.

After reading this chapter, you will have a good understanding of how to securely deploy Keycloak, including what is required to secure the database. Since this is a book about Keycloak and not about application security, you won't become an expert on application security, but if this is a topic that's new to you, you will have a basic understanding and an idea of how to learn more.

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

- Securing Keycloak
- Securing the database
- Securing cluster communication
- Securing applications

Securing Keycloak

In this section, we will look at some important aspects of securing the Keycloak server itself. We will start by looking at an example of a secure Keycloak deployment, as shown in the following diagram:

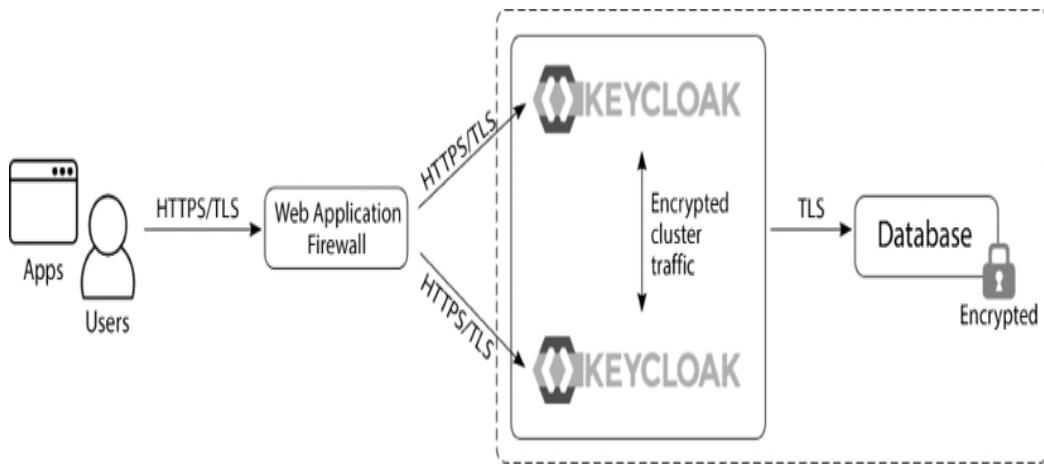


Figure 14.1: An example of a secure deployment

In this example, Keycloak and its database are isolated from users and applications with a **Web Application Firewall (WAF)**, all network requests are encrypted, and the database is also encrypted. Let's look at this in a bit more detail, starting with why **Transport Layer Security (TLS)** is a requirement for any ingoing and outgoing traffic to Keycloak.

Encrypting communication to Keycloak

It is recommended to use end-to-end encryption for all communication to and from Keycloak. This means always using HTTPS, and never using HTTP. At the time of writing this book, the most recent security layer in HTTPS is TLS 1.3, so this is what you

should use whenever possible. Most HTTP libraries will support at least TLS 1.2. If they do not support this, you should consider not using the library since TLS 1.2 has been around since 2008.

If you're leveraging a load balancer or reverse proxy in front of Keycloak, the most secure approach is to leverage TLS passthrough, which provides end-to-end encryption between the client and Keycloak.

In some cases, this may be infeasible, in which case you can use reencrypt, where the communication between the proxy and Keycloak is encrypted again with an internal certificate.

Using unencrypted communication between the proxy and Keycloak should be avoided and only considered if the network between the proxy and Keycloak can be fully isolated, such as in cases where the proxy and Keycloak reside on the same machine.

In the next section, we'll look at why Keycloak needs to know the URL on which it will be exposed.

Configuring the Keycloak hostname

Keycloak is required to know its `public hostname` for several reasons, such as when you're sending an email to a user. Out of the box, for convenience, Keycloak infers the hostname from the request by looking at the `Host` HTTP header sent by the client. This configuration should never be used in production deployments as it could allow an attacker to send requests to Keycloak with a different value for the header.

An example of such an attack is where an attacker uses the **Recover Password** capability in Keycloak and modifies the host header, resulting in the email that's being sent to the user containing a link to a site controlled by the attacker. Unless the users realize the URL is not correct, the attacker can intercept the request to update the password. By intercepting the request, the attacker can either obtain the updated password or set a different password. In either case, the attacker gains access to the user account.

To prevent this type of attack, you can either configure a fixed hostname for Keycloak or, if Keycloak has been exposed through a reverse proxy, you can verify the host header at the reverse proxy. Configuring a fixed hostname for Keycloak is the simplest and most secure approach. To learn how to do that, read *Chapter 9, Configuring Keycloak for Production*.

In the next section, we will look at the importance of regular key rotation.

Rotating the signing keys used by Keycloak

Regularly rotating all your signing and encryption keys is highly recommended. You may want to consider doing so as frequently as once a month.

Luckily, Keycloak allows you to rotate keys in a way that is seamless and non-interrupting. This is because new keys can be made active, while the old keys are still permitted to verify tokens for a while.

Rotating keys has several benefits, such as the following:

- Reduces the amount of content that's signed or encrypted with a specific key.
- Reduces the time available to anyone who wants to try and crack your keys.
- Cleans up unused refresh tokens or long expiration access tokens, regardless of user session timeout settings.
- If an attacker were able to gain access to the keys or – even worse – such a leak was not discovered, the impact would be reduced.

To rotate the signing keys in Keycloak, open the administration console of Keycloak in your browser. Select the realm you want to rotate keys for, go to **Realm Settings**, and click on **Keys**.

First, you will see all the active signing keys in the realm, as shown in the following screenshot:

The screenshot shows the 'Keys' tab selected in the navigation bar. Below it, the 'Active keys' section is displayed. There are four rows of key information:

Algorithm	Type	Kid	Provider	Public keys
RSA-OAEP	RSA	rj0VU554iTBlAZzPBDt4y_XGL3KJAYRdzkK99aMeKQ	rsa-enc-generated	Public key Certificate
AES	OCT	c42814bb-8b61-4a74-926f-aeb590e5c50	aes-generated	
RS256	RSA	2csknl6J3mAYjNUgpvGjIC_378VH0hiStcJ_JvCOV9c	rsa-generated	Public key Certificate
HS256	OCT	c2e33f29-6f2c-4acf-b657-7141b4d313d5	hmac-generated	

Figure 14.2: Active signing keys

The screenshot shows that the realm currently has four keys for different algorithms. Keys in Keycloak can have three different states:

- **Active:** Active keys are used to sign new tokens, where the highest priority key is used for a specific algorithm.
- **Passive:** Passive keys are not used to sign new tokens but are used to verify previously signed tokens.
- **Disabled:** Disabled keys are keys that are not currently in use.

To rotate the keys, the first step is to create a new active key. If, after creating a new active key, the previous key is disabled, all

active user sessions and tokens will be invalidated. This approach should only be taken if you suspect the keys may have been compromised. Otherwise, it is better to change the previous key to **passive** for a period to allow all user sessions and tokens to be updated with the new signing keys, before deleting the old key.

Creating additional keys is done by configuring additional key providers. Click on the **Providers** tab; then, under **Add provider**, select **rsa-generated**. Fill in the form with the values shown in the following screenshot:

The screenshot shows a modal dialog box titled "Add provider". The form contains the following fields:

- Name**: rsa-generated-2
- Priority**: 200
- Enabled**: On (radio button selected)
- Active**: On (radio button selected)
- Key size**: 2048
- Algorithm**: RS256

At the bottom of the dialog are two buttons: "Save" (highlighted in blue) and "Cancel".

