



[Getting Started](#)  
[Server Installation](#)  
[Securing Apps](#)  
[Server Admin](#)  
[Server Development](#)  
[Authorization Services](#)  
[Upgrading](#)  
[Release Notes](#)



[Getting Started](#)

[Server Installation](#)

[Securing Apps](#)

[Server Admin](#)

[Server Development](#)

[Authorization Services](#)

[Upgrading](#)

[Release Notes](#)

# Table of Contents

*{gettingstarted\_name}*

1. Overview

2. Installing and Booting

    2.1. Booting the Server

    2.2. Creating the Admin Account

    2.3. Logging in to the Admin Console

3. Creating a Realm and User

    3.1. Before You Start

    3.2. Creating a New Realm

    3.3. Creating a New User

    3.4. User Account Service

4. Securing a JBoss Servlet Application

    4.1. Before You Start

    4.2. Installing the Client Adapter

    4.3. Downloading, Building, and Deploying Application Code

    4.4. Creating and Registering the Client

    4.5. Configuring the Subsystem

---

{gettingstarted\_name}

# 1. Overview

This guide helps you get started with KeyCloak. It covers server configuration and use of the default database. Advanced deployment options are not covered. For a deeper description of features or configuration options, consult the other reference guides.

---

## 2. Installing and Booting

This section describes how to boot a Keycloak server in standalone mode, set up the initial admin user, and log in to the Keycloak admin console.

### 2.1. Booting the Server

To boot the Keycloak server, go to the `bin` directory of the server distribution and run the `standalone` boot script:

#### *Linux/Unix*

```
$ cd bin  
$ ./standalone.sh
```

#### *Windows*

```
> ...\\bin\\standalone.bat
```

### 2.2. Creating the Admin Account

After the server boots, open <http://localhost:8080/auth> in your web browser. The welcome page will indicate that the server is running.

Enter a username and password to create an initial admin user.

This account will be permitted to log in to the `master` realm's administration console, from which you will create realms and users and register applications to be secured by Keycloak.

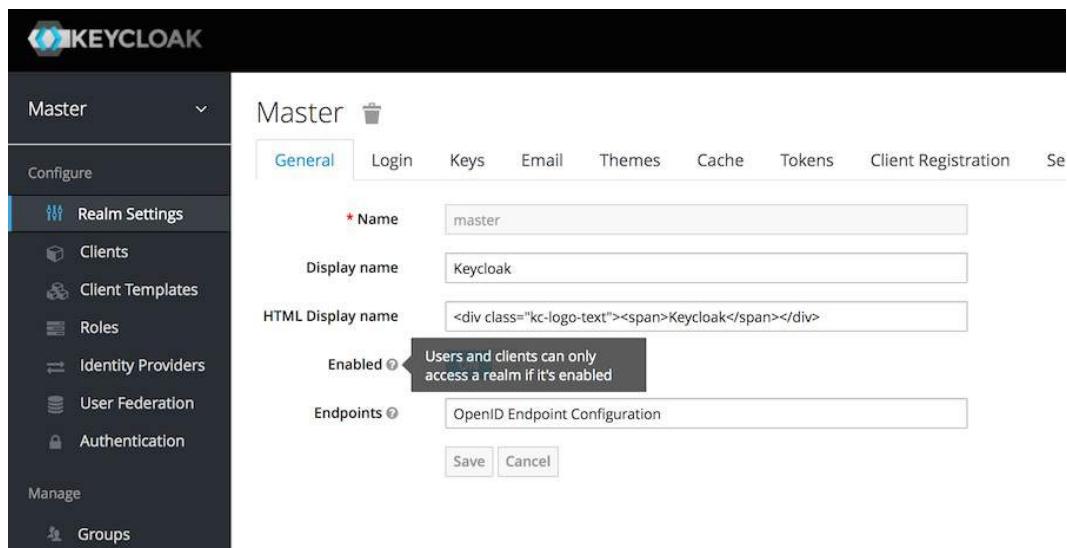
You can only create an initial admin user on the Welcome Page if you connect using `localhost`. This is a security precaution. You can create the initial admin user at the command line with the `add-user-keycloak.sh` script. For more information, see the [{installguide!}](#) and the [{adminguide!}](#).

## 2.3. Logging in to the Admin Console

After you create the initial admin account, use the following steps to log in to the admin console:

1. Click the **Administration Console** link on the **Welcome** page or go directly to the console URL <http://localhost:8080/auth/admin/>
2. Type the username and password you created on the **Welcome** page to open the **KeyCloak Admin Console**.

### *Admin Console*



---

## 3. Creating a Realm and User

In this section you will create a new realm within the KeyCloak admin console and add a new user to that realm. You will use that new user to log in to your new realm and visit the built-in user account service that all users have access to.

### 3.1. Before You Start

Before you can create your first realm, complete the installation of KeyCloak and create the initial admin user as shown in [Installing and Booting](#).

### 3.2. Creating a New Realm

To create a new realm, complete the following steps:

1. Go to <http://localhost:8080/auth/admin/> and log in to the KeyCloak Admin Console using the account you created in [Install and Boot](#).
2. From the **Master** drop-down menu, click **Add Realm**. When you are logged in to the master realm this drop-down menu lists all existing realms.
3. Type `demo` in the **Name** field and click **Create**.

When the realm is created, the main admin console page opens. Notice the current realm is now set to `demo`. Switch between managing the `master` realm and the realm you just created by clicking entries in the **Select realm** drop-down menu.

### 3.3. Creating a New User

To create a new user in the `demo` realm, along with a temporary password for that new user, complete the following steps:

1. From the menu, click **Users** to open the user list page.
2. On the right side of the empty user list, click **Add User** to open the add user page.
3. Enter a name in the `Username` field; this is the only required field. Click **Save** to save the data and open the management page for the new user.
4. Click the **Credentials** tab to set a temporary password for the new user.
5. Type a new password and confirm it. Click **Reset Password** to set the user password to the new one you specified.

This password is temporary and the user will be required to change it after the first login. To create a password that is persistent, flip the **Temporary** switch from **On** to **Off** before clicking **Reset Password**.

### 3.4. User Account Service

1. After you create the new user, log out of the management console by opening the user drop-down menu and selecting **Sign Out**.
2. Go to <http://localhost:8080/auth/realm/demo/account> and log in to the User Account Service of your `demo` realm with the user you just created.

3. Type the username and password you created. You will be required to create a permanent password after you successfully log in, unless you changed the **Temporary** setting to **Off** when you created the password.

The user account service page will open. Every user in a realm has access to this account service by default. From this page, you can update profile information and change or add additional credentials. For more information on this service see the [{adminguide!}](#).

---

## 4. Securing a JBoss Servlet Application

This section describes how to secure a Java servlet application on the WildFly application server by:

- Installing the KeyCloak client adapter on a WildFly application server distribution
- Creating and registering a client application in the KeyCloak admin console
- Configuring the application to be secured by KeyCloak

### 4.1. Before You Start

Before you can secure a Java servlet application, you must complete the installation of KeyCloak and create the initial admin user as shown in [Installing and Booting](#).

There is one caveat: Even though WildFly is bundled with KeyCloak, you cannot use this as an application container. Instead, you must run a separate WildFly instance on the same machine as the KeyCloak server to run your Java servlet application. Run the KeyCloak using a different port than the WildFly, to avoid port conflicts.

To adjust the port used, change the value of the `jboss.socket.binding.port-offset` system property when starting the server from the command line. The value of this property is a number that will be added to the base value of every port opened by the KeyCloak server.

To start the KeyCloak server while also adjusting the port:

### *Linux/Unix*

```
$ cd bin  
$ ./standalone.sh -Djboss.socket.binding.port-offset=100
```

### *Windows*

```
> ...\\bin\\standalone.bat -Djboss.socket.binding.port-off-  
set=100
```

After starting KeyCloak, go to <http://localhost:8180/auth/admin/> to access the admin console.

## 4.2. Installing the Client Adapter

Download the WildFly distribution and extract it from the compressed file into a directory on your machine.

Extract the contents of this file into the root directory of your WildFly distribution.

Run the appropriate script for your platform:

This script will make the necessary edits to the ...  
`/standalone/configuration/standalone.xml`  
file of your app server distribution and may take some time to complete.

Start the application server.

### *Linux/Unix*

```
$ cd bin  
$ ./standalone.sh
```

### *Windows*

```
> ...\\bin\\standalone.bat
```

## 4.3. Downloading, Building, and Deploying Application Code

You must have the following installed on your machine and available in your PATH before you continue:

- Java JDK 8
- Apache Maven 3.1.1 or higher
- Git

Make sure your WildFly application server is started before you continue.

To download, build, and deploy the code, complete the following steps.

### *Clone Project*

```
$ git clone {quickstartRepo_link}  
$ cd {quickstartRepo_dir}/app-profile-jee-vanilla  
$ mvn clean wildfly:deploy
```

During installation, you will see some text scroll by in the application server console window.

To confirm that the application is successfully deployed, go to <http://localhost:8080/vanilla> and a login page should appear.

If you click **Login**, the browser will pop up a BASIC auth login dialog. However, the application is not yet secured by any identity provider, so anything you enter in the dialog box will result in a **Forbidden** message being sent back by the server. You can confirm that the application is currently secured via **BASIC** authentication by finding the setting in the application's **web.xml** file.

## 4.4. Creating and Registering the Client

To define and register the client in the KeyCloak admin console, complete the following steps:

1. Log in to the admin console with your admin account.
2. In the top left drop-down menu select and manage the **Demo** realm. Click **Clients** in the left side menu to open the Clients page.

### *Clients*

The screenshot shows the Keycloak admin console interface. The top navigation bar has a logo on the left and 'Admin' on the right. Below the navigation is a sidebar with a dropdown menu set to 'Demo'. The sidebar contains links for 'Configure', 'Realm Settings', 'Clients' (which is highlighted in blue), 'Client Templates', 'Roles', 'Identity Providers', 'User Federation', and 'Authentication'. The main content area is titled 'Clients' with a question mark icon. It features a search bar and a 'Create' button. A table lists five clients: 'account', 'admin-cli', 'broker', 'realm-management', and 'security-admin-console'. Each row includes columns for 'Client ID', 'Enabled', 'Base URL', and 'Actions' (with 'Edit', 'Export', and 'Delete' buttons). The 'Clients' link in the sidebar is also highlighted in blue.

Client ID	Enabled	Base URL	Actions
account	True	/auth/realm/demo/account	Edit Export Delete
admin-cli	True	Not defined	Edit Export Delete
broker	True	Not defined	Edit Export Delete
realm-management	True	Not defined	Edit Export Delete
security-admin-console	True	/auth/admin/demo/console/index.html	Edit Export Delete

3. On the right side, click **Create**.
4. Complete the fields as shown here:

### *Add Client*

The screenshot shows the Keycloak Admin Console interface. The left sidebar is titled 'Demo' and contains sections for 'Configure' (Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication) and 'Manage' (Groups). The 'Clients' section is currently selected. The main content area is titled 'Add Client'. It includes fields for 'Import' (with a 'Select file' button), 'Client ID' (set to 'vanilla'), 'Client Protocol' (set to 'openid-connect'), 'Client Template' (selected from a dropdown), and 'Root URL' (set to 'http://localhost:8080/vanilla'). At the bottom are 'Save' and 'Cancel' buttons.

5. Click **Save** to create the client application entry.
6. Click the **Installation** tab in the KeyCloak admin console to obtain a configuration template.
7. Select **Keycloak OIDC JBoss Subsystem XML** to generate an XML template. Copy the contents for use in the next section.

### *Template XML*

The screenshot shows the Keycloak Admin Console interface for the 'vanilla' client. The left sidebar is identical to the previous screenshot. The main content area shows the 'vanilla' client details. The 'Installation' tab is selected. Under 'Format', 'Keycloak OIDC JBoss Subsystem XML' is chosen. A 'Download' button is available. Below it, the XML code is displayed:

```
<secure-deployment name="WAR MODULE NAME.war">
<realm>demo</realm>
<auth-server-url>http://localhost:8180/auth</auth-server-url>
<public-client>true</public-client>
<ssl-required>EXTERNAL</ssl-required>
<resource>vanilla</resource>
</secure-deployment>
```

## 4.5. Configuring the Subsystem

To configure the WildFly instance that the application is deployed on so that this app is secured by Keycloak, complete the following steps.

1. Open the `standalone/configuration/standalone.xml` file in the WildFly instance that the application is deployed on and search for the following text:

```
<subsystem xmlns="urn:jboss:domain:keycloak:1.1"/>
```

2. Modify this text to prepare the file for pasting in contents from the **Keycloak OIDC JBoss Subsystem XML** template we obtained Keycloak admin console **Installation** tab by changing the XML entry from self-closing to using a pair of opening and closing tags:

```
<subsystem xmlns="urn:jboss:domain:keycloak:1.1">  
</subsystem>
```

3. Paste the contents of the template within the `<subsystem>` element, as shown in this example:

```
<subsystem xmlns="urn:jboss:domain:keycloak:1.1">  
  <secure-deployment name="WAR MODULE NAME.war">  
    <realm>demo</realm>  
    <auth-server-url>http://localhost:8180/auth</auth-  
server-url>  
    <public-client>true</public-client>  
    <ssl-required>EXTERNAL</ssl-required>  
    <resource>vanilla</resource>  
  </secure-deployment>  
</subsystem>
```

4. Change the `name` to `vanilla.war`:

```
<subsystem xmlns="urn:jboss:domain:keycloak:1.1">
  <secure-deployment name="vanilla.war">
    ...
</subsystem>
```

5. Reboot the application server.
6. Go to <http://localhost:8080/vanilla> and click **Login**. When the Keycloak login page opens, log in using the user you created in [Creating a New User](#).

Last updated 2019-06-13 12:48:56 MESZ

# Table of Contents

{installguide!}

## 1. Guide Overview

### 1.1. Recommended Additional External Documentation

## 2. Installation

### 2.1. System Requirements

### 2.2. Distribution Directory Structure

## 3. Choosing an Operating Mode

### 3.1. Standalone Mode

#### 3.1.1. Standalone Boot Script

#### 3.1.2. Standalone Configuration

### 3.2. Standalone Clustered Mode

#### 3.2.1. Standalone Clustered Configuration

#### 3.2.2. Standalone Clustered Boot Script

### 3.3. Domain Clustered Mode

#### 3.3.1. Domain Configuration

#### 3.3.2. Host Controller Configuration

#### 3.3.3. Server Instance Working Directories

#### 3.3.4. Domain Boot Script

#### 3.3.5. Clustered Domain Example

### 3.4. Cross-Datacenter Replication Mode

#### 3.4.1. Prerequisites

#### 3.4.2. Technical details

#### 3.4.3. Request processing

#### 3.4.4. Modes

#### 3.4.5. Database

#### 3.4.6. Infinispan caches

- 3.4.7. Communication details
- 3.4.8. Basic setup
- 3.4.9. Administration of Cross DC deployment
- 3.4.10. Bringing sites offline and online
- 3.4.11. State transfer
- 3.4.12. Clear caches
- 3.4.13. Tuning the JDG cache configuration
- 3.4.14. SYNC or ASYNC backups
- 3.4.15. Troubleshooting

## 4. Manage Subsystem Configuration

- 4.1. Configure SPI Providers
- 4.2. Start the WildFly CLI
- 4.3. CLI Embedded Mode
- 4.4. CLI GUI Mode
- 4.5. CLI Scripting
- 4.6. CLI Recipes
  - 4.6.1. Change the web context of the server
  - 4.6.2. Set the global default theme
  - 4.6.3. Add a new SPI and a provider
  - 4.6.4. Disable a provider
  - 4.6.5. Change the default provider for an SPI
  - 4.6.6. Configure the dblock SPI
  - 4.6.7. Add or change a single property value for a provider
  - 4.6.8. Remove a single property from a provider
  - 4.6.9. Set values on a provider property of type `List`

## 5. Profiles

## 6. Relational Database Setup

- 6.1. RDBMS Setup Checklist
- 6.2. Package the JDBC Driver
- 6.3. Declare and Load JDBC Driver
- 6.4. Modify the Keycloak Datasource
- 6.5. Database Configuration
- 6.6. Unicode Considerations for Databases
  - 6.6.1. Oracle Database
  - 6.6.2. Microsoft SQL Server Database
  - 6.6.3. MySQL Database
  - 6.6.4. PostgreSQL Database
- 7. Network Setup
  - 7.1. Bind Addresses
  - 7.2. Socket Port Bindings
  - 7.3. Setting up HTTPS/SSL
    - 7.3.1. Enabling SSL/HTTPS for the Keycloak Server
  - 7.4. Outgoing HTTP Requests
    - 7.4.1. Proxy Mappings for Outgoing HTTP Requests
    - 7.4.2. Outgoing HTTPS Request Truststore
- 8. Clustering
  - 8.1. Recommended Network Architecture
  - 8.2. Clustering Example
  - 8.3. Setting Up a Load Balancer or Proxy
    - 8.3.1. Identifying Client IP Addresses
    - 8.3.2. Enable HTTPS/SSL with a Reverse Proxy
    - 8.3.3. Verify Configuration
    - 8.3.4. Using the Built-In Load Balancer
    - 8.3.5. Configuring Other Load Balancers
  - 8.4. Sticky sessions

- 8.4.1. Disable adding the route
  - 8.5. Multicast Network Setup
  - 8.6. Securing Cluster Communication
  - 8.7. Serialized Cluster Startup
  - 8.8. Booting the Cluster
  - 8.9. Troubleshooting
- 9. Server Cache Configuration
    - 9.1. Eviction and Expiration
    - 9.2. Replication and Failover
    - 9.3. Disabling Caching
    - 9.4. Clearing Caches at Runtime
-

{installguide!}

# 1. Guide Overview

The purpose of this guide is to walk through the steps that need to be completed prior to booting up the KeyCloak server for the first time. If you just want to test drive KeyCloak, it pretty much runs out of the box with its own embedded and local-only database. For actual deployments that are going to be run in production you'll need to decide how you want to manage server configuration at runtime (standalone or domain mode), configure a shared database for KeyCloak storage, set up encryption and HTTPS, and finally set up KeyCloak to run in a cluster. This guide walks through each and every aspect of any pre-boot decisions and setup you must do prior to deploying the server.

