

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

<packt>

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.

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.

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

Name Search by name Add client scope Change type to

<input type="checkbox"/>	Assigned client s...	Assigned type	Description
<input type="checkbox"/>	myclient-dedicated	none	Dedicated scope and mappers for this client
<input type="checkbox"/>	acr	Default	OpenID Connect scope for add acr (authentication context class reference) to the token
<input type="checkbox"/>	address	Optional	OpenID Connect built-in scope: address

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:

[Clients](#) > [Client details](#) > [Dedicated scopes](#) > [Mapper details](#)

Group Membership

Action ▾

d71819de-fa5a-43c1-b501-9750ddce6c7c

Mapper type	Group Membership
Name * ?	groups
Token Claim Name ?	groups
Full group path ?	<input checked="" type="checkbox"/> On
Add to ID token ?	<input checked="" type="checkbox"/> On
Add to access token ?	<input checked="" type="checkbox"/> On
Add to userinfo ?	<input checked="" type="checkbox"/> On

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](#) 



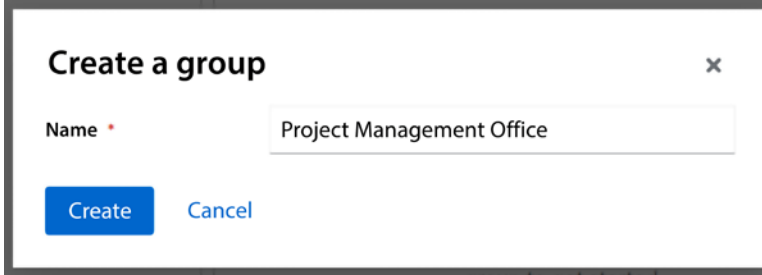
No groups in this realm

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:



Create a group ×

Name *

Create Cancel

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:

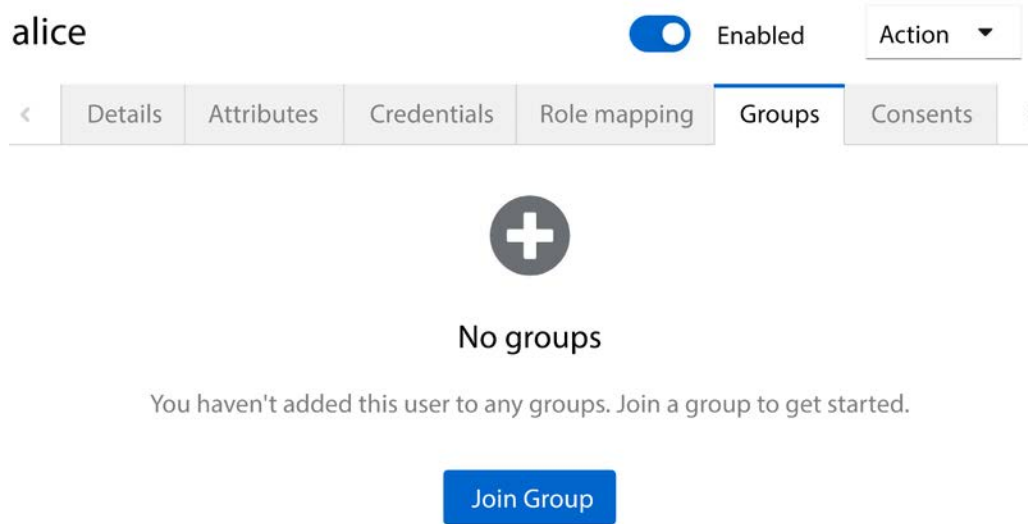


Figure 8.5: Managing group membership for a user

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.

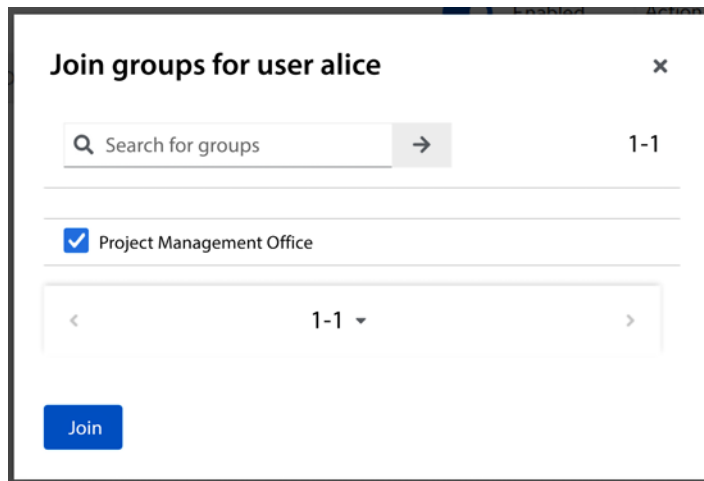


Figure 8.6: Assigning users as a member of a group

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 × Select scope parameters

User [?](#)

alice ×

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 ? openid x Select scope parameters

User ? alice

```
session_state: "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>