Figure 14.3: Creating a new signing key

After creating the key, go back to the **Active** tab. You will notice that there are now two signing keys for **RS256**, as shown in the following screenshot:

The screenshot shows the Keycloak administration interface with the 'Keys' tab selected. A search bar at the top right contains 'RS256'. Below it, a table lists two keys. Both keys are of type RSA and algorithm RS256. The first key is associated with the provider 'rsa-generated-2' and has two buttons: 'Public key' and 'Certificate'. The second key is associated with the provider 'rsa-generated' and also has 'Public key' and 'Certificate' buttons. The table has columns for Algorithm, Type, Kid, Provider, and Public keys.

Algorithm	Type	Kid	Provider	Public keys
RS256	RSA	HN5qkKYqN5Tj0l1re4r4P56TzicWViFRjq\$jh-hd7Wo	rsa-generated-2	Public key Certificate
RS256	RSA	2csknI6J3mAYjNUgpvGjIC_378VHohIStcJ_JvCOV9c	rsa-generated	Public key Certificate

Figure 14.4: Multiple signing keys for the same algorithm

Since the new key you created has the highest priority, it will be used to sign new tokens. Keycloak will automatically re-sign cookies and tokens with the new keys, which will be transparent to users and applications.

By default, Keycloak stores private keys in the database. Combined with good database security and regular key rotation, this is usually acceptable.

For additional security, Keycloak supports storing keys in an external store. At the time of writing this book, Keycloak can load keys from a Java keystore. There is also an extension mechanism that allows you to implement a custom source for keys.

Understanding how to develop custom providers for Keycloak was covered in *Chapter 13, Extending Keycloak*.

For the highest level of security, you could also consider using an external service such as a **Hardware Security Module (HSM)** for signing tokens. Out of the box, Keycloak does not currently support any such integrations but does have extension points that allow you to develop custom providers yourself.

Next, we will look at the importance of regular updates.

Regularly updating Keycloak

Potentially one of the best sources of inspiration for an attacker comes from known vulnerabilities in unpatched software. If you do not regularly update Keycloak or the operating system, the list of unpatched known vulnerabilities the attacker can try out becomes longer and longer.

It is especially important that you have a process in place to be able to discover new releases and quickly upgrade.

One thing to note here is that Keycloak does not have long-term supported versions. Instead, it uses a continuous delivery model or a rolling release. If there are any issues, then continuously upgrading Keycloak means that there are significantly fewer changes you need to make, so you are dealing with bite-sized chunks at a time.

Dealing with continuous releases does, in most cases, require automating the upgrade process, as well as being able to quickly test if an upgrade has any impact on your production systems.

If you prefer a long-term supported version, Red Hat offers Red Hat Single Sign-On, which is essentially a long-term supported version of Keycloak. At the time of writing this book, the most current version of Red Hat Single Sign-On is 7.6, which is based on Keycloak 18, and is continuously receiving security and bug patches. You can find more information about Red Hat Single Sign-On at <https://access.redhat.com/products/red-hat-single-sign-on>.

Future versions of Red Hat Single Sign-On will be rebranded to Red Hat Build of Keycloak. Even though Red Hat Single Sign-On was based on Keycloak, there are some minor differences, such as theming and configuration options. Red Hat Build of Keycloak, on the other hand, will not have any such changes, with the only difference being that it is built and supported by Red Hat.

In the next section, we will look at using an external vault to store secrets.

Loading secrets into Keycloak from an external vault

There are some use cases where you need to provide Keycloak with credentials to access external systems, such as connecting to a database, an email server, or federating users from a directory server. By default, Keycloak stores these credentials in configuration files or in the database, but it can also retrieve these from an external vault.

At the time of writing this edition of the book, Keycloak has support for Kubernetes/OpenShift secrets, as well as the ability to use an encrypted Java KeyStore to securely store credentials.

As we mentioned previously, Keycloak has an extension mechanism that allows you to integrate it with any external vault. To learn more about extending Keycloak, please refer to *Chapter 13, Extending Keycloak*.

For more information on using Kubernetes Secrets with Keycloak, please refer to the *Using Kubernetes Secrets guide* at <https://www.keycloak.org/server/vault>.

In addition, refer to the release notes for new Keycloak releases to find out when a local encrypted vault is available.

Protecting Keycloak with a firewall and an intrusion prevention system

At a minimum, it is a good idea to leverage a firewall to control incoming and outgoing traffic to Keycloak. If possible, you should also consider completely separating Keycloak and its database from even internal applications.

With regard to incoming traffic, this can include limiting incoming traffic to only accepting HTTPS. You may also want to consider only allowing access to the Keycloak admin console and admin REST APIs from an internal network.

For outgoing traffic, it may be a little bit more difficult, depending on your use case. Some outgoing traffic you may need to permit

includes the following:

- Backchannel requests over HTTPS to applications, such as logout requests.
- Connections to user federation providers, such as LDAP.
- Backchannel requests to external identity providers, such as an OpenID token request.

If you are only securing internal applications with Keycloak, it is most likely simpler to secure outgoing traffic, but this may be harder if you are also securing third-party applications that have been deployed outside your network.

It may also be a wise decision to utilize an intrusion prevention (or detection-only) system. An intrusion prevention system can be a great tool for detecting and preventing bad traffic, including helping you survive a denial-of-service attack.

For additional security, it can also be a good idea to leverage a WAF. It is relatively complex to set up a WAF properly and it may need to be updated regularly, but if this is done correctly, a WAF can provide an extra layer of protection against attacks.

Next, we will look at arguably one of the most important aspects of securing Keycloak, which is protecting the database.

Securing the database

Keycloak stores a lot of sensitive data in its database, which makes it especially important to secure it, thus preventing attackers from accessing or modifying the databases.

Some examples of the data Keycloak stores include the following:

- Realm configuration
- Users
- Clients

If your database became compromised, we must consider some examples of what could happen if an attacker were able to read your data:

- An attacker would get access to details about your employees or customers. The impact of this would depend on how much personal information you store about your users, but even a list of email addresses is valuable to an attacker.
- An attacker would get access to user credentials. Even though passwords are stored as one-way salted hashes in the database, the attacker may be able to crack some of the less secure passwords.
- If you are not using a vault or keystore, an attacker would have access to any secrets stored in the database, such as LDAP bind credentials, SMTP passwords, and even the private signing keys used by Keycloak.

These are only a few examples, but attackers are usually highly creative and can come up with all sorts of ways to exploit your data.

An important point to stress here is that an attacker does not care if they get the data directly from the database, or if they get it from a backup of the database, which makes it just as important to secure backups of the database as securing the database itself.

It would be potentially even worse if an attacker managed to gain access to write to the database, as this could give an attacker the ability to access any application secured by Keycloak – they would be able to alter realm configuration or user credentials to impersonate users.

It is beyond the scope of this book to cover database security in detail, but we will briefly look at some best practices, starting with using a firewall.

Protecting the database with a firewall

The first and most obvious thing to do when you're securing your database is to protect it with a firewall. All traffic should be denied by default, and only required access such as from the Keycloak servers should be permitted.

In addition, you should prevent outbound connections unless there is a strong reason to permit it.

The next thing you will want to do is enable authentication and access control.

Enabling authentication and access control for the database

Only the minimum amount of people possible should have access to the database, and they should have the minimum amount of access needed to do their job. As Keycloak manages the schema as well as the data in the database, ask yourself if anyone really needs permanent access to the database at all.

Keycloak, as well as any users accessing the database, should use strong passwords, and accounts should be locked after failed login attempts. Consider using stronger authentication mechanisms, such as client certificates.