One thing to particularly note is that KeyCloak is derived from the WildFly Application Server. Many aspects of configuring KeyCloak revolve around WildFly configuration elements. Often this guide will direct you to documentation outside of the manual if you want to dive into more detail.

## 1.1. Recommended Additional External Documentation

KeyCloak is built on top of the WildFly application server and it's sub-projects like Infinispan (for caching) and Hibernate (for persistence). This guide only covers basics for infrastructure-level configuration. It is highly recommended that you peruse the documentation for WildFly and its sub projects. Here is the link to the documentation:

- [{appserver\\_admindoc\\_name}](#)

---

## 2. Installation

### 2.1. System Requirements

These are the requirements to run the KeyCloak authentication server:

- Can run on any operating system that runs Java
- Java 8 JDK
- zip or gzip and tar
- At least 512M of RAM
- At least 1G of diskspace
- A shared external database like PostgreSQL, MySQL, Oracle, etc.  
KeyCloak requires an external shared database if you want to run in a cluster. Please see the [database configuration](#) section of this guide for more information.
- Network multicast support on your machine if you want to run in a cluster. KeyCloak can be clustered without multicast, but this requires a bunch of configuration changes. Please see the [clustering](#) section of this guide for more information.
- On Linux, it is recommended to use `/dev/urandom` as a source of random data to prevent KeyCloak hanging due to lack of available entropy, unless `/dev/random` usage is mandated by your security policy. To achieve that on Oracle JDK 8 and OpenJDK 8, set the `java.security.egd` system property on startup to `file:/dev/urandom`.

## 2.2. Distribution Directory Structure

This chapter walks you through the directory structure of the server distribution.

### *distribution directory structure*



Let's examine the purpose of some of the directories:

#### ***bin/***

This contains various scripts to either boot the server or perform some other management action on the server.

#### ***domain/***

This contains configuration files and working directory when running KeyCloak in [domain mode](#).

#### ***modules/***

These are all the Java libraries used by the server.

#### ***standalone/***

This contains configuration files and working directory when running KeyCloak in [standalone mode](#).

## ***themes/***

This directory contains all the html, style sheets, JavaScript files, and images used to display any UI screen displayed by the server. Here you can modify an existing theme or create your own. See the [{developerguide!}](#) for more information on this.

## 3. Choosing an Operating Mode

Before deploying Keycloak in a production environment you need to decide which type of operating mode you are going to use. Will you run Keycloak within a cluster? Do you want a centralized way to manage your server configurations? Your choice of operating mode effects how you configure databases, configure caching and even how you boot the server.

The Keycloak is built on top of the WildFly Application Server. This guide will only go over the basics for deployment within a specific mode. If you want specific information on this, a better place to go would be the [{appserver\\_admindoc\\_name}](#).

### 3.1. Standalone Mode

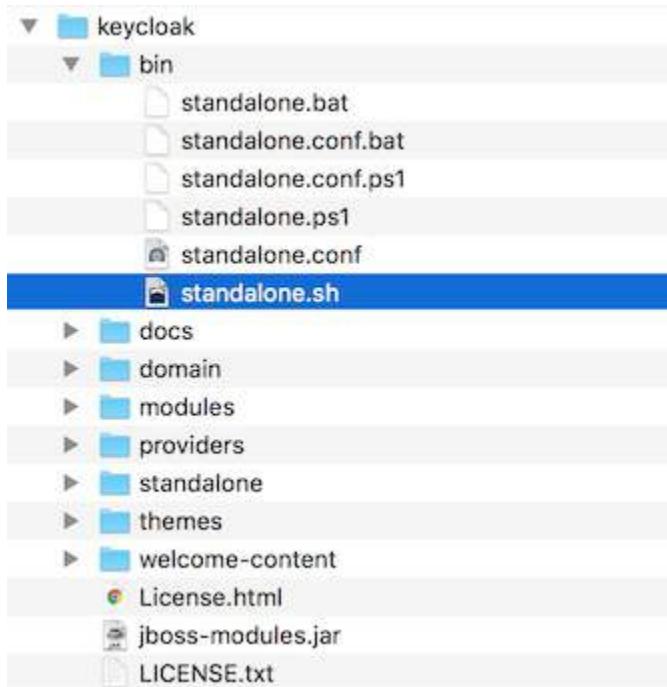
Standalone operating mode is only useful when you want to run one, and only one Keycloak server instance. It is not usable for clustered deployments and all caches are non-distributed and local-only. It is not recommended that you use standalone mode in production as you will have a single point of failure. If your standalone mode server goes down, users will not be able to log in. This mode is really only useful to test drive and play with the features of Keycloak

#### 3.1.1. Standalone Boot Script

When running the server in standalone mode, there is a specific script you need to run to boot the server depending on your operating system.

These scripts live in the `bin/` directory of the server distribution.

### *Standalone Boot Scripts*



To boot the server:

#### *Linux/Unix*

```
$ .../bin/standalone.sh
```

#### *Windows*

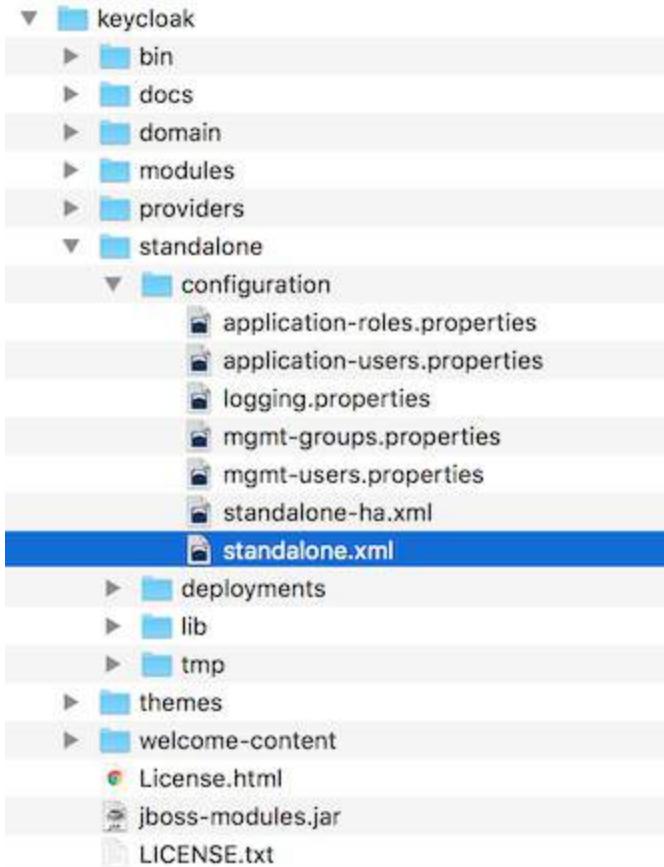
```
> ...\\bin\\standalone.bat
```

### **3.1.2. Standalone Configuration**

The bulk of this guide walks you through how to configure infrastructure level aspects of KeyCloak. These aspects are configured in a configuration file that is specific to the application server that KeyCloak is a derivative of. In the standalone operation mode, this file lives in `../standalone.conf`.

*dalone/configuration/standalone.xml*. This file is also used to configure non-infrastructure level things that are specific to Keycloak components.

### *Standalone Config File*



Any changes you make to this file while the server is running will not take effect and may even be overwritten by the server. Instead use the command line scripting or the web console of WildFly. See the [{appserver-admindoc\\_name}](#) for more information.

## 3.2. Standalone Clustered Mode

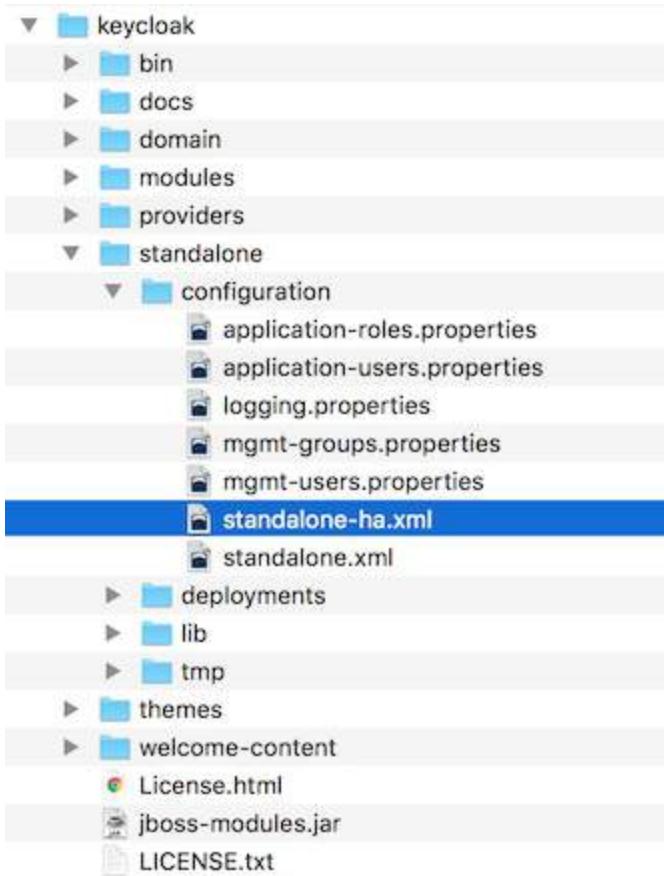
Standalone clustered operation mode is for when you want to run Key-

Cloak within a cluster. This mode requires that you have a copy of the KeyCloak distribution on each machine you want to run a server instance. This mode can be very easy to deploy initially, but can become quite cumbersome. To make a configuration change you'll have to modify each distribution on each machine. For a large cluster this can become time consuming and error prone.

### 3.2.1. Standalone Clustered Configuration

The distribution has a mostly pre-configured app server configuration file for running within a cluster. It has all the specific infrastructure settings for networking, databases, caches, and discovery. This file resides in `../standalone/configuration/standalone-ha.xml`. There's a few things missing from this configuration. You can't run KeyCloak in a cluster without configuring a shared database connection. You also need to deploy some type of load balancer in front of the cluster. The [clustering](#) and [database](#) sections of this guide walk you through these things.

*[Standalone HA Config](#)*

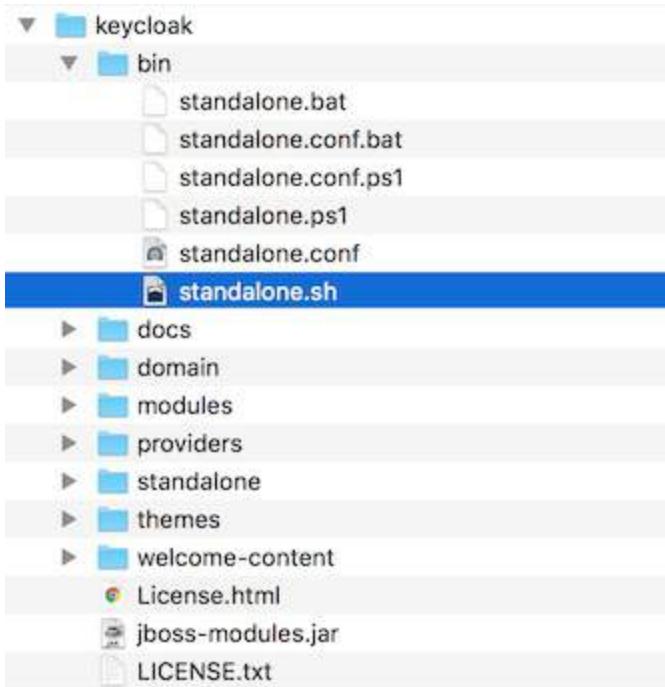


Any changes you make to this file while the server is running will not take effect and may even be overwritten by the server. Instead use the command line scripting or the web console of WildFly. See the [{appserver-admindoc\\_name}](#) for more information.

### 3.2.2. Standalone Clustered Boot Script

You use the same boot scripts to start KeyCloak as you do in standalone mode. The difference is that you pass in an additional flag to point to the HA config file.

#### *Standalone Clustered Boot Scripts*



To boot the server:

#### *Linux/Unix*

```
$ .../bin/standalone.sh --server-config=standalone-ha.xml
```

#### *Windows*

```
> ...\\bin\\standalone.bat --server-config=standalone-ha.xml
```

### 3.3. Domain Clustered Mode

Domain mode is a way to centrally manage and publish the configuration for your servers.

Running a cluster in standard mode can quickly become aggravating as the cluster grows in size. Every time you need to make a configuration change, you have to perform it on each node in the cluster. Domain mode solves this problem by providing a central place to store and publish

configuration. It can be quite complex to set up, but it is worth it in the end. This capability is built into the WildFly Application Server which Keycloak derives from.

The guide will go over the very basics of domain mode. Detailed steps on how to set up domain mode in a cluster should be obtained from the [\*{appserver\\_admindoc\\_name}\*](#).

Here are some of the basic concepts of running in domain mode.

### **domain controller**

The domain controller is a process that is responsible for storing, managing, and publishing the general configuration for each node in the cluster. This process is the central point from which nodes in a cluster obtain their configuration.

### **host controller**

The host controller is responsible for managing server instances on a specific machine. You configure it to run one or more server instances. The domain controller can also interact with the host controllers on each machine to manage the cluster. To reduce the number of running process, a domain controller also acts as a host controller on the machine it runs on.

### **domain profile**

A domain profile is a named set of configuration that can be used by a server to boot from. A domain controller can define multiple do-

main profiles that are consumed by different servers.

## server group

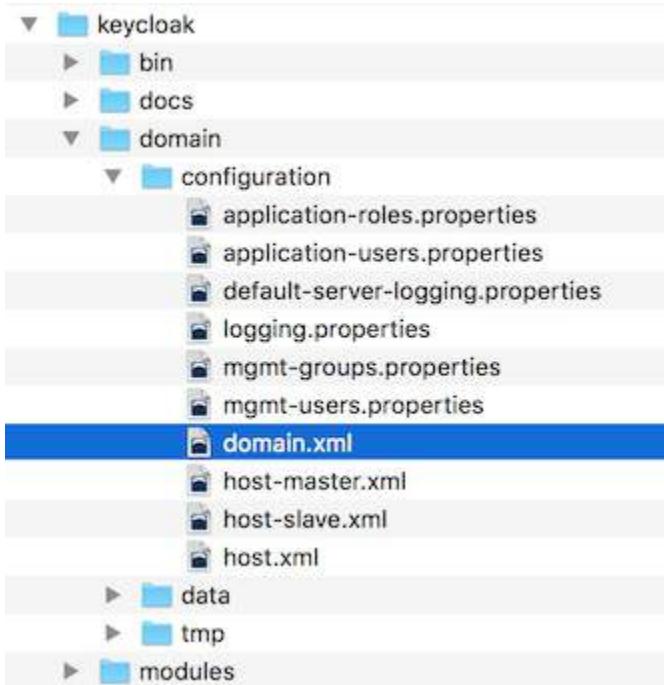
A server group is a collection of servers. They are managed and configured as one. You can assign a domain profile to a server group and every service in that group will use that domain profile as their configuration.

In domain mode, a domain controller is started on a master node. The configuration for the cluster resides in the domain controller. Next a host controller is started on each machine in the cluster. Each host controller deployment configuration specifies how many Keycloak server instances will be started on that machine. When the host controller boots up, it starts as many Keycloak server instances as it was configured to do. These server instances pull their configuration from the domain controller.

### 3.3.1. Domain Configuration

Various other chapters in this guide walk you through configuring various aspects like databases, HTTP network connections, caches, and other infrastructure related things. While standalone mode uses the *standalone.xml* file to configure these things, domain mode uses the *.../domain/configuration/domain.xml* configuration file. This is where the domain profile and server group for the Keycloak server are defined.

#### *domain.xml*



Any changes you make to this file while the domain controller is running will not take effect and may even be overwritten by the server. Instead use the command line scripting or the web console of WildFly. See the [\*{appserver\\_admindoc\\_name}\*](#) for more information.

Let's look at some aspects of this *domain.xml* file. The `auth-server-standalone` and `auth-server-clustered` profile XML blocks are where you are going to make the bulk of your configuration decisions. You'll be configuring things here like network connections, caches, and database connections.

### *auth-server profile*

```
<profiles>
    <profile name="auth-server-standalone">
        ...
    </profile>
    <profile name="auth-server-clustered">
```

```
...
```

```
</profile>
```

The `auth-server-standalone` profile is a non-clustered setup. The `auth-server-clustered` profile is the clustered setup.

If you scroll down further, you'll see various `socket-binding-groups` defined.

### *socket-binding-groups*

```
<socket-binding-groups>
    <socket-binding-group name="standard-sockets" de-
fault-interface="public">
        ...
        </socket-binding-group>
    <socket-binding-group name="ha-sockets" default-in-
terface="public">
        ...
        </socket-binding-group>
    <!-- load-balancer-sockets should be removed in
production systems and replaced with a better software or
hardware based one -->
    <socket-binding-group name="load-balancer-sockets"
default-interface="public">
        ...
        </socket-binding-group>
</socket-binding-groups>
```

This config defines the default port mappings for various connectors that are opened with each Keycloak server instance. Any value that contains  `${...}` is a value that can be overridden on the command line with the `-D` switch, i.e.

```
$ domain.sh -Djboss.http.port=80
```

The definition of the server group for KeyCloak resides in the `server-groups` XML block. It specifies the domain profile that is used ( `default` ) and also some default boot arguments for the Java VM when the host controller boots an instance. It also binds a `socket-binding-group` to the server group.

### *server group*

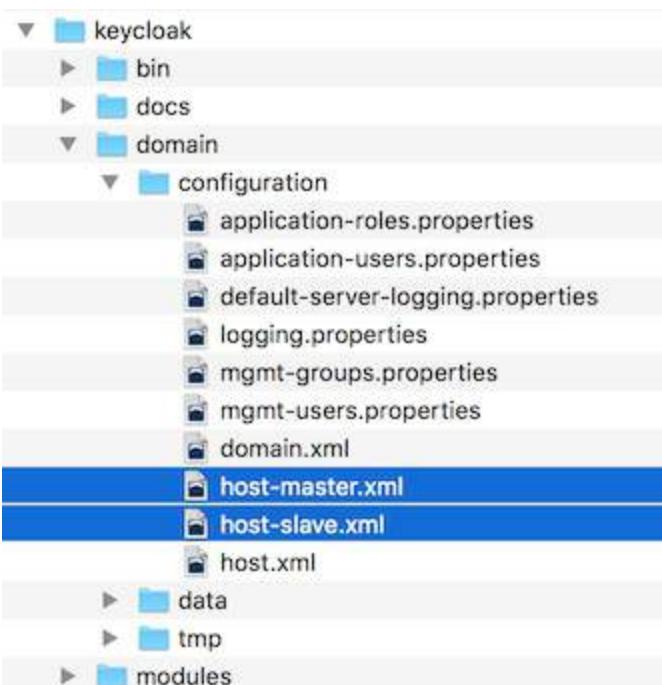
```
<server-groups>
    <!-- load-balancer-group should be removed in production systems and replaced with a better software or hardware based one -->
    <server-group name="load-balancer-group" profile="load-balancer">
        <jvm name="default">
            <heap size="64m" max-size="512m"/>
        </jvm>
        <socket-binding-group ref="load-balancer-sockets"/>
    </server-group>
    <server-group name="auth-server-group" profile="auth-server-clustered">
        <jvm name="default">
            <heap size="64m" max-size="512m"/>
        </jvm>
        <socket-binding-group ref="ha-sockets"/>
    </server-group>
</server-groups>
```

### **3.3.2. Host Controller Configuration**

KeyCloak comes with two host controller configuration files that reside in the `../domain/configuration/` directory: `host-master.xml` and `host-slave.xml`. `host-master.xml` is configured to boot up a domain controller, a load balancer, and one KeyCloak server instance. `host-slave.xml` is configured to talk to the domain controller and boot up one KeyCloak server instance.

The load balancer is not a required service. It exists so that you can easily test drive clustering on your development machine. While usable in production, you have the option of replacing it if you have a different hardware or software based load balancer you want to use.

## *Host Controller Config*



To disable the load balancer server instance, edit *host-master.xml* and comment out or remove the "load-balancer" entry.

```
<servers>
    <!-- remove or comment out next line -->
    <server name="load-balancer" group="loadbalancer-
group"/>
    ...
</servers>
```

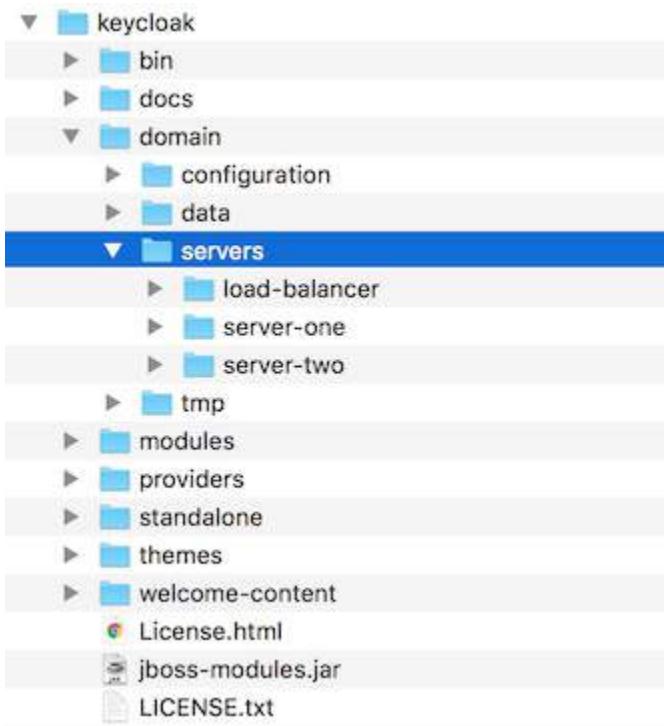
Another interesting thing to note about this file is the declaration of the authentication server instance. It has a `port-offset` setting. Any network port defined in the `domain.xml` `socket-binding-group` or the server group will have the value of `port-offset` added to it. For this example domain setup we do this so that ports opened by the load balancer server don't conflict with the authentication server instance that is started.

```
<servers>
  ...
    <server name="server-one" group="auth-server-group"
auto-start="true">
      <socket-bindings port-offset="150"/>
    </server>
</servers>
```

### 3.3.3. Server Instance Working Directories

Each Keycloak server instance defined in your host files creates a working directory under `../domain/servers/{SERVER NAME}`. Additional configuration can be put there, and any temporary, log, or data files the server instance needs or creates go there too. The structure of these per server directories ends up looking like any other WildFly booted server.

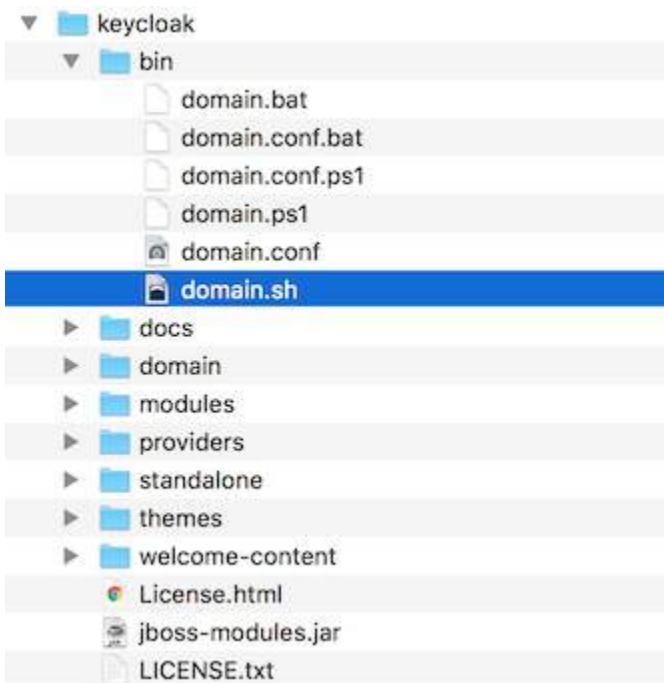
#### *Working Directories*



### 3.3.4. Domain Boot Script

When running the server in domain mode, there is a specific script you need to run to boot the server depending on your operating system. These scripts live in the *bin/* directory of the server distribution.

#### *Domain Boot Script*



To boot the server:

### *Linux/Unix*

```
$ .../bin/domain.sh --host-config=host-master.xml
```

### *Windows*

```
> ...\\bin\\domain.bat --host-config=host-master.xml
```

When running the boot script you will need pass in the host controlling configuration file you are going to use via the `--host-config` switch.

#### **3.3.5. Clustered Domain Example**

You can test drive clustering using the out-of-the-box `domain.xml` configuration. This example domain is meant to run on one machine and boots up:

- a domain controller
- an HTTP load balancer
- 2 KeyCloak server instances

To simulate running a cluster on two machines, you'll run the `domain.sh` script twice to start two separate host controllers. The first will be the master host controller which will start a domain controller, an HTTP load balancer, and one KeyCloak authentication server instance. The second will be a slave host controller that only starts up an authentication server instance.

#### **Setup Slave Connection to Domain Controller**

Before you can boot things up though, you have to configure the slave host controller so that it can talk securely to the domain controller. If you do not do this, then the slave host will not be able to obtain the centralized configuration from the domain controller. To set up a secure connection, you have to create a server admin user and a secret that will be shared between the master and the slave. You do this by running the `.../bin/add-user.sh` script.

When you run the script select `Management User` and answer `yes` when it asks you if the new user is going to be used for one AS process to connect to another. This will generate a secret that you'll need to cut and paste into the `.../domain/configuration/host-slave.xml` file.

### Add App Server Admin

```
$ add-user.sh
What type of user do you wish to add?
a) Management User (mgmt-users.properties)
b) Application User (application-users.properties)
(a): a
Enter the details of the new user to add.
Using realm 'ManagementRealm' as discovered from the existing property files.
Username : admin
Password recommendations are listed below. To modify these restrictions edit the add-user.properties configuration file.
- The password should not be one of the following restricted values {root, admin, administrator}
- The password should contain at least 8 characters, 1 alphabetic character(s), 1 digit(s), 1 non-alphanumeric symbol(s)
- The password should be different from the username
Password :
Re-enter Password :
What groups do you want this user to belong to? (Please enter a comma separated list, or leave blank for none)[ ]:
```

```
About to add user 'admin' for realm 'ManagementRealm'  
Is this correct yes/no? yes  
Added user 'admin' to file '/.../standalone/configuration/mgmt-users.properties'  
Added user 'admin' to file '/.../domain/configuration/mgmt-users.properties'  
Added user 'admin' with groups to file '/.../standalone/configuration/mgmt-groups.properties'  
Added user 'admin' with groups to file '/.../domain/configuration/mgmt-groups.properties'  
Is this new user going to be used for one AS process to connect to another AS process?  
e.g. for a slave host controller connecting to the master or for a Remoting connection for server to server EJB calls.  
yes/no? yes  
To represent the user add the following to the server-identities definition <secret value="bwtdtdDEyMyE=" />
```

The add-user.sh does not add user to Keycloak server but to the underlying JBoss Enterprise Application Platform. The credentials used and generated in the above script are only for example purpose. Please use the ones generated on your system.

Now cut and paste the secret value into the *.../domain/configuration/host-slave.xml* file as follows:

```
<management>  
  <security-realms>  
    <security-realm name="ManagementRealm">  
      <server-identities>  
        <secret value="bwtdtdDEyMyE=" />  
      </server-identities>
```

You will also need to add the *username* of the created user in the *.../domain/configuration/host-slave.xml* file as follows:

*main/configuration/host-slave.xml* file:

```
<remote security-realm="ManagementRealm" userna-  
me="admin">
```

## Run the Boot Scripts

Since we're simulating a two node cluster on one development machine, you'll run the boot script twice:

### *Boot up master*

```
$ domain.sh --host-config=host-master.xml
```

### *Boot up slave*

```
$ domain.sh --host-config=host-slave.xml
```

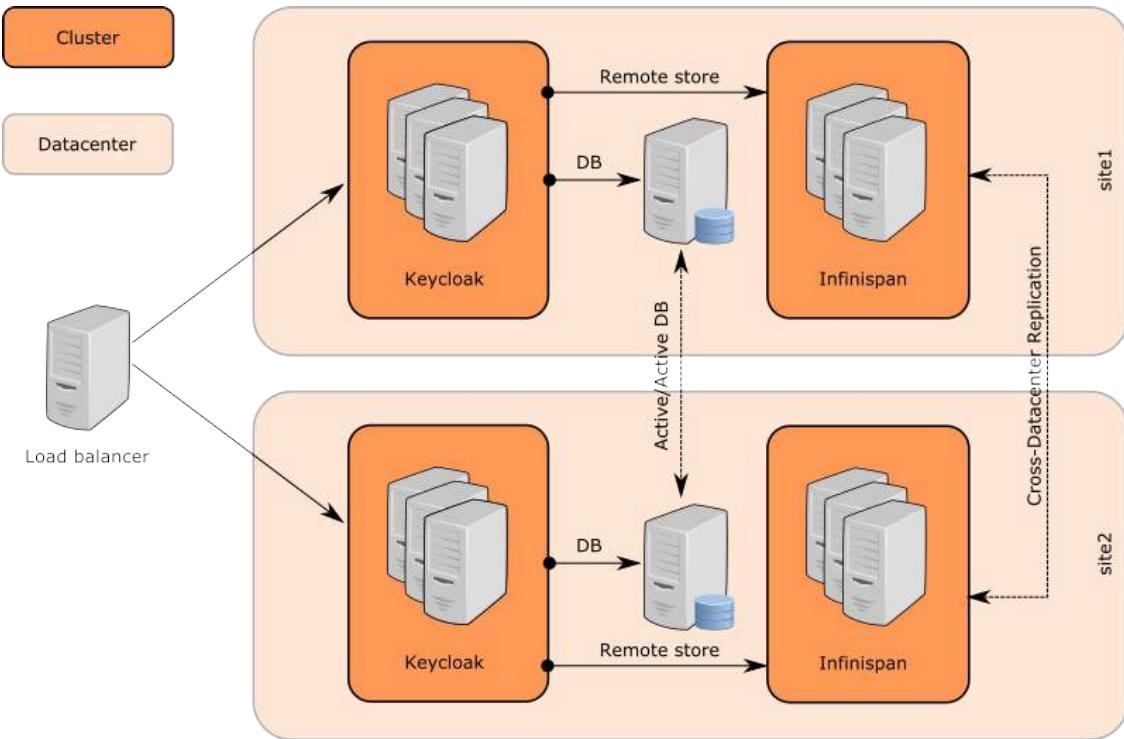
To try it out, open your browser and go to <http://localhost:8080/auth>.

## 3.4. Cross-Datacenter Replication Mode

Cross-Datacenter Replication mode is for when you want to run KeyCloak in a cluster across multiple data centers, most typically using data center sites that are in different geographic regions. When using this mode, each data center will have its own cluster of Keycloak servers.

This documentation will refer the following example architecture diagram to illustrate and describe a simple Cross-Datacenter Replication use case.

### *Example Architecture Diagram*



### 3.4.1. Prerequisites

As this is an advanced topic, we recommend you first read the following, which provide valuable background knowledge:

- [Clustering with Keycloak](#) When setting up for Cross-Datacenter Replication, you will use more independent Keycloak clusters, so you must understand how a cluster works and the basic concepts and requirements such as load balancing, shared databases, and multicasting.
- [JBoss Data Grid Cross-Datacenter Replication](#) Keycloak uses JBoss Data Grid (JDG) for the replication of Infinispan data between the data centers.