After limiting access to the database, you will want to secure the data in transit as well as at rest by enabling encryption.

Encrypting the database

To protect data in transit, all connections to the database should be encrypted by leveraging TLS.

In the event someone gains access to the server where the database is running, it is also important to encrypt data at rest. This includes making sure you encrypt any backups of the database.

There are a lot more steps involved in properly securing a database. If your company has its own data center, chances are you already have people that can help you with this task. If not, you could consider leveraging a relational database service in the cloud.

In the next section, we will look at how to secure communication between nodes in a cluster.

Securing cluster communication

Keycloak embeds Infinispan, which is leveraged when you create a cluster of Keycloak nodes. More sensitive data such as signing keys or user information is not sent across the cluster, as this information is only kept in a local cache in each node with the only

communication across the cluster being invalidation messages. It does store information about user sessions in the cluster, which are distributed across the cluster. Sessions themselves contain some information such as the session ID, the expiration date, and associated client sessions. Even if an attacker gains access to this information, they are limited in terms of what they can do with it, since accessing any session through Keycloak requires a token or cookie to be signed by Keycloak.

It would still be a good idea to secure cluster communication, at the very least with a firewall. For additional protection, you can enable authentication and/or encryption for cluster communication.

At the time of writing this book, the Keycloak documentation does not provide instructions on how to secure cluster communication. To find out more on how to secure cluster communication, refer to the Infinispan documentation at <https://infinispan.org/docs/dev/titles/security/security.html#secure-cluster-transport>.

Enabling cluster authentication

Enabling authentication prevents unauthorized nodes from joining the cluster, but it does not prevent non-members from communicating with cluster members.

For this reason, there is little value in adding authentication on its own, and this should be combined with asymmetric encryption.

Encrypting cluster communication

Cluster communication can be encrypted either with symmetric encryption with a shared key or asymmetric encryption. The simplest approach is enabling symmetric encryption, so we will look at how you can enable that.

The first step is to create a Java keystore that holds the shared secret. To create the keystore, run the following command in a Terminal:

```
$ cd $KC_HOME
$ keytool -gensecretkey -alias myKey \
    -keyalg aes \
    -keysize 128 \
    -keystore myKeystore.p12 \
    -storetype PKCS12 \
    -storepass changeit \
    -keypass changeit
```

This command will create a keystore in the root of your Keycloak home directory, with a key that can be used for symmetric encryption. You should copy this file to all Keycloak nodes.

The next step is to open the `conf/cache-ispn.xml` file in a text editor. Insert the following before the `<cache-container ..>` element:

```
<jgroups>
  <stack name="encrypt-udp" extends="udp">
```

```
<SYM_ENCRYPT keystore_name="myKeystore.p12"  
keystore_type="PKCS12"  
store_password="changeit"  
key_password="changeit"  
alias="myKey"  
stack.combine="INSERT_AFTER"  
stack.position="VERIFY_SUSPECT2"/>  
</stack>  
</jgroups>
```

This will add a new cache stack that extends the default **udp** stack to add symmetric encryption leveraging the keystore you created previously. Next, you need to configure the cache-container to use this stack by updating the `<transport ..>` element to:

```
<transport lock-timeout="60000" stack="encrypt-udp",
```

You should also make sure the changes you made to the `conf/cache-ispn.xml` file are made to all Keycloak nodes, as well as copy the `myKeystore.p12` file to all nodes.

At the time this was written, there is an issue with Keycloak not automatically picking up changes to the configuration file, so you need to run `kc.sh build` after updating the file.

In addition, you have to also specify to use this configuration file with the `cache-configuration-`

`file` option.

If you want to make sure encryption is enabled, you can try to start one Keycloak node with encryption enabled and one without, or you can try creating a different keystore on one node. The following example shows the output from Keycloak when one node is trying to join the cluster with a different keystore:

```
ERROR [org.jgroups.protocols.SYM_ENCRYPT] (jgroups-{
```

This message shows that the node was not permitted to join the cluster and will also not be able to read or send any messages to the cluster.

With that, you have learned how to secure cluster communication. In the next section, we will look at securing user accounts.

Securing user accounts

With regard to securing user accounts, you will want to protect against an attacker gaining access to the user account and also protect information about the user, including their password.

Preventing an attacker from accessing a user account is mostly about enabling strong authentication, and not just accepting a password as the means of authentication. If your users are relying on passwords, even in combination with a second factor, it is important that passwords are protected.

Passwords are protected by leveraging a strong password hashing algorithm, having a good password policy, and enabling **brute-force** protection for passwords. It is also important to educate users in terms of what is a strong password and that they should not reuse passwords with other services.

To configure a password policy, open the Keycloak administration console and select the realm you want to configure. Then, click on **Authentication**, then **Policies**, and select the **Password policy** tab. You can create your password policy by clicking on **Add policy** and selecting the policies you want to use. The following screenshot shows an example policy that requires passwords to have a minimum length of 8 and contain at least one lowercase letter, one uppercase letter, one special character, and one digit:

Flows Required actions Policies

Password policy OTP Policy Webauthn Policy Webauthn Passwordless Policy

Add policy ▾

Minimum Length * ⓘ - 8 +

Special Characters * ⓘ - 1 +

Uppercase Characters * ⓘ - 1 +

Lowercase Characters * ⓘ - 1 +

Digits * ⓘ - 1 +

The screenshot shows a user interface for creating a password policy. At the top, there are tabs for 'Flows', 'Required actions', and 'Policies', with 'Policies' being the active tab. Below the tabs, there are four policy types: 'Password policy', 'OTP Policy', 'Webauthn Policy', and 'Webauthn Passwordless Policy', with 'Password policy' selected. A large button labeled 'Add policy' with a dropdown arrow is below the tabs. The main area contains five sections, each with a requirement label (e.g., 'Minimum Length'), a red asterisk indicating it's required, a question mark icon for help, and a numeric input field with a minus sign on the left and a plus sign on the right. The current values are: Minimum Length is 8, Special Characters is 1, Uppercase Characters is 1, Lowercase Characters is 1, and Digits is 1.

Figure 14.5: An example password policy

It is also a good idea to enable password **brute-force** detection. Do this by clicking on **Realm Settings**, then **Security defenses**, and selecting the **Brute force detection** tab, as shown in the following screenshot:

General Login Email Themes Keys Events Localization Security defenses S

Headers Brute force detection

Enabled On

Max login failures 30

Permanent lockout Off

Wait increment 1 Minutes ▾

Max wait 15 Minutes ▾

Failure reset time 12 Hours ▾

Quick login check milliseconds 1000

Minimum quick login wait 1 Minutes ▾

Save Revert

Figure 14.6: Enabling password brute-force detection

Depending on your use case, you may be storing different levels of personal data, or personally identifiable information, about your users. There are a few steps you can take to limit your exposure to issues in this regard:

- Limit the information you store about users to only what is absolutely required.
- Limit what user information is exposed to applications.
- Secure the database.
- Understand legislation around personal information in regions where your business operates.

This topic should not be taken lightly. Personal information is invaluable to an attacker and is a commodity on its own that can be sold. Leaking such information can result in large fines and, in the worst cases, cause irreparable damage to your business.

In the last section of this chapter, we will look at the steps you should take to increase the security of applications.

Securing applications

Since more applications are being exposed on the internet, the number of attacks and data breaches is growing by the day. This means it is important to secure applications properly.