### 3.4.2. Technical details

This section provides an introduction to the concepts and details of how Keycloak Cross-Datacenter Replication is accomplished.

## *Data*

KeyCloak is stateful application. It uses the following as data sources:

- A database is used to persist permanent data, such as user information.
- An Infinispan cache is used to cache persistent data from the database and also to save some short-lived and frequently-changing metadata, such as for user sessions. Infinispan is usually much faster than a database, however the data saved using Infinispan are not permanent and is not expected to persist across cluster restarts.

In our example architecture, there are two data centers called `site1` and `site2`. For Cross-Datacenter Replication, we must make sure that both sources of data work reliably and that KeyCloak servers from `site1` are eventually able to read the data saved by KeyCloak servers on `site2`.

Based on the environment, you have the option to decide if you prefer:

- Reliability - which is typically used in Active/Active mode. Data written on `site1` must be visible immediately on `site2`.
- Performance - which is typically used in Active/Passive mode. Data written on `site1` does not need to be visible immediately on `site2`. In some cases, the data may not be visible on `site2` at all.

For more details, see [Modes](#).

### **3.4.3. Request processing**

An end user's browser sends an HTTP request to the [front end load balancer](#)

[lancer](#). This load balancer is usually HTTPD or WildFly with mod\_cluster, NGINX, HA Proxy, or perhaps some other kind of software or hardware load balancer.

The load balancer then forwards the HTTP requests it receives to the underlying Keycloak instances, which can be spread among multiple data centers. Load balancers typically offer support for [sticky sessions](#), which means that the load balancer is able to always forward all HTTP requests from the same user to the same Keycloak instance in same data center.

HTTP requests that are sent from client applications to the load balancer are called `backchannel requests`. These are not seen by an end user's browser and therefore can not be part of a sticky session between the user and the load balancer. For backchannel requests, the loadbalancer can forward the HTTP request to any Keycloak instance in any data center. This is challenging as some OpenID Connect and some SAML flows require multiple HTTP requests from both the user and the application. Because we can not reliably depend on sticky sessions to force all the related requests to be sent to the same Keycloak instance in the same data center, we must instead replicate some data across data centers, so the data are seen by subsequent HTTP requests during a particular flow.

#### 3.4.4. Modes

According your requirements, there are two basic operating modes for Cross-Datacenter Replication:

- Active/Passive - Here the users and client applications send the re-

quests just to the Keycloak nodes in just a single data center. The second data center is used just as a `backup` for saving the data. In case of the failure in the main data center, the data can be usually restored from the second data center.

- Active/Active - Here the users and client applications send the requests to the Keycloak nodes in both data centers. It means that data need to be visible immediately on both sites and available to be consumed immediately from Keycloak servers on both sites. This is especially true if Keycloak server writes some data on `site1`, and it is required that the data are available immediately for reading by Keycloak servers on `site2` immediately after the write on `site1` is finished.

The active/passive mode is better for performance. For more information about how to configure caches for either mode, see: [SYNC or ASYNC backups](#).

### 3.4.5. Database

Keycloak uses a relational database management system (RDBMS) to persist some metadata about realms, clients, users, and so on. See [this chapter](#) of the server installation guide for more details. In a Cross-Datacenter Replication setup, we assume that either both data centers talk to the same database or that every data center has its own database node and both database nodes are synchronously replicated across the data centers. In both cases, it is required that when a Keycloak server on `site1` persists some data and commits the transaction, those data are immediately visible by subsequent DB transactions on `site2`.

Details of DB setup are out-of-scope for Keycloak, however many

RDBMS vendors like MariaDB and Oracle offer replicated databases and synchronous replication. We test KeyCloak with these vendors:

- Oracle Database 12c Release 1 (12.1) RAC
- Galera 3.12 cluster for MariaDB server version 10.1.19-MariaDB

### 3.4.6. Infinispan caches

This section begins with a high level description of the Infinispan caches. More details of the cache setup follow.

#### *Authentication sessions*

In KeyCloak we have the concept of authentication sessions. There is a separate Infinispan cache called `authenticationSessions` used to save data during authentication of particular user. Requests from this cache usually involve only a browser and the KeyCloak server, not the application. Here we can rely on sticky sessions and the `authenticationSessions` cache content does not need to be replicated across data centers, even if you are in Active/Active mode.

#### *Caching and invalidation of persistent data*

KeyCloak uses Infinispan to cache persistent data to avoid many unnecessary requests to the database. Caching improves performance, however it adds an additional challenge. When some KeyCloak server updates any data, all other KeyCloak servers in all data centers need to be aware of it, so they invalidate particular data from their caches. KeyCloak uses local Infinispan caches called `realms`, `users`, and `authorization` to cache persistent data.

We use a separate cache, `work`, which is replicated across all data cen-

ters. The work cache itself does not cache any real data. It is used only for sending invalidation messages between cluster nodes and data centers. In other words, when data is updated, such as the user `john`, the Keycloak node sends the invalidation message to all other cluster nodes in the same data center and also to all other data centers. After receiving the invalidation notice, every node then invalidates the appropriate data from their local cache.

### *User sessions*

There are Infinispan caches called `sessions`, `clientSessions`, `offlineSessions`, and `offlineClientSessions`, all of which usually need to be replicated across data centers. These caches are used to save data about user sessions, which are valid for the length of a user's browser session. The caches must handle the HTTP requests from the end user and from the application. As described above, sticky sessions can not be reliably used in this instance, but we still want to ensure that subsequent HTTP requests can see the latest data. For this reason, the data are usually replicated across data centers.

### *Brute force protection*

Finally the `loginFailures` cache is used to track data about failed logins, such as how many times the user `john` entered a bad password. The details are described [here](#). It is up to the admin whether this cache should be replicated across data centers. To have an accurate count of login failures, the replication is needed. On the other hand, not replicating this data can save some performance. So if performance is more important than accurate counts of login failures, the replication can be avoided.

For more detail about how caches can be configured see [Tuning the JDG cache configuration](#).

### 3.4.7. Communication details

Keycloak uses multiple, separate clusters of Infinispan caches. Every Keycloak node is in the cluster with the other Keycloak nodes in same data center, but not with the Keycloak nodes in different data centers. A Keycloak node does not communicate directly with the Keycloak nodes from different data centers. Keycloak nodes use external JDG (actually Infinispan servers) for communication across data centers.

This is done using the [Infinispan HotRod protocol](#).

The Infinispan caches on the Keycloak side must be configured with the [remoteStore](#) to ensure that data are saved to the remote cache. There is separate Infinispan cluster between JDG servers, so the data saved on JDG1 on `site1` are replicated to JDG2 on `site2`.

Finally, the receiving JDG server notifies the Keycloak servers in its cluster through the Client Listeners, which are a feature of the HotRod protocol. Keycloak nodes on `site2` then update their Infinispan caches and the particular user session is also visible on Keycloak nodes on `site2`.

See the [Example Architecture Diagram](#) for more details.

### 3.4.8. Basic setup

For this example, we describe using two data centers, `site1` and `site2`. Each data center consists of 1 Infinispan server and 2 Keycloak servers. We will end up with 2 Infinispan servers and 4 Keycloak servers in total.

- `Site1` consists of Infinispan server, `jdg1`, and 2 KeyCloak servers, `node11` and `node12`.
- `Site2` consists of Infinispan server, `jdg2`, and 2 KeyCloak servers, `node21` and `node22`.
- Infinispan servers `jdg1` and `jdg2` are connected to each other through the RELAY2 protocol and `backup` based Infinispan caches in a similar way as described in the [JDG documentation](#).
- KeyCloak servers `node11` and `node12` form a cluster with each other, but they do not communicate directly with any server in `site2`. They communicate with the Infinispan server `jdg1` using the HotRod protocol (Remote cache). See [Communication details](#) for the details.
- The same details apply for `node21` and `node22`. They cluster with each other and communicate only with `jdg2` server using the HotRod protocol.

Our example setup assumes all that all 4 KeyCloak servers talk to the same database. In production, it is recommended to use separate synchronously replicated databases across data centers as described in [Database](#).

## Infinispan server setup

Follow these steps to set up the Infinispan server:

1. Download Infinispan {jdgserver\_version} server and unzip to a directory you choose. This location will be referred in later steps as `JDG1_HOME`.

2. Change those things in the `JDG1_HOME/standalone/configuration/clustered.xml` in the configuration of JGroups subsystem:
  - a. Add the `xsite` channel, which will use `tcp` stack, under `channels` element:

```
<channels default="cluster">
  <channel name="cluster"/>
  <channel name="xsite" stack="tcp"/>
</channels>
```

- b. Add a `relay` element to the end of the `udp` stack. We will configure it in a way that our site is `site1` and the other site, where we will backup, is `site2`:

```
<stack name="udp">
  ...
  <relay site="site1">
    <remote-site name="site2" channel="xsite"/>
    <property name="relay_multicast">false</property>
  </relay>
</stack>
```

- c. Configure the `tcp` stack to use `TCPPING` protocol instead of `MPING`. Remove the `MPING` element and replace it with the `TCPPING`. The `initial_hosts` element points to the hosts `jdg1` and `jdg2`:

```
<stack name="tcp">
  <transport type="TCP" socket-binding="jgroups-tcp"/>
  <protocol type="TCPPING">
    <property name="initi-
```

```
al_hosts">jdg1[7600],jdg2[7600]</property>
    <property name="ergonomics">false</property>
</protocol>
<protocol type="MERGE3"/>
...
</stack>
```

This is just an example setup to have things quickly running. In production, you are not required to use `tcp` stack for the JGroups `RE-LAY2`, but you can configure any other stack. For example, you could use the default `udp` stack, if the network between your data centers is able to support multicast. Just make sure that the Infinispan and KeyCloak clusters are mutually indiscernible. Similarly, you are not required to use `TCPPING` as discovery protocol. And in production, you probably won't use `TCPPING` due its static nature. Finally, site names are also configurable. Details of this more-detailed setup are out-of-scope of the KeyCloak documentation. See the Infinispan documentation and JGroups documentation for more details.

3. Add this into `JDG1_HOME/standalone/configuration/clustered.xml` under cache-container named `clustered`:

```
<cache-container name="clustered" default-cache="de-
fault" statistics="true">
...
```

```

<replicated-cache-configuration name="sessions-cfg" mode="SYNC" start="EAGER" batching="false">
    <locking acquire-timeout="0" />
    <backups>
        <backup site="site2" failure-policy="FAIL" strategy="SYNC" enabled="true">
            <take-offline min-wait="60000" after-failures="3" />
        </backup>
    </backups>
</replicated-cache-configuration>

<replicated-cache name="work" configuration="sessions-cfg"/>
    <replicated-cache name="sessions" configuration="sessions-cfg"/>
        <replicated-cache name="clientSessions" configuration="sessions-cfg"/>
        <replicated-cache name="offlineSessions" configuration="sessions-cfg"/>
        <replicated-cache name="offlineClientSessions" configuration="sessions-cfg"/>
        <replicated-cache name="actionTokens" configuration="sessions-cfg"/>
        <replicated-cache name="loginFailures" configuration="sessions-cfg"/>
    </replicated-cache>
</cache-container>
```

Details about the configuration options inside `replicated-cache-configuration` are explained in [Tuning the JDG cache configuration](#), which includes information about tweaking some of those options.

4. Some Infinispan server releases require authorization before accessing protected caches over network.

You should not see any issue if you use recommended Infinispan {jdgserver\_version} server and this step can (and should) be ignored. Issues related to authorization may exist just for some other versions of Infinispan server.

Keycloak requires updates to `__script_cache` cache containing scripts. If you get errors accessing this cache, you will need to set up authorization in `clustered.xml` configuration as described below:

- a. In the `<management>` section, add a security realm:

```
<management>
  <security-realms>
    ...
      <security-realm name="AllowScriptManager">
        <authentication>
          <users>
            <user username="__script_manager">
              <password>not-so-secret-password</password>
            </user>
          </users>
        </authentication>
      </security-realm>
    </security-realms>
```

- b. In the server core subsystem, add `<security>` as below:

```
<subsystem xmlns="urn:infinispan:server:core:8.4">
  <cache-container name="clustered" default-cache="default" statistics="true">
    <security>
      <authorization>
```

```

        <identity-role-mapper/>
        <role name="__script_manager" per-
missions="ALL"/>
    </authorization>
</security>
...

```

- c. In the endpoint subsystem, add authentication configuration to Hot Rod connector:

```

<subsystem xmlns="urn:infinispan:server:end-
point:8.1">
    <hotrod-connector cache-container="clustered"
socket-binding="hotrod">
    ...
        <authentication security-realm="AllowScript-
Manager">
            <sasl mechanisms="DIGEST-MD5" qop="auth"
server-name="keycloak-jdg-server">
                <policy>
                    <no-anonymous value="false" />
                </policy>
            </sasl>
        </authentication>

```

5. Copy the server to the second location, which will be referred to later as `JDG2_HOME`.
6. In the `JDG2_HOME/standalone/configuration/cluster.xml` exchange `site1` with `site2` and vice versa, both in the configuration of `relay` in the JGroups subsystem and in configuration of `backups` in the cache-subsystem. For example:
- a. The `relay` element should look like this:

```

<relay site="site2">
    <remote-site name="site1" channel="xsite"/>
    <property name="relay_multicasts">false</proper-

```

```
ty>
</relay>
```

- b. The `backups` element like this:

```
<backups>
  <backup site="site1" ....
  ...
  ...
```

It is currently required to have different configuration files for the JDG servers on both sites as the Infinispan subsystem does not support replacing site names with expressions. See [this issue](#) for more details.

7. Start server `jdg1`:

```
cd JDG1_HOME/bin
./standalone.sh -c clustered.xml -Djava.net.preferIP-
v4Stack=true \
-Djboss.default.multicast.address=234.56.78.99 \
-Djboss.node.name=jdg1 -b PUBLIC_IP_ADDRESS
```

8. Start server `jdg2`. There is a different multicast address, so the `jdg1` and `jdg2` servers are not directly clustered with each other; rather, they are just connected through the RELAY2 protocol, and the TCP JGroups stack is used for communication between them. The start up command looks like this:

```
cd JDG2_HOME/bin
./standalone.sh -c clustered.xml -Djava.net.preferIP-
v4Stack=true \
-Djboss.default.multicast.address=234.56.78.100 \
-Djboss.node.name=jdg2 -b PUBLIC_IP_ADDRESS
```

9. To verify that channel works at this point, you may need to use JConsole and connect either to the running JDG1 or the JDG2 server. When you use the MBean `jgroups:type=protocol,cluster="cluster",protocol=RELAY2` and operation `printRoutes`, you should see output like this:

```
site1 --> _jdg1:site1
site2 --> _jdg2:site2
```

When you use the MBean `jgroups:type=protocol,cluster="cluster",protocol=GMS`, you should see that the attribute member contains just single member:

- a. On JDG1 it should be like this:

```
(1) jdg1
```

- b. And on JDG2 like this:

```
(1) jdg2
```

In production, you can have more Infinispan servers in every data center. You just need to ensure that Infinispan servers in same data center are using the same multicast address (In other words, the same `jboss.default.multicast.address` during startup). Then in jconsole in GMS protocol view, you will see all the members of current cluster.

## KeyCloak servers setup

1. Unzip KeyCloak server distribution to a location you choose. It will be referred to later as `NODE11`.
2. Configure a shared database for KeycloakDS datasource. It is recommended to use MySQL or MariaDB for testing purposes. See [Database](#) for more details.

In production you will likely need to have a separate database server in every data center and both database servers should be synchronously replicated to each other. In the example setup, we just use a single database and connect all 4 KeyCloak servers to it.