Up until recently, a common practice was to leverage firewalls and VPNs as the main layer of defense against attacks. Often, this was combined with questionable security within the boundaries of the enterprise environment. This is becoming less viable with more employees working from home or using their personal laptops or phones. More and more services are also being exposed to partners or the public. This is blurring the line of the enterprise network. The whole idea of trusting what is on the inside, but not what is on the outside, was also somewhat questionable as there are often

ways for attackers to get inside the enterprise network, and it also provides less protection against an internal attack.

Essentially, something better is needed than just a firewall.

Keycloak is a great tool that can help increase the security of your applications, but your applications are not secure simply by using Keycloak.

It is beyond the scope of this book to provide you with all the information you need to secure your applications. Reading this section will give you some idea of this. We will start by looking at web application security.

Web application security

There are plenty of books and good resources on the internet that can help you learn how to secure web applications. Some of the steps involved in securing web applications include the following:

- **Authentication:** Since you are reading a book about Keycloak, chances are you are planning to use Keycloak to authenticate users to your applications. Once a user has been authenticated and a session has been established, it is important that the session is also secure.
- **Authorization:** Least privilege access is a great principle to follow. If you limit the access that's granted to users for them to perform their job, you are reducing the impact of a compromised account or a rogue employee.
- **Understand and protect against common attacks:** Make sure your applications are protected against common vulnerabilities

such as injection and **cross-site scripting (XSS)**.

- **Regular updates:** Web application security is a continuous effort, and you should continuously strive to improve the security of your application. You should also regularly update frameworks, libraries, and any tools you are leveraging.
- **Data security:** Sensitive data should be encrypted at rest, and all data should be encrypted in transit. This should also apply to any backup data. Just like the web application itself, it is important to have good authentication and authorization in place.
- **Logging and monitoring:** Without sufficient logging and monitoring, you will not be able to identify if you have been compromised. Logging and monitoring can also be a valuable tool to prevent larger impacts due to an ongoing attack.
- **Firewall:** Firewalls and WAFs add an extra layer of defense to your web applications. Relying on only a WAF for protection is far from ideal, though – you should build security into the application itself.

One of the best places to start learning more about web application security is the **Open Web Application Security Project (OWASP) Top 10**. The OWASP Top 10 is a list of some of the most critical security risks to web applications. For each risk, it provides easy-to-understand details of the vulnerabilities, as well as tips on how to protect your application.

Another great resource is the OWASP Cheat Sheet Series, which provides several cheat sheets with very concise information on specific areas of application security.

Next, we will look at how to securely leverage OAuth 2.0 and OpenID Connect in your application.

OAuth 2.0 and OpenID Connect best practice

There are a lot of mistakes that can be made when you're using OAuth 2.0 and OpenID Connect in your applications. The specifications themselves have a lot of flexibility in how they are used, and a lot of the mechanisms to protect against common vulnerabilities are optional.

Consider the following authorization request, for example:



In this request, neither the state nor the nonce parameter is included. A **Proof Key for Code Exchange (PKCE)** is also not being used. Unless the authorization server explicitly requires these parameters, this is a perfectly valid authorization request, but at the same time, it is open to several known vulnerabilities.

The same problem applies to the **JSON Web Token (JWT)** specification. It is relatively easy to make mistakes here. One example is the *none* algorithm, which is included in the specification. A valid token, according to the specification, can simply specify that it uses no signing algorithm, which obviously makes it easy for an attacker to create their own tokens.

Chapter 6, Securing Different Application Types, covered a fair portion of what you need to know to use OAuth 2.0 securely, but it is recommended that you learn more about this topic through other resources. The OAuth 2.0 website (<https://oauth.net/2/>) contains links to several invaluable resources that are all worth reading, including the following:

- OAuth 2.0 for mobile and native apps
- OAuth 2.0 for browser-based apps
- OAuth 2.0 threat model and security considerations
- OAuth 2.0 security best current practice
- The OAuth 2.1 Authorization Framework

With all the options in OAuth 2.0 and the potential for not following the best practices, this may be a bit confusing. Luckily, some improvements in this regard are on the way with the introduction of OAuth 2.1. OAuth 2.1 incorporates several best practices into the specification itself, making it easier to follow best practices by simply being compliant with the specification.

Other important work regarding security is happening in the **Financial-Grade API (FAPI)** working group. This working group derives from establishing highly secure profiles of OIDC in order to leverage OIDC for open banking. However, you should not get too hung up on the name as the work they have produced applies to any use case for OIDC where additional security is required. The most important work that is coming out of this is two profiles for OIDC that are providing it with best practices:

- **FAPI 1.0 – Part 1:** Baseline API Security Profile
- **FAPI 1.0 – Part 2:** Advanced Security Profile

These profiles allow you to balance the complexity of applying the best practice with the level of security required so that it fits the use cases you have.

The Keycloak team is also making great progress in making it easier for you to enforce secure usage of OAuth 2.0 and OpenID Connect in your applications; a key part of this is the introduction of the client policies feature. Through client policies, it is easy for you to enforce the use of best practices for applications, including allowing you to select different profiles for different applications, depending on the level of security needed.

We will conclude this chapter by looking at various configuration options available in Keycloak that affect the security of your applications.

Keycloak client configurations

In this section, we will look at some configuration options for an OIDC client in Keycloak that affect security.

The following screenshot shows the basic settings for an OIDC client:

Access settings

Root URL [?](#)

Home URL [?](#)

Valid redirect URIs [?](#) [-](#)
[+ Add valid redirect URIs](#)

Valid post logout redirect URIs [?](#) [-](#)
[+ Add valid post logout redirect URIs](#)

Web origins [?](#) [-](#)
[+ Add web origins](#)

Admin URL [?](#)

Capability config

Client authentication [?](#) Off

Authorization [?](#) Off

Authentication flow Standard flow [?](#) Direct access grants [?](#)
 Implicit flow [?](#) Service accounts roles [?](#)
 OAuth 2.0 Device Authorization Grant [?](#)

Figure 14.7: Client settings

Let's review these and consider which are more related to security:

- **Consent required:** If this is not enabled, the user will not see what level of access the application receives. You should enable this option for any third-party application. You should also enable this option for a native application such as a CLI.
- **Client authentication:** Enabling this is more secure when the client credentials can be kept securely on the server side.
- **Standard flow:** Enables the authorization code flow for the client.
- **Implicit flow:** Enables the now deprecated implicit flow. You should only enable this when it's absolutely required, and plan to update the application so that this can be disabled.
- **Direct access grants:** Enables the now deprecated resource owner password flow. You should only enable this when it's absolutely required, and plan to update the application so this can be disabled.
- **Valid redirect URIs:** It is recommended to use an exact match for the redirect URI. An example of a good redirect URI would be <https://acme.corp/myclient/oauth-callback>. Keycloak does support a wildcard in the redirect URI – any redirect URI, for that matter. You should avoid using a wildcard in the redirect URIs, but if you do, limit it to only requests that are available to the application, such as https://acme.corp/myclient/*.

Next, we will look at different signing algorithms that Keycloak supports:

- **Rivest–Shamir–Adleman (RSA) signature:** This is the default algorithm used by Keycloak. It is not the most secure option,

but it is used as the default as it is the most widely available.

- **Elliptic Curve Digital Signature Algorithm (ECDSA):** Considered more secure than RSA and is also significantly faster.
- **Hash-based message authentication code (HMAC):** A symmetric signing algorithm that requires access to a shared secret.

When possible, you should use ECDSA in favor of RSA, even though RSA is still considered secure. If you want to enforce applications to use the token introspection endpoint to verify the token, you can use HMAC as the required secret. This is only available to Keycloak.