3. Edit `NODE11/standalone/configuration/standalone-ha.xml` :
  - a. Add the attribute `site` to the JGroups UDP protocol:

```
<stack name="udp">
    <transport type="UDP" socket-
binding="jgroups-udp" site="${jboss.site.name}" />
```
  - b. Add this `module` attribute under `cache-container` element of name `keycloak` :

```
<cache-container name="keycloak" module="org.
keycloak.keycloak-model-infinispan">
```
  - c. Add the `remote-store` under `work` cache:

```
<replicated-cache name="work">
    <remote-store cache="work" remote-servers="remo-
te-cache" passivation="false" fetch-state="false"
purge="false" preload="false" shared="true">
```

```

        <property name="rawValues">true</property>
        <property name="marshaller">org.keycloak.
cluster.infinispan.KeycloakHotRodMarshallerFacto-
ry</property>
    </remote-store>
</replicated-cache>

```

- d. Add the `remote-store` like this under `sessions` cache:

```

<distributed-cache name="sessions" owners="1">
    <remote-store cache="sessions" remote-ser-
vers="remote-cache" passivation="false" fetch-sta-
te="false" purge="false" preload="false" sha-
red="true">
        <property name="rawValues">true</property>
        <property name="marshaller">org.keycloak.
cluster.infinispan.KeycloakHotRodMarshallerFacto-
ry</property>
    </remote-store>
</distributed-cache>

```

- e. Do the same for `offlineSessions`, `clientSessions`, `offlineClientSessions`, `loginFailures`, and `action-Tokens` caches (the only difference from `sessions` cache is that `cache` property value are different):

```

<distributed-cache name="offlineSessions" ow-
ners="1">
    <remote-store cache="offlineSessions" remote-
servers="remote-cache" passivation="false" fetch-
state="false" purge="false" preload="false" sha-
red="true">
        <property name="rawValues">true</property>
        <property name="marshaller">org.keycloak.
cluster.infinispan.KeycloakHotRodMarshallerFacto-
ry</property>
    </remote-store>
</distributed-cache>

```

```

<distributed-cache name="clientSessions" owners="1">
    <remote-store cache="clientSessions" remote-servers="remote-cache" passivation="false" fetch-state="false" purge="false" preload="false" shared="true">
        <property name="rawValues">true</property>
        <property name="marshaller">org.keycloak.cluster.infinispan.KeycloakHotRodMarshallerFactory</property>
    </remote-store>
</distributed-cache>

<distributed-cache name="offlineClientSessions" owners="1">
    <remote-store cache="offlineClientSessions" remote-servers="remote-cache" passivation="false" fetch-state="false" purge="false" preload="false" shared="true">
        <property name="rawValues">true</property>
        <property name="marshaller">org.keycloak.cluster.infinispan.KeycloakHotRodMarshallerFactory</property>
    </remote-store>
</distributed-cache>

<distributed-cache name="loginFailures" owners="1">
    <remote-store cache="loginFailures" remote-servers="remote-cache" passivation="false" fetch-state="false" purge="false" preload="false" shared="true">
        <property name="rawValues">true</property>
        <property name="marshaller">org.keycloak.cluster.infinispan.KeycloakHotRodMarshallerFactory</property>
    </remote-store>
</distributed-cache>

<distributed-cache name="actionTokens" owners="2">
    <object-memory size="-1"/>
    <expiration max-idle="-1" interval="300000"/>
    <remote-store cache="actionTokens" remote-servers="remote-cache" passivation="false" fetch-state="false" purge="false" preload="true" shared="true">

```

```
        <property name="rawValues">true</property>
        <property name="marshaller">org.keycloak.
cluster.infinispan.KeycloakHotRodMarshallerFacto-
ry</property>
    </remote-store>
</distributed-cache>
```

- f. Add outbound socket binding for the remote store into `socket-binding-group` element configuration:

```
<outbound-socket-binding name="remote-cache">
    <remote-destination host="${remote.cache.
host:localhost}" port="${remote.cache.port:11222}"/>
</outbound-socket-binding>
```

- g. The configuration of distributed cache `authenticationSessions` and other caches is left unchanged.
- h. Optionally enable DEBUG logging under the `logging` subsystem:

```
<logger category="org.keycloak.cluster.infinispan">
    <level name="DEBUG"/>
</logger>
<logger category="org.keycloak.connections.infinis-
pan">
    <level name="DEBUG"/>
</logger>
<logger category="org.keycloak.models.cache.infinis-
pan">
    <level name="DEBUG"/>
</logger>
<logger category="org.keycloak.models.sessions.
infinispan">
    <level name="DEBUG"/>
</logger>
```

4. Copy the `NODE11` to 3 other directories referred later as `NODE12`, `NODE21` and `NODE22`.

5. Start `NODE11` :

```
cd NODE11/bin  
.standalone.sh -c standalone-ha.xml -Djboss.node.  
name=node11 -Djboss.site.name=site1 \  
-Djboss.default.multicast.address=234.56.78.1 -Dremo-  
te.cache.host=jdg1 \  
-Djava.net.preferIPv4Stack=true -b PUBLIC_IP_ADDRESS
```

6. Start `NODE12` :

```
cd NODE12/bin  
.standalone.sh -c standalone-ha.xml -Djboss.node.  
name=node12 -Djboss.site.name=site1 \  
-Djboss.default.multicast.address=234.56.78.1 -Dremo-  
te.cache.host=jdg1 \  
-Djava.net.preferIPv4Stack=true -b PUBLIC_IP_ADDRESS
```

The cluster nodes should be connected. Something like this should be in the log of both `NODE11` and `NODE12`:

```
Received new cluster view for channel keycloak:  
[node11|1] (2) [node11, node12]
```

The channel name in the log might be different.

7. Start `NODE21` :

```
cd NODE21/bin  
.standalone.sh -c standalone-ha.xml -Djboss.node.  
name=node21 -Djboss.site.name=site2 \  
-Djboss.default.multicast.address=234.56.78.2 -Dremo-
```

```
te.cache.host=jdg2 \
-Djava.net.preferIPv4Stack=true -b PUBLIC_IP_ADDRESS
```

It shouldn't be connected to the cluster with `NODE11` and `NODE12`, but to separate one:

```
Received new cluster view for channel keycloak:
[node21|0] (1) [node21]
```

#### 8. Start `NODE22` :

```
cd NODE22/bin
./standalone.sh -c standalone-ha.xml -Djboss.node.name=node22 -Djboss.site.name=site2 \
-Djboss.default.multicast.address=234.56.78.2 -Dremote.cache.host=jdg2 \
-Djava.net.preferIPv4Stack=true -b PUBLIC_IP_ADDRESS
```

It should be in cluster with `NODE21` :

```
Received new cluster view for channel keycloak:
[node21|1] (2) [node21, node22]
```

The channel name in the log might be different.

#### 9. Test:

- a. Go to <http://node11:8080/auth/> and create the initial admin user.
- b. Go to <http://node11:8080/auth/admin> and login as admin to admin console.
- c. Open a second browser and go to any of nodes

<http://node12:8080/auth/admin> or  
<http://node21:8080/auth/admin> or  
<http://node22:8080/auth/admin>. After login, you should be able to see the same sessions in tab `Sessions` of particular user, client or realm on all 4 servers.

- d. After doing any change in Keycloak admin console (eg. update some user or some realm), the update should be immediately visible on any of 4 nodes as caches should be properly invalidated everywhere.
- e. Check server.logs if needed. After login or logout, the message like this should be on all the nodes `NODEXY/standalone/log/server.log` :

```
2017-08-25 17:35:17,737 DEBUG [org.keycloak.models.sessions.infinispan.remotestore.RemoteCacheSessionListener] (Client-Listener-sessions-30012a77422542f5) Received event from remote store. Event 'CLIENT_CACHE_ENTRY_REMOVED', key '193489e7-e2bc-4069-afe8-f1dfa73084ea', skip 'false'
```

### 3.4.9. Administration of Cross DC deployment

This section contains some tips and options related to Cross-Datacenter Replication.

- When you run the KeyCloak server inside a data center, it is required that the database referenced in `KeycloakDS` datasource is already running and available in that data center. It is also necessary that the Infinispan server referenced by the `outbound-socket-binding`, which is referenced from the Infinispan cache `remote-store` element, is already running. Otherwise the KeyCloak server

will fail to start.

- Every data center can have more database nodes if you want to support database failover and better reliability. Refer to the documentation of your database and JDBC driver for the details how to set this up on the database side and how the `KeycloakDS` datasource on Keycloak side needs to be configured.
- Every datacenter can have more Infinispan servers running in the cluster. This is useful if you want some failover and better fault tolerance. The HotRod protocol used for communication between Infinispan servers and KeyCloak servers has a feature that Infinispan servers will automatically send new topology to the KeyCloak servers about the change in the Infinispan cluster, so the remote store on KeyCloak side will know to which Infinispan servers it can connect. Read the Infinispan and WildFly documentation for more details.
- It is highly recommended that a master Infinispan server is running in every site before the KeyCloak servers in **any** site are started. As in our example, we started both `jdg1` and `jdg2` first, before all KeyCloak servers. If you still need to run the KeyCloak server and the backup site is offline, it is recommended to manually switch the backup site offline on the Infinispan servers on your site, as described in [Bringing sites offline and online](#). If you do not manually switch the unavailable site offline, the first startup may fail or they may be some exceptions during startup until the backup site is taken offline automatically due the configured count of failed operations.

### 3.4.10. Bringing sites offline and online

For example, assume this scenario:

1. Site `site2` is entirely offline from the `site1` perspective. This means that all Infinispan servers on `site2` are off **or** the network between `site1` and `site2` is broken.
2. You run Keycloak servers and Infinispan server `jdg1` in site `site1`
3. Someone logs in on a Keycloak server on `site1`.
4. The Keycloak server from `site1` will try to write the session to the remote cache on `jdg1` server, which is supposed to backup data to the `jdg2` server in the `site2`. See [Communication details](#) for more information.
5. Server `jdg2` is offline or unreachable from `jdg1`. So the backup from `jdg1` to `jdg2` will fail.
6. The exception is thrown in `jdg1` log and the failure will be propagated from `jdg1` server to Keycloak servers as well because the default `FAIL` backup failure policy is configured. See [Backup failure policy](#) for details around the backup policies.
7. The error will happen on Keycloak side too and user may not be able to finish his login.

According to your environment, it may be more or less probable that the network between sites is unavailable or temporarily broken (split-brain). In case this happens, it is good that Infinispan servers on `site1` are aware of the fact that Infinispan servers on `site2` are unavailable, so they will stop trying to reach the servers in the `jdg2` site and the backup failures won't happen. This is called `Take site offline`.

### *Take site offline*

There are 2 ways to take the site offline.

**Manually by admin** - Admin can use the `jconsole` or other tool and run some JMX operations to manually take the particular site offline. This is useful especially if the outage is planned. With `jconsole` or CLI, you can connect to the `jdg1` server and take the `site2` offline. More details about this are available in the [JDG documentation](#).

These steps usually need to be done for all the Keycloak caches mentioned in [SYNC or ASYNC backups](#).

**Automatically** - After some amount of failed backups, the `site2` will usually be taken offline automatically. This is done due the configuration of `take-offline` element inside the cache configuration as configured in [Infinispan server setup](#).

```
<take-offline min-wait="60000" after-failures="3" />
```

This example shows that the site will be taken offline automatically for the particular single cache if there are at least 3 subsequent failed backups and there is no any successful backup within 60 seconds.

Automatically taking a site offline is useful especially if the broken network between sites is unplanned. The disadvantage is that there will be some failed backups until the network outage is detected, which could also mean failures on the application side. For example, there will be failed logins for some users or big login timeouts. Especially if `failure-policy` with value `FAIL` is used.

The tracking of whether a site is offline is tracked separately for every cache.

### *Take site online*

Once your network is back and `site1` and `site2` can talk to each other, you may need to put the site online. This needs to be done manually through JMX or CLI in similar way as taking a site offline. Again, you may need to check all the caches and bring them online.

Once the sites are put online, it's usually good to:

- Do the [State transfer](#).
- Manually [Clear caches](#).

#### **3.4.11. State transfer**

State transfer is a required, manual step. Infinispan server does not do this automatically, for example during split-brain, it is only the admin who may decide which site has preference and hence if state transfer needs to be done bidirectionally between both sites or just unidirectionally, as in only from `site1` to `site2`, but not from `site2` to `site1`.

A bidirectional state transfer will ensure that entities which were created **after** split-brain on `site1` will be transferred to `site2`. This is not an issue as they do not yet exist on `site2`. Similarly, entities created **after** split-brain on `site2` will be transferred to `site1`. Possibly problematic parts are those entities which exist **before** split-brain on both sites and which were updated during split-brain on both sites. When this happens, one of the sites will **win** and will overwrite the updates done du-

ring split-brain by the second site.

Unfortunately, there is no any universal solution to this. Split-brains and network outages are just state, which is usually impossible to be handled 100% correctly with 100% consistent data between sites. In the case of KeyCloak, it typically is not a critical issue. In the worst case, users will need to re-login again to their clients, or have the improper count of loginFailures tracked for brute force protection. See the Infinispan/JGroups documentation for more tips how to deal with split-brain.

The state transfer can be also done on the Infinispan server side through JMX. The operation name is `pushState`. There are few other operations to monitor status, cancel push state, and so on. More info about state transfer is available in the [Infinispan docs](#).

### 3.4.12. Clear caches

After split-brain it is safe to manually clear caches in the KeyCloak admin console. This is because there might be some data changed in the database on `site1` and because of the event, that the cache should be invalidated wasn't transferred during split-brain to `site2`. Hence KeyCloak nodes on `site2` may still have some stale data in their caches.

To clear the caches, see `{adminguide_clearcache_link}[{adminguide_clearcache_name}]`.

When the network is back, it is sufficient to clear the cache just on one KeyCloak node on any random site. The cache invalidation event will be sent to all the other KeyCloak nodes in all sites. However, it needs to be done for all the caches (realms, users, keys). See [{adminguide\\_clearcache\\_name}](#) for more information.

### 3.4.13. Tuning the JDG cache configuration

This section contains tips and options for configuring your JDG cache.

#### *Backup failure policy*

By default, the configuration of backup `failure-policy` in the Infinispan cache configuration in the JDG `clustered.xml` file is configured as `FAIL`. You may change it to `WARN` or `IGNORE`, as you prefer.

The difference between `FAIL` and `WARN` is that when `FAIL` is used and the Infinispan server tries to back data up to the other site and the backup fails then the failure will be propagated back to the caller (the Keycloak server). The backup might fail because the second site is temporarily unreachable or there is a concurrent transaction which is trying to update same entity. In this case, the Keycloak server will then retry the operation a few times. However, if the retry fails, then the user might see the error after a longer timeout.

When using `WARN`, the failed backups are not propagated from the Infinispan server to the Keycloak server. The user won't see the error and the failed backup will be just ignored. There will be a shorter timeout, typically 10 seconds as that's the default timeout for backup. It can be changed by the attribute `timeout` of `backup` element. There won't be retries. There will just be a WARNING message in the Infinispan server log.

The potential issue is, that in some cases, there may be just some a short network outage between sites, where the retry (usage of the `FAIL` policy) may help, so with `WARN` (without retry), there will be some data inconsistencies across sites. This can also happen if there is an attempt to

update the same entity concurrently on both sites.

How bad are these inconsistencies? Usually only means that a user will need to re-authenticate.

When using the `WARN` policy, it may happen that the single-use cache, which is provided by the `actionTokens` cache and which handles that particular key is really single use, but may "successfully" write the same key twice. But, for example, the OAuth2 specification [mentions](#) that code must be single-use. With the `WARN` policy, this may not be strictly guaranteed and the same code could be written twice if there is an attempt to write it concurrently in both sites.

If there is a longer network outage or split-brain, then with both `FAIL` and `WARN`, the other site will be taken offline after some time and failures as described in [Bringing sites offline and online](#). With the default 1 minute timeout, it is usually 1-3 minutes until all the involved caches are taken offline. After that, all the operations will work fine from an end user perspective. You only need to manually restore the site when it is back online as mentioned in [Bringing sites offline and online](#).

In summary, if you expect frequent, longer outages between sites and it is acceptable for you to have some data inconsistencies and a not 100% accurate single-use cache, but you never want end-users to see the errors and long timeouts, then switch to `WARN`.

The difference between `WARN` and `IGNORE` is, that with `IGNORE` warnings are not written in the JDG log. See more details in the Infinispan documentation.

### *Lock acquisition timeout*

The default configuration is using transaction in NON\_DURABLE\_XA mode with acquire timeout 0. This means that transaction will fail-fast if there is another transaction in progress for the same key.

The reason to switch this to 0 instead of default 10 seconds was to avoid possible deadlock issues. With KeyCloak, it can happen that the same entity (typically session entity or loginFailure) is updated concurrently from both sites. This can cause deadlock under some circumstances, which will cause the transaction to be blocked for 10 seconds. See [this JIRA report](#) for details.

With timeout 0, the transaction will immediately fail and then will be retried from KeyCloak if backup `failure-policy` with the value `FAIL` is configured. As long as the second concurrent transaction is finished, the retry will usually be successful and the entity will have applied updates from both concurrent transactions.

We see very good consistency and results for concurrent transaction with this configuration, and it is recommended to keep it.

The only (non-functional) problem is the exception in the Infinispan server log, which happens every time when the lock is not immediately available.