You can also choose between different lengths of the signature hash, where longer hashes provide greater security. With relatively short-lived tokens such as refresh tokens and access tokens, a 256-bit length is considered more than secure enough for most use cases.

Some other important options include configuring the lifespan of tokens. Keycloak allows you to override the access token lifespan for individual clients. It also allows you to override the client session lifespan, which controls the lifespan of refresh tokens. This allows you to have a long-lived SSO session (which could be days or weeks) with shorter-lived refresh tokens (which could be less than an hour). Shorter-lived refresh tokens reduce the impact if any refresh tokens are leaked and shorten the application HTTP session's lifespan.

Summary

In this chapter, you learned about several important aspects of deploying Keycloak securely into production. You learned how important it is to secure the database that's used by Keycloak, as well as communication between nodes. You also learned how important it is to protect user accounts from being compromised, as well as how important it is to keep information about your users secure. Finally, you got some insight into what it means to secure an application by focusing on web applications, as well as how to utilize OAuth 2.0 and OpenID Connect to secure your applications.

You should now have a good understanding of how to securely run Keycloak in production, as well as an idea of where you can start learning more about securing your applications.

This is the final chapter of this book. We hope you have enjoyed this book and have gained a good understanding of Keycloak and how you can utilize it to help secure your applications. While this book has not covered everything you may need to know, you should now have the knowledge to get started on your journey with Keycloak. As a next step, you may want to learn more about OAuth 2.0 and OpenID Connect, or web application security in general.

Questions

1. Why is it important to regularly update Keycloak?
2. Why is it especially important to protect the database that's used by Keycloak?
3. Is it sufficient to use a WAF to protect web applications?

Further reading

Please refer to the following links for more information on the topics that were covered in this chapter:

- OWASP Top 10: <https://owasp.org/www-project-top-ten/>
- OWASP Cheat Sheet Series:
<https://cheatsheetseries.owasp.org/index.html>
- OAuth 2.0 for Mobile and Native Apps:
<https://tools.ietf.org/html/rfc8252>
- OAuth 2.0 for Browser-Based Apps:
<https://tools.ietf.org/html/draft-ietf-oauth-browser-based-apps>
- OAuth 2.0 Threat Model and Security Considerations:
<https://tools.ietf.org/html/rfc6819>
- OAuth 2.0 Security Best Current Practice:
<https://tools.ietf.org/html/draft-ietf-oauth-security-topics>
- OAuth 2.1: <https://tools.ietf.org/html/draft-parecki-oauth-v2-1>

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/SecNet>



Assessments

Chapter 1

1. Yes. Keycloak distributes container images for Docker, which runs on Kubernetes. There is also a Kubernetes Operator for Keycloak that makes it easier to install and manage Keycloak on Kubernetes.
2. The Keycloak admin console provides an extensive console to allow you to configure and manage Keycloak, including managing applications and users.
3. The Keycloak account console provides a self-service console for end users of your applications to manage their own accounts, including updating their profile and changing their password.

Chapter 2

1. The application redirects the user to the login pages provided by Keycloak. Following authentication, the user is redirected back to the application and the application obtains an ID token from Keycloak that it can use to discover information about the authenticated user.
2. For an application to be permitted to authenticate users with Keycloak, it must first be registered as a client with Keycloak.
3. The application includes an access token in the request, which the backend service can verify to decide whether access should be granted.

Chapter 3

1. OAuth 2.0 enables an application to obtain an access token that grants access to a set of resources provided by a different application on behalf of the user.
2. OpenID Connect adds an authentication layer on top of OAuth 2.0.
3. OAuth 2.0 does not define a standard format for tokens. By leveraging JWT as the token format, applications are able to directly verify and understand the contents of the token.

Chapter 4

1. Through the Discovery endpoint, an application can find out a lot of useful information about an OpenID Provider, which allows it to automatically configure itself to a specific provider.
2. The application retrieves an ID token, a signed JWT, from the OpenID Provider, which contains information about the authenticated user.
3. By adding a protocol mapper, or a client scope, to a client you can control exactly what information is included in the ID token that is made available to an application.

Chapter 5

1. An application can leverage the OAuth 2.0 Authorization Code grant type to obtain an access token from the authorization server. The application then includes the access token in the request sent to the REST API.
2. An access token can be limited through the use of the audience, roles, or scopes.
3. A service can either invoke the token introspection endpoint to verify the access token, or if the token is a JWT, it can verify and read the contents of the token directly.

Chapter 6

1. As an SPA is running in the browser, it cannot use a confidential client directly, which results in a greater risk if a refresh token is leaked. For this reason, it is more secure to have a backend running in a web server that can use a confidential client and store tokens on the server side.
2. No, any type of application can use OAuth 2.0 through an external user agent to obtain an access token, and many different types of services have support for bearer tokens.
3. An application should never collect user credentials directly as this increases the chance of credentials being leaked and provides the application with full access to the user account. For this reason, native and mobile applications should use an external user agent to authenticate with Keycloak.

Chapter 7

1. In this chapter, you were presented with different integration options for different programming languages and platforms. If the programming language you are using already supports OpenID Connect, even if this is being done through a library or framework, you should consider using it. Alternatively, you can also use a reverse proxy such as Apache HTTP Server.
2. No, the Keycloak adapters were created when there were not many trusted client libraries. Nowadays, programming languages, and the frameworks built on top of these languages, already provide support for OpenID Connect. As a rule of thumb, do the opposite: only consider using any of the Keycloak adapters if you are left with no other option.
3. If you are using Reactive Native, you might want to look at <https://github.com/FormidableLabs/react-native-app-auth/blob/main/docs/config-examples/keycloak.md>. There you should find examples on how to use it with Keycloak. Remember that Keycloak is a fully compliant OpenID Connect Provider implementation, and you should be able to use any other library too.
4. For applications running on Kubernetes or OpenShift, both integration architecture styles would fit. Depending on the service mesh you are using (for instance, Istio), you should be able to leverage its capabilities. But still, you can use the embedded architectural style. This makes a lot of sense if you

are already familiar with the options you have from the technology stack your application is using.

Chapter 8

1. When you put data into tokens, they actually grow disproportionately in size. One option to help here is to include only the minimum information that your application needs and, for additional information, to use the token introspection endpoint. The drawback is that your application will need an additional request to Keycloak when serving requests. You should also consider disabling the **Full Scope Allowed** setting in your client settings, so that only information relevant to your client is included in tokens.
2. Realm roles should be used to represent the user's role within an organization. These roles have the same semantics regardless of the clients created in a realm. On the other hand, the semantics for a client role are specific to the client they belong to. In this chapter, we created a realm role and a client role using the same name: **manager**. While the realm role could represent users with the role of manager in an organization, the manager client role could represent the permissions to manage resources in the application. It is up to you to choose what best fits your case; just keep in mind this conceptual difference so that you do not overuse one or the other.
3. Yes; for that, you would need to customize Keycloak through the Authentication SPI. It should be possible to have, for instance, additional pages in your flows to gather user information, or just use a custom authenticator to push contextual information as a session note, so that later you can

map information from these session notes to tokens using a protocol mapper.

4. Yes, and this is a common task when your application is expecting to obtain roles from a claim other than the **realm_access** or **resource_access** claims. You can always change the protocol mappers to better suit your application's needs.
5. No. Applications can use different strategies depending on their security requirements. It is perfectly fine to use RBAC or groups, for instance, at the same application, or even to use ABAC or Keycloak Authorization Services, if you need fine-grained access to protected resources.