#### **3.4.14. SYNC or ASYNC backups**

An important part of the `backup` element is the `strategy` attribute. You must decide whether it needs to be `SYNC` or `ASYNC`. We have 7 caches which might be Cross-Datacenter Replication aware, and these can be configured in 3 different modes regarding cross-dc:

1. SYNC backup
2. ASYNC backup
3. No backup at all

If the `SYNC` backup is used, then the backup is synchronous and operation is considered finished on the caller (Keycloak server) side once the backup is processed on the second site. This has worse performance than `ASYNC`, but on the other hand, you are sure that subsequent reads of the particular entity, such as user session, on `site2` will see the updates from `site1`. Also, it is needed if you want data consistency. As with `ASYNC` the caller is not notified at all if backup to the other site failed.

For some caches, it is even possible to not backup at all and completely skip writing data to the Infinispan server. To set this up, do not use the `remote-store` element for the particular cache on the Keycloak side (file `KEYCLOAK_HOME/standalone/configuration/standalone-ha.xml`) and then the particular `replicated-cache` element is also not needed on the Infinispan server side.

By default, all 7 caches are configured with `SYNC` backup, which is the safest option. Here are a few things to consider:

- If you are using active/passive mode (all Keycloak servers are in single site `site1` and the Infinispan server in `site2` is used purely as backup. See [Modes](#) for more details), then it is usually fine to use `ASYNC` strategy for all the caches to save the performance.
- The `work` cache is used mainly to send some messages, such as

cache invalidation events, to the other site. It is also used to ensure that some special events, such as userStorage synchronizations, happen only on single site. It is recommended to keep this set to `SYNC`.

- The `actionTokens` cache is used as single-use cache to track that some tokens/tickets were used just once. For example action tokens or OAuth2 codes. It is possible to set this to `ASYNC` to slightly improved performance, but then it is not guaranteed that particular ticket is really single-use. For example, if there is concurrent request for same ticket in both sites, then it is possible that both requests will be successful with the `ASYNC` strategy. So what you set here will depend on whether you prefer better security (`SYNC` strategy) or better performance (`ASYNC` strategy).
- The `loginFailures` cache may be used in any of the 3 modes. If there is no backup at all, it means that count of login failures for a user will be counted separately for every site (See [Infinispan caches](#) for details). This has some security implications, however it has some performance advantages. Also it mitigates the possible risk of denial of service (DoS) attacks. For example, if an attacker simulates 1000 concurrent requests using the username and password of the user on both sites, it will mean lots of messages being passed between the sites, which may result in network congestion. The `ASYNC` strategy might be even worse as the attacker requests won't be blocked by waiting for the backup to the other site, resulting in potentially even more congested network traffic. The count of login failures also will not be accurate with the `ASYNC` strategy.

For the environments with slower network between data centers and probability of DoS, it is recommended to not backup the `loginFailu-`

`res` cache at all.

- It is recommended to keep the `sessions` and `clientSessions` caches in `SYNC`. Switching them to `ASYNC` is possible only if you are sure that user requests and backchannel requests (requests from client applications to KeyCloak as described in [Request processing](#)) will be always processed on same site. This is true, for example, if:
  - You use active/passive mode as described [Modes](#).
  - All your client applications are using the KeyCloak [JavaScript Adapter](#). The JavaScript adapter sends the backchannel requests within the browser and hence they participate on the browser sticky session and will end on same cluster node (hence on same site) as the other browser requests of this user.
  - Your load balancer is able to serve the requests based on client IP address (location) and the client applications are deployed on both sites.

For example you have 2 sites LON and NYC. As long as your applications are deployed in both LON and NYC sites too, you can ensure that all the user requests from London users will be redirected to the applications in LON site and also to the KeyCloak servers in LON site. Backchannel requests from the LON site client deployments will end on KeyCloak servers in LON site too. On the other hand, for the American users, all the KeyCloak requests, application requests and backchannel requests will be processed on NYC site.

- For `offlineSessions` and `offlineClientSessions` it is similar, with the difference that you even don't need to backup them at

all if you never plan to use offline tokens for any of your client applications.

Generally, if you are in doubt and performance is not a blocker for you, it's safer to keep the caches in `SYNC` strategy.

Regarding the switch to SYNC/ASYNC backup, make sure that you edit the `strategy` attribute of the `backup` element. For example like this:

```
<backup site="site2" failure-policy="FAIL" strategy="ASYNC"  
enabled="true">
```

Note the `mode` attribute of cache-configuration element.

### 3.4.15. Troubleshooting

The following tips are intended to assist you should you need to troubleshoot:

- It is recommended to go through the [Basic setup](#) and have this one working first, so that you have some understanding of how things work. It is also wise to read this entire document to have some understanding of things.
- Check in jconsole cluster status (GMS) and the JGroups status (RELAY) of Infinispan as described in [Infinispan server setup](#). If things do not look as expected, then the issue is likely in the setup of Infinispan servers.
- For the KeyCloak servers, you should see a message like this during

the server startup:

```
18:09:30,156 INFO [org.keycloak.connections.infinispan.DefaultInfinispanConnectionProviderFactory] (ServerService Thread Pool -- 54)
Node name: node11, Site name: site1
```

Check that the site name and the node name looks as expected during the startup of KeyCloak server.

- Check that KeyCloak servers are in cluster as expected, including that only the KeyCloak servers from the same data center are in cluster with each other. This can be also checked in JConsole through the GMS view. See [cluster troubleshooting](#) for additional details.
- If there are exceptions during startup of KeyCloak server like this:

```
17:33:58,605 ERROR [org.infinispan.client.hotrod.impl.operations.RetryOnFailureOperation] (ServerService Thread Pool -- 59) ISPN004007: Exception encountered. Retry 10 out of 10: org.infinispan.client.hotrod.exceptions.TransportException:: Could not fetch transport
...
Caused by: org.infinispan.client.hotrod.exceptions.TransportException:: Could not connect to server: 127.0.0.1:12232
    at org.infinispan.client.hotrod.impl.transport.tcp.TcpTransport.<init>(TcpTransport.java:82)
```

it usually means that KeyCloak server is not able to reach the Infinispan server in his own datacenter. Make sure that firewall is set as expected and Infinispan server is possible to connect.

- If there are exceptions during startup of KeyCloak server like this:

```
16:44:18,321 WARN [org.infinispan.client.hotrod.impl.protocol.Codec21] (ServerService Thread Pool -- 57) IS-PN004005: Error received from the server: javax.transaction.RollbackException: ARJUNA016053: Could not commit transaction.  
...
```

then check the log of corresponding Infinispan server of your site and check if has failed to backup to the other site. If the backup site is unavailable, then it is recommended to switch it offline, so that Infinispan server won't try to backup to the offline site causing the operations to pass successfully on KeyCloak server side as well. See [Administration of Cross DC deployment](#) for more information.

- Check the Infinispan statistics, which are available through JMX. For example, try to login and then see if the new session was successfully written to both Infinispan servers and is available in the `sessions` cache there. This can be done indirectly by checking the count of elements in the `sessions` cache for the MBean `jboss.datagrid-infinispan:type=Cache, name="sessions(repl-sync)", manager="clustered", component=Statistics` and attribute `numberOfEntries`. After login, there should be one more entry for `numberOfEntries` on both Infinispan servers on both sites.
- Enable DEBUG logging as described [KeyCloak servers setup](#). For example, if you log in and you think that the new session is not available on the second site, it's good to check the KeyCloak server logs and check that listeners were triggered as described in the [KeyCloak servers setup](#). If you do not know and want to ask on keycloak-user mailing list, it is helpful to send the log files from KeyCloak servers on both datacenters in the email. Either add the log snippets

to the mails or put the logs somewhere and reference them in the email.

- If you updated the entity, such as `user`, on Keycloak server on `site1` and you do not see that entity updated on the Keycloak server on `site2`, then the issue can be either in the replication of the synchronous database itself or that Keycloak caches are not properly invalidated. You may try to temporarily disable the Keycloak caches as described [here](#) to nail down if the issue is at the database replication level. Also it may help to manually connect to the database and check if data are updated as expected. This is specific to every database, so you will need to consult the documentation for your database.
- Sometimes you may see the exceptions related to locks like this in Infinispan server log:

```
(HotRodServerHandler-6-35) ISPN000136: Error executing
command ReplaceCommand,
writing keys [[B0x033E243034396234..[39]]: org.
infinispan.util.concurrent.TimeoutException: ISPN000299:
Unable to acquire lock after
0 milliseconds for key [B0x033E243034396234..[39] and
requestor GlobalTx:jdg1:4353. Lock is held by Global-
Tx:jdg1:4352
```

Those exceptions are not necessarily an issue. They may happen anytime when a concurrent edit of the same entity is triggered on both DCs. This is common in a deployment. Usually the Keycloak server is notified about the failed operation and will retry it, so from the user's point of view, there is usually not any issue.

- If there are exceptions during startup of Keycloak server, like this:

```
16:44:18,321 WARN [org.infinispan.client.hotrod.impl.protocol.Codec21] (ServerService Thread Pool -- 55) IS-PN004005: Error received from the server: java.lang.SecurityException: ISP000287: Unauthorized access: subject 'Subject with principal(s): []' lacks 'READ' permission  
...
```

These log entries are the result of KeyCloak automatically detecting whether authentication is required on Infinispan and mean that authentication is necessary. At this point you will notice that either the server starts successfully and you can safely ignore these or that the server fails to start. If the server fails to start, ensure that Infinispan has been configured properly for authentication as described in [Infinispan server setup](#). To prevent this log entry from being included, you can force authentication by setting `remoteStoreSecurityEnabled` property to `true` in `spi=connectionsInfinispan/provider=default` configuration:

```
<subsystem xmlns="urn:jboss:domain:keycloak-server:1.1">  
  ...  
  <spi name="connectionsInfinispan">  
    ...  
    <provider name="default" enabled="true">  
      <properties>  
        ...  
        <property name="remoteStoreSecurityEnabled" value="true"/>  
      </properties>  
    </provider>  
  </spi>
```

- If you try to authenticate with KeyCloak to your application, but authentication fails with an infinite number of redirects in your browser.

ser and you see the errors like this in the KeyCloak server log:

```
2017-11-27 14:50:31,587 WARN [org.keycloak.events] (default task-17) type=LOGIN_ERROR, realmId=master, clientId=null, userId=null, ipAddress=aa.bb.cc.dd, error=expired_code, restart_after_timeout=true
```

it probably means that your load balancer needs to be set to support sticky sessions. Make sure that the provided route name used during startup of KeyCloak server (Property `jboss.node.name`) contains the correct name used by the load balancer server to identify the current server.

- If the Infinispan `work` cache grows indefinitely, you may be experiencing [this Infinispan issue](#), which is caused by cache items not being properly expired. In that case, update the cache declaration with an empty `<expiration />` tag like this:

```
<replicated-cache name="work" configuration="sessions-cfg">
    <expiration />
</replicated-cache>
```

- If you see Warnings in the Infinispan server log like:

```
18:06:19,687 WARN [org.infinispan.server.hotrod.Decoder2x] (HotRod-ServerWorker-7-12) ISPN006011: Operation 'PUT_IF_ABSENT' forced to
    return previous value should be used on transactional
    caches, otherwise data inconsistency issues could arise
    under failure situations
18:06:19,700 WARN [org.infinispan.server.hotrod.Decoder2x] (HotRod-ServerWorker-7-10) ISPN006010: Conditional operation 'REPLACE_IF_UNMODIFIED' should
    be used with transactional caches, otherwise data in-
    consistency issues could arise under failure situations
```

you can just ignore them. To avoid the warning, the caches on Infinispan server side could be changed to transactional caches, but this is not recommended as it can cause some other issues caused by the bug <https://issues.jboss.org/browse/ISPN-9323>. So for now, the warnings just need to be ignored.

- If you see errors in the Infinispan server log like:

```
12:08:32,921 ERROR [org.infinispan.server.hotrod.  
CacheDecodeContext] (HotRod-ServerWorker-7-11) IS-  
PN005003: Exception reported: org.infinispan.server.  
hotrod.InvalidMagicIdException: Error reading magic byte  
or message id: 7  
    at org.infinispan.server.hotrod.HotRodDecoder.  
readHeader(HotRodDecoder.java:184)  
    at org.infinispan.server.hotrod.HotRodDecoder.  
decodeHeader(HotRodDecoder.java:133)  
    at org.infinispan.server.hotrod.HotRodDecoder.  
decode(HotRodDecoder.java:92)  
    at io.netty.handler.codec.ByteToMessageDecoder.  
callDecode(ByteToMessageDecoder.java:411)  
    at io.netty.handler.codec.ByteToMessageDecoder.  
channelRead(ByteToMessageDecoder.java:248)
```

and you see some similar errors in the Keycloak log, it can indicate that there are incompatible versions of the HotRod protocol being used. This is likely happen when you try to use Keycloak with the JDG 7.2 server or an old version of the Infinispan server. It will help if you add the `protocolVersion` property as an additional property to the `remote-store` element in the Keycloak configuration file. For example:

```
<property name="protocolVersion">2.6</property>
```



## 4. Manage Subsystem Configuration

Low-level configuration of Keycloak is done by editing the `standalone.xml`, `standalone-ha.xml`, or `domain.xml` file in your distribution. The location of this file depends on your [operating mode](#).

While there are endless settings you can configure here, this section will focus on configuration of the *keycloak-server* subsystem. No matter which configuration file you are using, configuration of the *keycloak-server* subsystem is the same.

The *keycloak-server* subsystem is typically declared toward the end of the file like this:

```
<subsystem xmlns="urn:jboss:domain:keycloak-server:1.1">
    <web-context>auth</web-context>
    ...
</subsystem>
```

Note that anything changed in this subsystem will not take effect until the server is rebooted.

### 4.1. Configure SPI Providers

The specifics of each configuration setting is discussed elsewhere in context with that setting. However, it is useful to understand the format used to declare settings on SPI providers.

Keycloak is a highly modular system that allows great flexibility. There

are more than 50 service provider interfaces (SPIs), and you are allowed to swap out implementations of each SPI. An implementation of an SPI is known as a *provider*.

All elements in an SPI declaration are optional, but a full SPI declaration looks like this:

```
<spi name="myspi">
    <default-provider>myprovider</default-provider>
    <provider name="myprovider" enabled="true">
        <properties>
            <property name="foo" value="bar"/>
        </properties>
    </provider>
    <provider name="mysecondprovider" enabled="true">
        <properties>
            <property name="foo" value="foo"/>
        </properties>
    </provider>
</spi>
```

Here we have two providers defined for the SPI `myspi`. The `default-provider` is listed as `myprovider`. However it is up to the SPI to decide how it will treat this setting. Some SPIs allow more than one provider and some do not. So `default-provider` can help the SPI to choose.

Also notice that each provider defines its own set of configuration properties. The fact that both providers above have a property called `foo` is just a coincidence.

The type of each property value is interpreted by the provider. However, there is one exception. Consider the `jpa` provider for the `eventsSto-`

re SPI:

```
<spi name="eventsStore">
    <provider name="jpa" enabled="true">
        <properties>
            <property name="exclude-events" value=""
["EVENT1",
"EVENT2"]"/>
        </properties>
    </provider>
</spi>
```

We see that the value begins and ends with square brackets. That means that the value will be passed to the provider as a list. In this example, the system will pass the provider a list with two element values *EVENT1* and *EVENT2*. To add more values to the list, just separate each list element with a comma. Unfortunately, you do need to escape the quotes surrounding each list element with `"`.

## 4.2. Start the WildFly CLI

Besides editing the configuration by hand, you also have the option of changing the configuration by issuing commands via the *jboss-cli* tool. CLI allows you to configure servers locally or remotely. And it is especially useful when combined with scripting.

To start the WildFly CLI, you need to run `jboss-cli`.

### Linux/Unix

```
$ ./bin/jboss-cli.sh
```

## *Windows*

```
> ...\\bin\\jboss-cli.bat
```

This will bring you to a prompt like this:

## *Prompt*

```
[disconnected /]
```

If you wish to execute commands on a running server, you will first execute the `connect` command.

## *connect*

```
[disconnected /] connect  
connect  
[standalone@localhost:9990 /]
```

You may be thinking to yourself, "I didn't enter in any username or password!". If you run `jboss-cli` on the same machine as your running standalone server or domain controller and your account has appropriate file permissions, you do not have to setup or enter in an admin username and password. See the [`{appserver\_admindoc\_name}`](#) for more details on how to make things more secure if you are uncomfortable with that setup.