Chapter 9

1. Yes. It is recommended that you have an active-passive or active-active database so that in the event of failures, you can easily switch database instances. Note, however, that Keycloak keeps as much data as possible in caches, where reads should not be impacted at all depending on how hot the caches are (how much data is cached). Writes, however, will fail until the connection is re-established. Keycloak also supports setting some useful configuration options to improve failover in the event of network failures. You might want to enable background validation of connections to make sure available connections are usable, validate connections prior to obtaining them from the connection pool, or even configure the pool to fail fast when a connection is terminated to avoid validating and iterating over all connections in the pool.
2. No. The default configuration uses IP multicast to broadcast messages across nodes and form a cluster. The proper configuration depends on where Keycloak is being deployed. If you are deploying on bare metal or in a VM, you should consider using a different JGroups stack using either **TCPPING** or **JDBC_PING** for discovery.
3. Keycloak provides an operator that takes care of setting up most of the things we discussed in this chapter. We highly recommend using it to run Keycloak on any of these platforms. In terms of clustering, when running on OpenShift and Kubernetes **DNS_PING** is recommended for discovery.

4. By default, there is no security in this communication so that any instance listening on the same multicast address can join the cluster. To prevent unexpected nodes from joining a cluster, you can configure your JGroups stack to rely on X.509 certificates to authenticate nodes. You should also be able to enable encryption to prevent data from being intercepted and transferred in cleartext. For more details, look at the recommendations from *Chapter 14, Securing Keycloak and Applications*.
5. Ideally, yes. The reason being that, even though your instances are running within a private network, Keycloak is constantly exchanging sensitive data about users (privacy) and tokens issued by the server to applications. It is recommended to use end-to-end encryption. Yes. Keycloak is CPU intensive due to password hashing, token issuance, and verification using signatures and encryption. Depending on your load and how many concurrent requests you need to handle, you might want to allocate two or more CPUs for each node in the cluster. High CPU usage can also be caused by frequent GC runs. Common causes might be related to a small metaspace size, small young generation size, or the JVM reaching the overall heap size. You should constantly be monitoring and adjusting GC runs until you get as few pauses and counts as possible.
6. Keep in mind that TLS demands CPU. Depending on your requirements you might want to configure your reverse proxy with TLS termination and save some CPU usage on Keycloak nodes.

7. It depends on the use case. You should start small, using the default settings, and adjust accordingly to your load and performance tests.
8. Yes. Check the Keycloak Benchmark tool at <https://github.com/keycloak/keycloak-benchmark>.

Chapter 10

1. Yes. As we will see in the following chapters, Keycloak provides a **Service Provider Interface (SPI)** that allows you to integrate not only with databases but with any other form of identity store.
2. No. In addition to storing information from LDAP in its own database, Keycloak also caches data for entries that have been imported from LDAP. You have complete control over how information is cached and when it expires. Here, together with the synchronization settings, information from the LDAP directory is periodically updated without it impacting the overall performance of the server.
3. Keycloak allows you to configure mappers for identity providers. Through these mappers, you can customize how users are created by setting a specific user attribute or setting a specific role when the user authenticates for the very first time.

Chapter 11

1. Keycloak allows you to customize its look and feel entirely, not just for the pages that were presented in this chapter. As we are going to see in *Chapter 13, Extending Keycloak*, you should be able to change the look and feel of pages by changing the different themes provided by Keycloak. You can find more details in the documentation at https://www.keycloak.org/docs/latest/server_development/#_themes.
2. WebAuthn requires you to use a FIDO or FIDO2 compliant security device. You should also consider accessing Keycloak using HTTPS and using a valid domain name. WebAuthn is strict about domain names and secure connections if the server is accessed from a different domain than the client. You should also make sure the browser you are using has support for the WebAuthn API. You should also consider looking at the demo on the WebAuthn site to check how your security device works there.

Chapter 12

1. Most of the time, sessions – both user and client – are not stored in the database. As you learnt from *Chapter 9, Configuring Keycloak for Production*, sessions are stored in-memory and shared across the different cluster nodes. However, there is a specific type of session called an **offline session** that is stored in the database. For more details about offline sessions, look at the documentation at https://www.keycloak.org/docs/latest/server_admin/#_offline-access.
2. User sessions hold state about the authenticated user, regardless of the client. On the other hand, client sessions are bound to the client the user authenticated with and they hold the state about the user within the context of a specific client.
3. Keycloak provides different ways for revoking tokens and expiring sessions. As you learnt from this chapter, tokens can be revoked by invoking the revocation endpoint and the sessions can be destroyed through the administration console. Note that when a session is terminated, tokens associated with that session are no longer valid.
4. Applications can validate tokens locally or by invoking specific endpoints at Keycloak. By validating tokens locally, applications are saving a server roundtrip with the cost of not checking if the session to which a token is bound is still valid on the server.

5. The access token lifetime impacts clients from a security, usability, and performance perspective. From a security perspective, you should prefer using a short lifetime to reduce the attack window in case a malicious actor has access to an access token. On the other hand, a very short lifetime might force the client to execute refresh token requests more frequently, therefore, introducing more complexity on clients and more server roundtrips.

Chapter 13

1. In Keycloak's code base, SPIs are organized into two main modules, the `keycloak-server-spi` and `keycloak-server-spi-private` modules. Public SPIs are located in the `keycloak-server-spi` module and Keycloak does its best to keep their interfaces backward compatible between releases. These SPIs are also usually documented. On the other hand, private SPIs are not supported in terms of backward compatibility and they usually lack documentation. As a rule of thumb, consider first looking at the SPIs available from the documentation, as they are usually what Keycloak expects people to use for extending the server.
2. In order to deploy your extensions, you must deploy your extension's JAR file to the `$KC_HOME/providers` directory and restart the server. If you are using an optimized server image, you should re-run the `build` command. For more details about running optimized images, look at the documentation at https://www.keycloak.org/server/configuration#_optimize_the_keycloak_startup.
3. Yes. As you learnt from this chapter, Keycloak is very flexible about changing its look and feel and most of its UIs provide hooks for customization.
4. Yes. As you learnt from this chapter, Keycloak provides a specific Java SPI for this purpose. The example provided in this chapter is a very simple example of what you can achieve.

5. This depends on the SPI you need to extend. As you learned from this chapter, in order to extend Keycloak themes, a basic CSS, JavaScript, and HTML background is more than enough. However, for other forms of customization, you will require a basic background in Java.

Chapter 14

1. There is no such thing as perfectly secure software, and mistakes are frequently made. Luckily, both the Keycloak team and its community are continuously looking for vulnerabilities and are continuously fixing any issues they find. If you don't update Keycloak, you will not receive these fixes, but anyone wanting to attack your Keycloak server will.
2. Keycloak stores a lot of sensitive data in the database, which is valuable information to an attacker. If an attacker gains write access to the database, the attacker can make changes that could allow the attacker to gain access to any application secured by Keycloak.
3. No; only relying on a web application firewall is not a good idea. You will want to enable strong authentication, as well as provide a good level of security within the application itself.



packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Cybersecurity Threats, Malware Trends, and Strategies - Second Edition

Tim Rains

ISBN: 9781804613672

- Discover enterprise cybersecurity strategies and the ingredients critical to their success
- Improve vulnerability management by reducing risks and costs for your organization

- Mitigate internet-based threats such as drive-by download attacks and malware distribution sites
- Learn the roles that governments play in cybersecurity and how to mitigate government access to data
- Weigh the pros and cons of popular cybersecurity strategies such as Zero Trust, the Intrusion Kill Chain, and others
- Implement and then measure the outcome of a cybersecurity strategy
- Discover how the cloud can provide better security and compliance capabilities than on-premises IT environments