## 4.3. CLI Embedded Mode

If you do happen to be on the same machine as your standalone server and you want to issue commands while the server is not active, you can embed the server into CLI and make changes in a special mode that di-

sallows incoming requests. To do this, first execute the `embed-server` command with the config file you wish to change.

### *embed-server*

```
[disconnected /] embed-server --server-config=standalone.xml  
[standalone@embedded /]
```

## 4.4. CLI GUI Mode

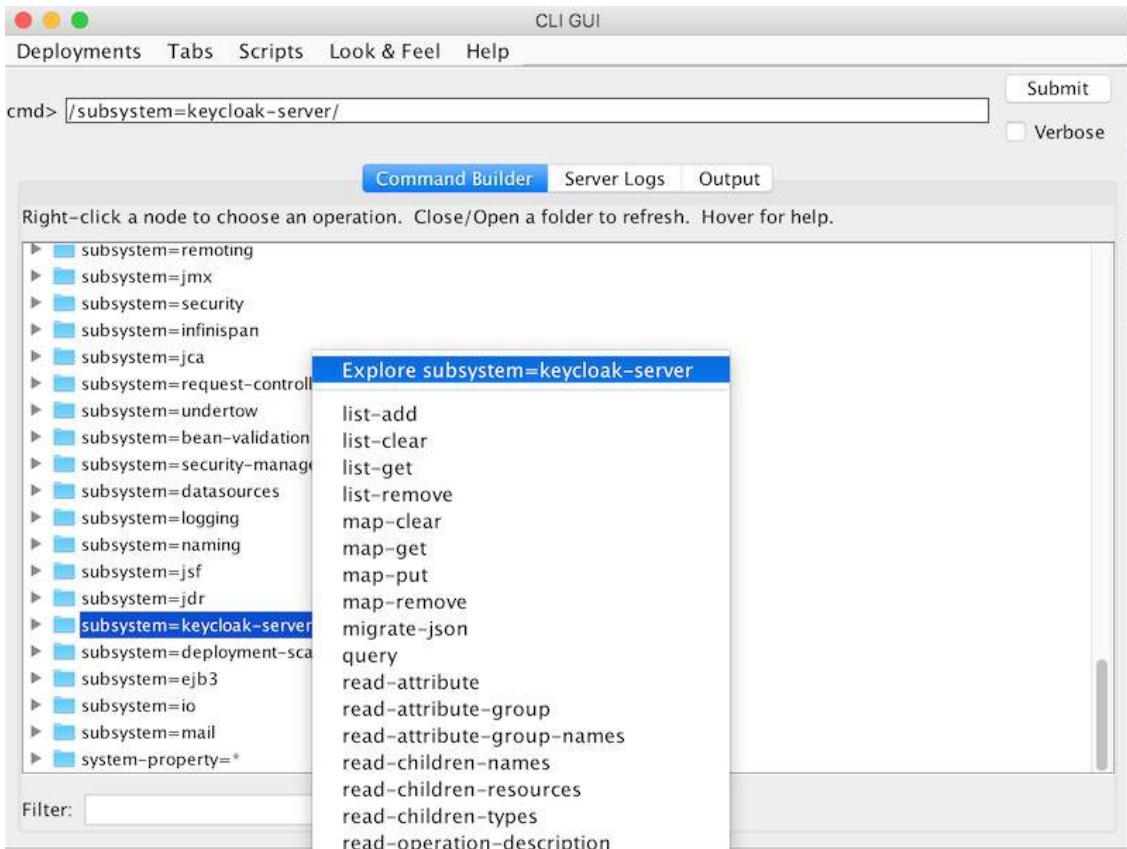
The CLI can also run in GUI mode. GUI mode launches a Swing application that allows you to graphically view and edit the entire management model of a *running* server. GUI mode is especially useful when you need help formatting your CLI commands and learning about the options available. The GUI can also retrieve server logs from a local or remote server.

### *Start in GUI mode*

```
$ .../bin/jboss-cli.sh --gui
```

*Note: to connect to a remote server, you pass the `--connect` option as well. Use the `--help` option for more details.*

After launching GUI mode, you will probably want to scroll down to find the node, `subsystem=keycloak-server`. If you right-click on the node and click `Explore subsystem=keycloak-server`, you will get a new tab that shows only the keycloak-server subsystem.



## 4.5. CLI Scripting

The CLI has extensive scripting capabilities. A script is just a text file with CLI commands in it. Consider a simple script that turns off theme and template caching.

### *turn-off-caching.cli*

```
/subsystem=keycloak-server/theme=defaults/:write-attribute(name=cacheThemes,value=false)
/subsystem=keycloak-server/theme=defaults/:write-attribute(name=cacheTemplates,value=false)
```

To execute the script, I can follow the `Scripts` menu in CLI GUI, or execute the script from the command line as follows:

```
$ ./bin/jboss-cli.sh --file=turn-off-caching.cli
```

## 4.6. CLI Recipes

Here are some configuration tasks and how to perform them with CLI commands. Note that in all but the first example, we use the wildcard path `**` to mean you should substitute or the path to the keycloak-server subsystem.

For standalone, this just means:

```
** = /subsystem=keycloak-server
```

For domain mode, this would mean something like:

```
** = /profile=auth-server-clustered/subsystem=keycloak-server
```

### 4.6.1. Change the web context of the server

```
/subsystem=keycloak-server/:write-attribute(name=web-context,value=myContext)
```

### 4.6.2. Set the global default theme

```
**/theme=defaults/:write-attribute(name=default,value=myTheme)
```

### 4.6.3. Add a new SPI and a provider

```
**/spi=mySPI/:add  
**/spi=mySPI/provider=myProvider/:add(enabled=true)
```

### 4.6.4. Disable a provider

```
**/spi=mySPI/provider=myProvider/:write-attribu-
```

```
te(name=enabled,value=false)
```

#### 4.6.5. Change the default provider for an SPI

```
**/spi=mySPI/:write-attribute(name=default-provider,value=myProvider)
```

#### 4.6.6. Configure the dblock SPI

```
**/spi=dblock/:add(default-provider=jpa)
**/spi=dblock/provider=jpa/:add(properties={lockWaitTimeout=> "900"},enabled=true)
```

#### 4.6.7. Add or change a single property value for a provider

```
**/spi=dblock/provider=jpa/:map-put(name=properties,key=lockWaitTimeout,value=3)
```

#### 4.6.8. Remove a single property from a provider

```
**/spi=dblock/provider=jpa/:map-remove(name=properties,key=lockRecheckTime)
```

#### 4.6.9. Set values on a provider property of type `List`

```
**/spi=eventsStore/provider=jpa/:map-put(name=properties,key=exclude-events,value=[EVENT1,EVENT2])
```

---

## 5. Profiles

There are features in Keycloak that are not enabled by default, these include features that are not fully supported. In addition there are some features that are enabled by default, but that can be disabled.

The features that can be enabled and disabled are:

Name	Description	Enabled by default	Support level
account_api	Account Management REST API	No	Preview
admin_fine_grained_authz	Fine-Grained Admin Permissions	No	Preview
authz_drools_policy	Drools Policy for Authorization Services	No	Preview
docker	Docker Registry protocol	No	Supported
impersonation	Ability for admins to impersonate users	Yes	Supported

	sonate users		
openshift_integration	Extension to enable securing OpenShift	No	Preview
scripts	Write custom authenticators using Java-Script	No	Preview
token_exchange	Token Exchange Service	No	Preview

To enable all preview features start the server with:

```
bin/standalone.sh|bat -Dkeycloak.profile=preview
```

You can set this permanently by creating the file `standalone/configuration/profile.properties` (or `domain/servers/server-one/configuration/profile.properties` for `server-one` in domain mode). Add the following to the file:

```
profile=preview
```

To enable a specific feature start the server with:

```
bin/standalone.sh|bat -Dkeycloak.profile.feature.<feature>
```

```
name>=enabled
```

For example to enable Docker use `-Dkeycloak.profile.feature.docker=enabled`.

You can set this permanently in the `profile.properties` file by adding:

```
feature.docker=enabled
```

To disable a specific feature start the server with:

```
bin/standalone.sh|bat -Dkeycloak.profile.feature.<feature name>=disabled
```

For example to disable Impersonation use `-Dkeycloak.profile.feature.impersonation=disabled`.

You can set this permanently in the `profile.properties` file by adding:

```
feature.impersonation=disabled
```

---

## 6. Relational Database Setup

KeyCloak comes with its own embedded Java-based relational database called H2. This is the default database that KeyCloak will use to persist data and really only exists so that you can run the authentication server out of the box. We highly recommend that you replace it with a more production ready external database. The H2 database is not very viable in high concurrency situations and should not be used in a cluster either. The purpose of this chapter is to show you how to connect KeyCloak to a more mature database.

KeyCloak uses two layered technologies to persist its relational data. The bottom layered technology is JDBC. JDBC is a Java API that is used to connect to a RDBMS. There are different JDBC drivers per database type that are provided by your database vendor. This chapter discusses how to configure KeyCloak to use one of these vendor-specific drivers.

The top layered technology for persistence is Hibernate JPA. This is a object to relational mapping API that maps Java Objects to relational data. Most deployments of KeyCloak will never have to touch the configuration aspects of Hibernate, but we will discuss how that is done if you run into that rare circumstance.

Datasource configuration is covered much more thoroughly in [the datasource configuration chapter](#) in the *{appserver\_admindoc\_name}*.

## 6.1. RDBMS Setup Checklist

These are the steps you will need to perform to get an RDBMS configured for Keycloak.

1. Locate and download a JDBC driver for your database
2. Package the driver JAR into a module and install this module into the server
3. Declare the JDBC driver in the configuration profile of the server
4. Modify the datasource configuration to use your database's JDBC driver
5. Modify the datasource configuration to define the connection parameters to your database

This chapter will use PostgreSQL for all its examples. Other databases follow the same steps for installation.

## 6.2. Package the JDBC Driver

Find and download the JDBC driver JAR for your RDBMS. Before you can use this driver, you must package it up into a module and install it into the server. Modules define JARs that are loaded into the Keycloak classpath and the dependencies those JARs have on other modules. They are pretty simple to set up.

Within the `../modules/` directory of your Keycloak distribution, you need to create a directory structure to hold your module definition. The convention is use the Java package name of the JDBC driver for the name of the directory structure. For PostgreSQL, create the directory

`org/postgresql/main`. Copy your database driver JAR into this directory and create an empty `module.xml` file within it too.

### Module Directory



After you have done this, open up the `module.xml` file and create the following XML:

### Module XML

```
<?xml version="1.0" ?>
<module xmlns="urn:jboss:module:1.3" name="org.postgresql">
```

```

<resources>
    <resource-root path="postgresql-9.4.1212.jar"/>
</resources>

<dependencies>
    <module name="javax.api"/>
    <module name="javax.transaction.api"/>
</dependencies>
</module>

```

The module name should match the directory structure of your module. So, `org/postgresql` maps to `org.postgresql`. The `resource-root path` attribute should specify the JAR filename of the driver. The rest are just the normal dependencies that any JDBC driver JAR would have.

### 6.3. Declare and Load JDBC Driver

The next thing you have to do is declare your newly packaged JDBC driver into your deployment profile so that it loads and becomes available when the server boots up. Where you perform this action depends on your [operating mode](#). If you're deploying in standard mode, edit `.../standalone/configuration/standalone.xml`. If you're deploying in standard clustering mode, edit `.../standalone/configuration/standalone-ha.xml`. If you're deploying in domain mode, edit `.../domain/configuration/domain.xml`. In domain mode, you'll need to make sure you edit the profile you are using: either `auth-server-standalone` or `auth-server-clustered`

Within the profile, search for the `drivers` XML block within the `datasources` subsystem. You should see a pre-defined driver declared for the H2 JDBC driver. This is where you'll declare the JDBC driver for your external database.

## JDBC Drivers

```
<subsystem xmlns="{subsystem.datasources_xml_urn}">
  <datasources>
    ...
    <drivers>
      <driver name="h2" module="com.h2database.h2">
        <xa-datasource-class>org.h2.jdbcx.JdbcData-
Source</xa-datasource-class>
      </driver>
    </drivers>
  </datasources>
</subsystem>
```

Within the `drivers` XML block you'll need to declare an additional JDBC driver. It needs to have a `name` which you can choose to be anything you want. You specify the `module` attribute which points to the `module` package you created earlier for the driver JAR. Finally you have to specify the driver's Java class. Here's an example of installing PostgreSQL driver that lives in the module example defined earlier in this chapter.

## Declare Your JDBC Drivers

```
<subsystem xmlns="{subsystem.datasources_xml_urn}">
  <datasources>
    ...
    <drivers>
      <driver name="postgresql" module="org.postgres-
ql">
        <xa-datasource-class>org.postgresql.xa.
PGXADataSource</xa-datasource-class>
      </driver>
      <driver name="h2" module="com.h2database.h2">
        <xa-datasource-class>org.h2.jdbcx.JdbcData-
Source</xa-datasource-class>
      </driver>
    </drivers>
```

```
</datasources>
</subsystem>
```

## 6.4. Modify the Keycloak Datasource

After declaring your JDBC driver, you have to modify the existing datasource configuration that Keycloak uses to connect it to your new external database. You'll do this within the same configuration file and XML block that you registered your JDBC driver in. Here's an example that sets up the connection to your new database:

### *Declare Your JDBC Drivers*

```
<subsystem xmlns="{subsystem.datasources_xml_urn}">
    <datasources>
        ...
        <datasource jndi-name="java:jboss/datasources/Key-
cloakDS" pool-name="KeycloakDS" enabled="true" use-java-
context="true">
            <connection-url>jdbc:postgresql://localhost/key-
cloak</connection-url>
            <driver>postgresql</driver>
            <pool>
                <max-pool-size>20</max-pool-size>
            </pool>
            <security>
                <user-name>William</user-name>
                <password>password</password>
            </security>
        </datasource>
        ...
    </datasources>
</subsystem>
```

Search for the `datasource` definition for `KeycloakDS`. You'll first need to modify the `connection-url`. The documentation for your vendor's JDBC implementation should specify the format for this con-

nnection URL value.

Next define the `driver` you will use. This is the logical name of the JDBC driver you declared in the previous section of this chapter.

It is expensive to open a new connection to a database every time you want to perform a transaction. To compensate, the datasource implementation maintains a pool of open connections. The `max-pool-size` specifies the maximum number of connections it will pool. You may want to change the value of this depending on the load of your system.

Finally, with PostgreSQL at least, you need to define the database user-name and password that is needed to connect to the database. You may be worried that this is in clear text in the example. There are methods to obfuscate this, but this is beyond the scope of this guide.

For more information about datasource features, see [the datasource configuration chapter](#) in the `{appserver_admindoc_name}`.

## 6.5. Database Configuration

The configuration for this component is found in the `standalone.xml`, `standalone-ha.xml`, or `domain.xml` file in your distribution. The location of this file depends on your [operating mode](#).

### Database Config

```
<subsystem xmlns="urn:jboss:domain:keycloak-server:1.1">
  ...
  <spi name="connectionsJpa">
    <provider name="default" enabled="true">
```

```

<properties>
    <property name="dataSource"
value="java:jboss/datasources/KeycloakDS"/>
    <property name="initializeEmpty"
value="false"/>
    <property name="migrationStrategy" value="ma-
nual"/>
    <property name="migrationExport"
value="${jboss.home.dir}/keycloak-database-update.sql"/>
</properties>
</provider>
</spi>
...
</subsystem>

```

Possible configuration options are:

### **dataSource**

JNDI name of the dataSource

### **jta**

boolean property to specify if datasource is JTA capable

### **driverDialect**

Value of database dialect. In most cases you don't need to specify this property as dialect will be autodetected by Hibernate.

### **initializeEmpty**

Initialize database if empty. If set to false the database has to be manually initialized. If you want to manually initialize the database set migrationStrategy to `manual` which will create a file with SQL commands to initialize the database. Defaults to true.

## **migrationStrategy**

Strategy to use to migrate database. Valid values are `update`, `manual` and `validate`. Update will automatically migrate the database schema. Manual will export the required changes to a file with SQL commands that you can manually execute on the database. Validate will simply check if the database is up-to-date.

## **migrationExport**

Path for where to write manual database initialization/migration file.

## **showSql**

Specify whether Hibernate should show all SQL commands in the console (false by default). This is very verbose!

## **formatSql**

Specify whether Hibernate should format SQL commands (true by default)

## **globalStatsInterval**

Will log global statistics from Hibernate about executed DB queries and other things. Statistics are always reported to server log at specified interval (in seconds) and are cleared after each report.

## **schema**

Specify the database schema to use

These configuration switches and more are described in the [\*{appserver\\_jpa\\_name}\*](#).