Cybersecurity Blue Team Strategies

Kunal Sehgal, Nikolaos Thymianis

ISBN: 9781801072472

- Understand blue team operations and its role in safeguarding businesses
- Explore everyday blue team functions and tools used by them
- Become acquainted with risk assessment and management from a blue team perspective
- Discover the making of effective defense strategies and their operations
- Find out what makes a good governance program
- Become familiar with preventive and detective controls for minimizing risk

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Keycloak - Identity and Access Management for Modern Applications, Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here to go straight to the Amazon review page](#) for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A

access tokens

- access, limiting [69](#)
- access, limiting with audience [70](#), [71](#)
- lifetimes, managing [243-245](#)
- limiting, with roles [71-74](#)
- limiting, with scope [75-77](#)
- obtaining [62-65](#)
- validating [77-80](#)

Active Directory [3](#)

- integrating with [183-186](#)

active sessions

- managing [237-239](#)

admin URL

- setting [156](#), [157](#)

Apache Freemarker [270](#)

application

- access, authorizing with OAuth 2.0 [25-29](#)
- user consent, requiring [65-69](#)

applications

securing [295](#)

attribute-based access control (ABAC) [182](#)

using [145](#)

authentication and access control

enabling, for database [289](#), [290](#)

authentication flows [202](#), [203](#)

configuring [204-209](#)

customizing [272-276](#)

WebAuthn, enabling [225-227](#)

authentication request [30](#)

authorization code [15](#), [44](#)

authorization code flow [15](#), [26](#)

Authorization Code Injection [113](#)

authorization request [27](#), [30](#)

authorization system [134](#), [135](#)

B

backend for frontends (BFF) patterns [91](#)

backend REST API [13](#)

invoking, securely [23](#)

backend URL

setting [155](#), [156](#)

base theme [263](#)

browser-based applications [29](#)

browser flow [203](#), [227](#)

brute force detection feature

reference link [276](#)

brute-force protection [293](#)

C

cache-stack option [162](#)

centralized authorization server

Keycloak, using as [146](#), [147](#)

Certificate Authority (CA) [157](#)

claimed HTTPS scheme [97](#)

Client authentication flow [204](#)

client configurations [298-300](#)

Client Credentials flow [26](#)

client roles [135](#)

client, with Keycloak

settings [98](#)

cluster authentication

enabling [290](#)

cluster communication

encrypting [291](#), [292](#)

securing [290](#)

clustering

enabling [161-163](#)

Common Name (CN) attribute [188](#)

communication

encrypting, to Keycloak [282](#)
composite roles [136](#)
confidential clients [28](#)
cookies [241](#)
 for session tracking [242](#)
Credentials tab [112](#)
Cross-Origin Resource Sharing (CORS) [20](#), [91](#)
Cross-Site Request Forgery (CSRF) [113](#)
cross-site scripting (XSS) [93](#), [241](#), [296](#)
custom URI scheme [97](#)

D

database

authentication and access control, enabling for [289](#), [290](#)
configuring [159](#), [160](#)
encrypting [290](#)
protecting, with firewall [289](#)
securing [288](#), [289](#)

dedicated REST API

used, for securing SPA [90](#), [91](#)

Device flow [26](#)

Direct Access Grants option [108](#)

Direct grant flow [204](#)

Discovery endpoint [40](#), [41](#)

Distinguished Name (DN) [187](#)

Docker

Keycloak, running on [4](#), [5](#)

E

education [34](#)

Elliptic Curve Digital Signature Algorithm (ECDSA) [299](#)

environment

testing [168](#)

environment test

backchannel URL, testing [170](#)

frontend URL, testing [170](#)

load balancing and failover, testing [169](#)

Event Listener SPI [277](#)

external application [84](#), [85](#)

external REST API

used, for securing SPA [93](#), [94](#)

external vault

secrets, loading from [287](#)

F

federated user [185](#)

Financial-Grade API (FAPI) [32](#), [297](#)

firewall

database, protecting with [289](#)

Keycloak, protecting with [287](#), [288](#)

FreeOTP [219](#)

frontend URL

setting [153-155](#)

frontend web application [13](#)

Fully Qualified Name (FQN) [188](#)

G

Golang applications

integrating with [111-113](#)

Google Authenticator [219](#)

government [34](#)

group-based access control (GBAC)

group membership, mapping into tokens [137-143](#)

using [136](#), [137](#)

group membership

mapping, into tokens [137-143](#)

H

Hardware Security Module (HSM) [286](#)

Hash-based message authentication code (HMAC) [299](#)

hashing algorithm [210](#)

HMAC-Based One-Time Password (HOTP) [218](#)

HMAC-SHA-256 [210](#)

HMAC-SHA-512 [210](#)

hostname, for Keycloak

admin URL, setting [156](#), [157](#)

backend URL, setting [155](#), [156](#)

frontend URL, setting [153-155](#)

setting [153](#)

HTTP [241](#)

HTTPS [157](#)

I

ID and access token expiration

leveraging [55](#)

Identity and Access Management (IAM) [2](#), [253](#)

ID tokens [46-49](#)

custom property, adding [50](#), [51](#)

lifetimes, managing [243-245](#)

roles, adding to [51](#)

user profile, updating [49](#)

input-constrained device

authenticating with [99](#), [100](#)

Integrated Development Environments (IDEs) [113](#)

integration

implementing [129](#), [130](#)

integration architecture

option, selecting [110](#), [111](#)

selecting [109](#), [110](#)

intermediary REST API

used, for securing SPA [91](#), [93](#)

internal application [84](#), [85](#)

intrusion prevention system

Keycloak, protecting with [287](#), [288](#)

iterations [210](#)

J

Java applications

integrating with [113](#)

Java Archive (JAR) [258](#)

Java Runtime Environment (JRE) [5](#)

JavaScript [14](#)

JavaScript applications

integrating with [121](#)-[123](#)

JavaScript Object Signing and Encryption (JOSE) [33](#)

JSON Web Key Set (JWKS) [33](#)

JSON Web Signature (JWS) [17](#)

JSON Web Token (JWT) [46](#), [63](#), [117](#), [242](#)

leveraging, for tokens [33](#), [34](#)

specification [33](#), [297](#)

K

Keycloak [2](#), [3](#), [254](#), [255](#)

communication, encrypting to [282](#)

customization points [276](#), [277](#)

download link [5](#)
hostname, configuring [283](#)
installing [3-5](#)
installing, with OpenJDK [5](#)
look and feel, modifying [262](#)
protecting, with firewall [287](#), [288](#)
protecting, with intrusion prevention system [287](#), [288](#)
running [3](#), [4](#)
running, on Docker [4](#), [5](#)
running, with OpenJDK [5](#)
securing [282](#)
signing keys, rotating [283-286](#)
templates, extending [270](#)
theme, creating [266-270](#)
theme, deploying [266-270](#)
theme-related SPIs, extending [270](#), [271](#)
themes [262-266](#)
updating, regularly [286](#)
used, for securing mobile applications [95-98](#)
using, as centralized authorization server [146](#), [147](#)
working with [6](#)

Keycloak account console

discovering [6](#)
used, for allowing users to manager their data [196-198](#)

working with [11](#)

Keycloak admin console

discovering [6](#)

global role, creating [10](#)

group, creating [10](#)

realm, configuring [7](#), [8](#)

realm, creating [7](#), [8](#)

user, creating [8](#), [9](#)

working with [7](#)

Keycloak Authorization Services

reference link [147](#)

Keycloak, header

proxies [166](#)

KEYCLOAK_IDENTITY cookie [241](#)

Keycloak Quickstart

reference link [276](#)

KeycloakSession component [259](#)

KeycloakSessionFactory component [259](#)

keys

active keys [284](#)

disabled keys [284](#)

passive keys [284](#)

Kubernetes Secrets, with Keycloak

reference link [287](#)

L

Lightweight Directory Access Protocol (LDAP)

group data, synchronizing [187-189](#)

integrating with [183-186](#)

mappers [186](#), [187](#)

roles, synchronizing [189](#), [190](#)

servers [3](#)

local users [185](#)

creating [174-176](#)

credentials, managing [176-178](#)

managing [174](#)

self-registration, enabling [181](#)

user attributes, managing [182](#), [183](#)

user information, obtaining and validating [178-180](#)

logout events, handling

reference link [241](#)

loopback interface [97](#)

M

mobile applications

securing, with Keycloak [95-98](#)

Multi-Factor Authentication (MFA) [2](#)

N

native applications [29](#)

Near-Field Communication (NFC) [225](#)

Node.js applications

client, creating [124](#), [125](#)

integrating with [123](#), [124](#)

resource server, creating [125-128](#)

non-opaque tokens [17](#)

O

OAuth 2.0

best practice [296](#), [297](#)

playground, running [60](#), [61](#)

roles [26](#)

URL [297](#)

used, for authorizing application access [25-29](#)

OAuth2 scopes

using [144](#)

OIDC Back-Channel Logout

leveraging [56](#)

OIDC Front-Channel Logout

note [56](#)

OIDC Session Management

leveraging [55](#), [56](#)

One-Time Password (OTP) [205](#)

policies, modifying [218](#), [219](#)

used, for forcing users to authenticate [222-224](#)

users, allowing to select whether they want to use [220-222](#) using [217](#)

OpenID Connect [29](#)

best practice [296](#), [297](#)

used, for authenticating users [29-32](#)

OpenID Connect Core specification [29](#)

OpenID Connect identity provider

creating [191-194](#)

OpenID Connect (OIDC) playground application

running [38](#), [39](#)

user, authenticating [42-46](#)

OpenID Connect Provider (OP) [192](#)

OpenID Provider Metadata [40](#)

OpenID Provider (OP) [30](#)

OpenJDK

installing [5](#)

used, for installing Keycloak [5](#)

used, for running Keycloak [5](#)

Open Web Application Security Project (OWASP) [93](#)

Open Web Application Security Project (OWASP) Top 10 [296](#)

P

passwords

policies, modifying [212](#), [213](#)

user passwords, resetting [213-217](#)

using [209-211](#)

Proof Key for Code Exchange (PKCE) [28](#), [86](#), [113](#), [296](#)

Provider [256](#)

ProviderFactory [255](#)

life cycle [260-262](#)

public clients [28](#)

Q

Quarkus

client, creating [114-116](#)

resource server, creating [116-118](#)

using [113](#), [114](#)

R

READ_ONLY strategy [185](#)

realm roles [135](#)

Recover Password capability [283](#)

Red Hat Single Sign-On

reference link [286](#)

reencrypt [166](#)

refreshing token rotation

enabling [247](#), [248](#)

refresh tokens

lifetimes, managing [245-247](#)

revoking [93](#)

Relying Party (RP) [30](#)

REST APIs and services

securing [100-103](#)

reverse proxy

client information, forwarding [165-167](#)

configuring [164](#)

load across nodes, distributing [165](#)

session affinity [167](#), [168](#)

using [128](#), [129](#)

Rivest-Shamir-Adleman (RSA) signature [299](#)

robust database

configuring [159](#)

role-based access control (RBAC)

using [135](#), [136](#)

S

salt [210](#)

SAML 2.0

significance [34](#), [35](#)

sample application [14-18](#)

features [14](#)

frontend web application [14](#)

logging in, with Keycloak [18-23](#)

REST API [14](#)

running [18](#)

secrets

loading, from external vault [287](#)

Server Administration Guide

reference link [250](#)

server-side web applications

securing [88-90](#)

Service-Level Agreements (SLAs) [161](#)

Service Provider Interface (SPI) [230](#), [254-257](#)

custom provider, deploying [258](#)

custom provider, packaging [258](#)

KeycloakSession component [259](#)

KeycloakSessionFactory component [259](#)

session affinity [167](#), [168](#)

sessions

active sessions, managing [237-239](#)

lifetimes, managing [234-237](#)

managing [233](#), [234](#)

user sessions, expiring prematurely [239-241](#)

signing keys

benefits [283](#)

rotating [283-286](#)

Simple Authentication and Security Layer (SASL) [103](#)

single-page application (SPA) [14](#), [86](#), [107](#)

securing, with dedicated REST API [90](#), [91](#)

securing, with external REST API [93](#), [94](#)

securing, with intermediary REST API [91](#), [93](#)

single sign-on (SSO) [29](#), [51](#), [87](#), [234](#)

social identity providers

integrating with [195](#), [196](#)

Software as a Service (SaaS)-hosted application [84](#)

special redirect URI [98](#)

Spring Boot applications

client, creating [118](#), [119](#)

resource server, creating [120](#), [121](#)

using [118](#)

SSO Session Idle setting [236](#)

SSO Session Max setting [236](#)

Strong Authentication (SA) [2](#)

using [230](#)

stronger authentication [29](#)

subflows [204](#)

T

TAR.GZ [5](#)

third-party identity providers

integrating with [190](#)

third-party-provider [191](#)

Time-Based One-Time Password (TOTP) [218](#)

token exchange [194](#)

tokens

group membership, mapping into [137](#)
lifetimes, managing [243-245](#)
managing [242](#), [243](#)
refreshing token rotation, enabling [247](#), [248](#)
revoking [248-250](#)

Transport Layer Security (TLS) [282](#)

enabling [157-159](#)

U

user accounts

securing [292-295](#)

User Experience (UX) [253](#)

UserInfo endpoint

invoking [52-54](#)

User Interface (UI) [253](#)

users

allowing, to manager their data with Keycloak account console
[196](#)

allowing, to select whether they want to use OTP [220-222](#)

authenticating [42-46](#)

forcing, to authenticate with OTP [222-224](#)

user sessions

expiring, prematurely [239-241](#)

users logging out

dealing with [54](#)

users logging out, in SSO

dealing, considerations [57](#)

ID and access token expiration, leveraging [55](#)

logout, initiating [54](#)

OIDC Back-Channel Logout, leveraging [56](#)

OIDC Front-Channel Logout [56](#)

OIDC Session Management, leveraging [55](#), [56](#)

User Storage SPI [277](#)

V

valid post redirect URIs [20](#)

valid redirect URIs [19](#)

W

Web Application Firewall (WAF) [282](#)

web applications

securing [86](#)- [88](#)

securing, approaches [86](#)

securing, architecture [86](#)

web application security [295](#)

authentication [295](#)

authorization [295](#)

data security [296](#)

firewall [296](#)

logging and monitoring [296](#)

protection, against common attacks [296](#)

regular updates [296](#)

Web Authentication (WebAuthn) [29](#)

authenticating [227-229](#)

enabling, for authentication flow [225-227](#)

security device, registering [227-229](#)

use cases [225](#)

using [224](#), [225](#)

web origins [20](#)

Web origins field [107](#)

WRITABLE strategy [185](#)

Z

ZIP [5](#)

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781804616444>

2. Submit your proof of purchase

3. That's it! We'll send your free PDF and other benefits to your email directly