## 6.6. Unicode Considerations for Databases

Database schema in KeyCloak only accounts for Unicode strings in the following special fields:

- Realms: display name, HTML display name
- Federation Providers: display name
- Users: username, given name, last name, attribute names and values
- Groups: name, attribute names and values
- Roles: name
- Descriptions of objects

Otherwise, characters are limited to those contained in database encoding which is often 8-bit. However, for some database systems, it is possible to enable UTF-8 encoding of Unicode characters and use full Unicode character set in all text fields. Often, this is counterbalanced by shorter maximum length of the strings than in case of 8-bit encodings.

Some of the databases require special settings to database and/or JDBC driver to be able to handle Unicode characters. Please find the settings for your database below. Note that if a database is listed here, it can still work properly provided it handles UTF-8 encoding properly both on the level of database and JDBC driver.

Technically, the key criterion for Unicode support for all fields is whether the database allows setting of Unicode character set for `VARCHAR` and `CHAR` fields. If yes, there is a high chance that Unicode will be plausible, usually at the expense of field length. If it only supports Uni-

code in `NVARCHAR` and `NCHAR` fields, Unicode support for all text fields is unlikely as Keycloak schema uses `VARCHAR` and `CHAR` fields extensively.

### 6.6.1. Oracle Database

Unicode characters are properly handled provided the database was created with Unicode support in `VARCHAR` and `CHAR` fields (e.g. by using `AL32UTF8` character set as the database character set). No special settings is needed for JDBC driver.

If the database character set is not Unicode, then to use Unicode characters in the special fields, the JDBC driver needs to be configured with the connection property `oracle.jdbc.defaultNChar` set to `true`. It might be wise, though not strictly necessary, to also set the `oracle.jdbc.convertNcharLiterals` connection property to `true`. These properties can be set either as system properties or as connection properties. Please note that setting `oracle.jdbc.defaultNChar` may have negative impact on performance. For details, please refer to Oracle JDBC driver configuration documentation.

### 6.6.2. Microsoft SQL Server Database

Unicode characters are properly handled only for the special fields. No special settings of JDBC driver or database is necessary.

### 6.6.3. MySQL Database

Unicode characters are properly handled provided the database was created with Unicode support in `VARCHAR` and `CHAR` fields in the `CREATE DATABASE` command (e.g. by using `utf8` character set as the default database character set in MySQL 5.5. Please note that `utf8mb4`

character set does not work due to different storage requirements to `utf8` character set [1]). Note that in this case, length restriction to non-special fields does not apply because columns are created to accommodate given amount of characters, not bytes. If the database default character set does not allow storing Unicode, only the special fields allow storing Unicode values.

At the side of JDBC driver settings, it is necessary to add a connection property `characterEncoding=UTF-8` to the JDBC connection settings.

#### 6.6.4. PostgreSQL Database

Unicode is supported when the database character set is `UTF8`. In that case, Unicode characters can be used in any field, there is no reduction of field length for non-special fields. No special settings of JDBC driver is necessary.

---

## 7. Network Setup

KeyCloak can run out of the box with some networking limitations. For one, all network endpoints bind to `localhost` so the auth server is really only usable on one local machine. For HTTP based connections, it does not use default ports like 80 and 443. HTTPS/SSL is not configured out of the box and without it, KeyCloak has many security vulnerabilities. Finally, KeyCloak may often need to make secure SSL and HTTPS connections to external servers and thus need a trust store set up so that endpoints can be validated correctly. This chapter discusses all of these things.

### 7.1. Bind Addresses

By default KeyCloak binds to the localhost loopback address `127.0.1`. That's not a very useful default if you want the authentication server available on your network. Generally, what we recommend is that you deploy a reverse proxy or load balancer on a public network and route traffic to individual KeyCloak server instances on a private network. In either case though, you still need to set up your network interfaces to bind to something other than `localhost`.

Setting the bind address is quite easy and can be done on the command line with either the `standalone.sh` or `domain.sh` boot scripts discussed in the [Choosing an Operating Mode](#) chapter.

```
$ standalone.sh -b 192.168.0.5
```

The `-b` switch sets the IP bind address for any public interfaces.

Alternatively, if you don't want to set the bind address at the command line, you can edit the profile configuration of your deployment. Open up the profile configuration file (*standalone.xml* or *domain.xml* depending on your [operating mode](#)) and look for the `interfaces` XML block.

```
<interfaces>
    <interface name="management">
        <inet-address value="${jboss.bind.address.
management:127.0.0.1}" />
    </interface>
    <interface name="public">
        <inet-address value="${jboss.bind.address:127.
0.0.1}" />
    </interface>
</interfaces>
```

The `public` interface corresponds to subsystems creating sockets that are available publicly. An example of one of these subsystems is the web layer which serves up the authentication endpoints of Keycloak. The `management` interface corresponds to sockets opened up by the management layer of the WildFly. Specifically the sockets which allow you to use the `jboss-cli.sh` command line interface and the WildFly web console.

In looking at the `public` interface you see that it has a special string `${jboss.bind.address:127.0.0.1}`. This string denotes a value `127.0.0.1` that can be overridden on the command line by setting a Java system property, i.e.:

```
$ domain.sh -Djboss.bind.address=192.168.0.5
```

The `-b` is just a shorthand notation for this command. So, you can either change the bind address value directly in the profile config, or change it on the command line when you boot up.

There are many more options available when setting up `interface` definitions. For more information, see [the network interface](#) in the `{appserver_network_name}`.

## 7.2. Socket Port Bindings

The ports opened for each socket have a pre-defined default that can be overridden at the command line or within configuration. To illustrate this configuration, let's pretend you are running in [standalone mode](#) and open up the `../standalone/configuration/standalone.xml`. Search for `socket-binding-group`.

```
<socket-binding-group name="standard-sockets" default-interface="public" port-offset="#{jboss.socket.binding.port-offset:0}">
    <socket-binding name="management-http" interface="management" port="#{jboss.management.http.port:9990}" />
    <socket-binding name="management-https" interface="management" port="#{jboss.management.https.port:9993}" />
    <socket-binding name="ajp" port="#{jboss.ajp.port:8009}" />
    <socket-binding name="http" port="#{jboss.http.port:8080}" />
    <socket-binding name="https" port="#{jboss.https.port:8443}" />
    <socket-binding name="txn-recovery-environment" port="4712" />
    <socket-binding name="txn-status-manager" port="4713" />
    <outbound-socket-binding name="mail-smtp">
```

```
<remote-destination host="localhost"  
port="25"/>  
    </outbound-socket-binding>  
</socket-binding-group>
```

`socket-bindings` define socket connections that will be opened by the server. These bindings specify the `interface` (bind address) they use as well as what port number they will open. The ones you will be most interested in are:

### **http**

Defines the port used for KeyCloak HTTP connections

### **https**

Defines the port used for KeyCloak HTTPS connections

### **ajp**

This socket binding defines the port used for the AJP protocol. This protocol is used by Apache HTTPD server in conjunction `mod-cluster` when you are using Apache HTTPD as a load balancer.

### **management-http**

Defines the HTTP connection used by WildFly CLI and web console.

When running in [domain mode](#) setting the socket configurations is a bit trickier as the example `domain.xml` file has multiple `socket-binding-groups` defined. If you scroll down to the `server-group` definitions you can see what `socket-binding-group` is used for each `server-group`.

## *domain socket bindings*

```
<server-groups>
    <server-group name="load-balancer-group" profile="load-balancer">
        ...
        <socket-binding-group ref="load-balancer-sockets"/>
    </server-group>
    <server-group name="auth-server-group" profile="auth-server-clustered">
        ...
        <socket-binding-group ref="ha-sockets"/>
    </server-group>
</server-groups>
```

There are many more options available when setting up `socket-binding-group` definitions. For more information, see [the socket binding group](#) in the `{appserver\_socket_name}`.

## 7.3. Setting up HTTPS/SSL

KeyCloak is not set up by default to handle SSL/HTTPS. It is highly recommended that you either enable SSL on the KeyCloak server itself or on a reverse proxy in front of the KeyCloak server.

This default behavior is defined by the SSL/HTTPS mode of each KeyCloak realm. This is discussed in more detail in the [{adminguide!}](#), but let's give some context and a brief overview of these modes.

### **external requests**

Keycloak can run out of the box without SSL so long as you stick to private IP addresses like `localhost`, `127.0.0.1`, `10.0.x.x`, `192.168.x.x`, and `172.16.x.x`. If you don't have SSL/HTTPS configured on the server or you try to access Keycloak over HTTP from a non-private IP address you will get an error.

### **none**

Keycloak does not require SSL. This should really only be used in development when you are playing around with things.

### **all requests**

Keycloak requires SSL for all IP addresses.

The SSL mode for each realm can be configured in the Keycloak admin console.

#### **7.3.1. Enabling SSL/HTTPS for the Keycloak Server**

If you are not using a reverse proxy or load balancer to handle HTTPS traffic for you, you'll need to enable HTTPS for the Keycloak server. This involves

1. Obtaining or generating a keystore that contains the private key and certificate for SSL/HTTP traffic
2. Configuring the Keycloak server to use this keypair and certificate.

#### **Creating the Certificate and Java Keystore**

In order to allow HTTPS connections, you need to obtain a self signed or third-party signed certificate and import it into a Java keystore before you can enable HTTPS in the web container you are deploying the Key-

Cloak Server to.

## Self Signed Certificate

In development, you will probably not have a third party signed certificate available to test a KeyCloak deployment so you'll need to generate a self-signed one using the `keytool` utility that comes with the Java JDK.

```
$ keytool -genkey -alias localhost -keyalg RSA -keystore  
keycloak.jks -validity 10950  
Enter keystore password: secret  
Re-enter new password: secret  
What is your first and last name?  
[Unknown]: localhost  
What is the name of your organizational unit?  
[Unknown]: Keycloak  
What is the name of your organization?  
[Unknown]: Red Hat  
What is the name of your City or Locality?  
[Unknown]: Westford  
What is the name of your State or Province?  
[Unknown]: MA  
What is the two-letter country code for this unit?  
[Unknown]: US  
Is CN=localhost, OU=Keycloak, O=Test, L=Westford,  
ST=MA, C=US correct?  
[no]: yes
```

You should answer `What is your first and last name ?` question with the DNS name of the machine you're installing the server on. For testing purposes, `localhost` should be used. After executing this command, the `keycloak.jks` file will be generated in the same directory as you executed the `keytool` command in.

If you want a third-party signed certificate, but don't have one, you can

obtain one for free at [cacert.org](http://cacert.org). You'll have to do a little set up first before doing this though.

The first thing to do is generate a Certificate Request:

```
$ keytool -certreq -alias yourdomain -keystore keycloak.jks  
> keycloak.careq
```

Where `yourdomain` is a DNS name for which this certificate is generated for. Keytool generates the request:

```
-----BEGIN NEW CERTIFICATE REQUEST-----  
MIIC2jCCAcICAQAwZTELMAKGA1UEBhMCVVMxCzAJBgNVBAgTAK1BMREwDw-  
YDVQQHEwhXZXN0Zm9y  
ZDEQMA4GA1UEChMHUmVKIEhhdDEQMA4GA1UECxMHUmVKIEhhdDESBAGA1-  
UEAxMJbG9jYWxob3N0  
MIIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIIBCgKCAQEAr7kck2Taav1EOG-  
bcpi9c0rncY4HhdzmY  
Ax2nZfq1eZEaIPqI5aTxwQZzzLDK9qbeAd8Ji79HzSqnRDxNYaZu7mAYhF-  
KHgixsolE3o5Yfzbw1  
29RvyeUVe+WZxv5oo9wolVVpdSINIMEL2LaFhtX/c1dqiqYVpfnvFshZ-  
QaIg2nL8juzZcBjj4as  
H98gIS7khql/dkZKsw9NLvyxgJvp7PaXurX29fNf3ihG+oFrL22oFy-  
V54BWxXCKU/GPn61EGZGw  
Ft2qsIGLdctpMD1aJR2bcn1hEjZKDksjQZoQ5YMXaAGkcYkG6QkgrocDE2-  
YXDbi7GIdf9MegVJ35  
2DQMpwIDAQABoAwLgYJKoZIhvcNAQkOMSEwHzAdBgNVHQ4EFgQUQwlZJ-  
BA+fjiDdiVza09vrE/i  
n2swDQYJKoZIhvcNAQELBQADggEBAC5FRvMkhal3q86tHPBYWBuTtmcSjs-  
4qUm6V6f63frhveWhf  
PzRRI1xH272XUIeBk0gtzWo0nNZnf0mMCTUBbHhhDcG82xolikfqibZijo-  
QZCiGiedVjHJFtniDQ  
9bMDUOXEMQ7gHZg5q6mJfNG9MbMpQaUVEEFvfGEQQxbiFK7hRWU8S-  
23/d80e8nExgQxdJWJ6vd0X  
MzzFK6j4Dj55bJVuM7GFmfNC52pNOD5vYe47Aqh8oajHX9XTycVtPX145-  
rrWAH33ftbrs8SrZ2S  
vqIFQeuLL3BaHwp13t7j21MwcK1p80laAxEASib/fAwrRHpLHBXRc-  
q6uALU0Z14Alt8=
```

```
-----END NEW CERTIFICATE REQUEST-----
```

Send this ca request to your CA. The CA will issue you a signed certificate and send it to you. Before you import your new cert, you must obtain and import the root certificate of the CA. You can download the cert from CA (ie.: root.crt) and import as follows:

```
$ keytool -import -keystore keycloak.jks -file root.crt -alias root
```

Last step is to import your new CA generated certificate to your keystore:

```
$ keytool -import -alias yourdomain -keystore keycloak.jks -file your-certificate.cer
```

## Configure KeyCloak to Use the Keystore

Now that you have a Java keystore with the appropriate certificates, you need to configure your KeyCloak installation to use it. First, you must edit the *standalone.xml*, *standalone-ha.xml*, or *host.xml* file to use the keystore and enable HTTPS. You may then either move the keystore file to the *configuration/* directory of your deployment or the file in a location you choose and provide an absolute path to it. If you are using absolute paths, remove the optional `relative-to` parameter from your configuration (See [operating mode](#)).

Add the new `security-realm` element using the CLI:

```
$ /core-service=management/security-realm=UndertowRealm:add()
```

```
$ /core-service=management/security-realm=UndertowRe-  
alm/server-identity=ssl:add(keystore-path=keycloak.jks,  
keystore-relative-to=jboss.server.config.dir, keystore-  
password=secret)
```

If using domain mode, the commands should be executed in every host using the `/host=<host_name>/` prefix (in order to create the `security-realm` in all of them), like this, which you would repeat for each host:

```
$ /host=<host_name>/core-service=management/security-re-  
alm=UndertowRealm/server-identity=ssl:add(keystore-  
path=keycloak.jks, keystore-relative-to=jboss.server.  
config.dir, keystore-password=secret)
```

In the standalone or host configuration file, the `security-realms` element should look like this:

```
<security-realm name="UndertowRealm">  
    <server-identities>  
        <ssl>  
            <keystore path="keycloak.jks" relative-  
to="jboss.server.config.dir" keystore-password="secret" />  
        </ssl>  
    </server-identities>  
</security-realm>
```

Next, in the standalone or each domain configuration file, search for any instances of `security-realm`. Modify the `https-listener` to use the created realm:

```
$ /subsystem=undertow/server=default-server/httpslis-  
tener=https:write-attribute(name=security-realm, value=Un-
```

```
derToRealm)
```

If using domain mode, prefix the command with the profile that is being used with: `/profile=<profile_name>/`.

The resulting element, `server name="default-server"`, which is a child element of `subsystem xmlns="{subsystem_undertow_xml_urn}"`, should contain the following stanza:

```
<subsystem xmlns="{subsystem_undertow_xml_urn}">
    <buffer-cache name="default"/>
    <server name="default-server">
        <https-listener name="https" socket-binding="https"
security-realm="UndertowRealm"/>
        ...
    </server>
</subsystem>
```

## 7.4. Outgoing HTTP Requests

The Keycloak server often needs to make non-browser HTTP requests to the applications and services it secures. The auth server manages these outgoing connections by maintaining an HTTP client connection pool. There are some things you'll need to configure in `standalone.xml`, `standalone-ha.xml`, or `domain.xml`. The location of this file depends on your [operating mode](#).

### *HTTP client Config example*

```
<spi name="connectionsHttpClient">
    <provider name="default" enabled="true">
        <properties>
            <property name="connection-pool-size"
value="256"/>
        </properties>
    </provider>
```

```
</spi>
```

Possible configuration options are:

### **establish-connection-timeout-millis**

Timeout for establishing a socket connection.

### **socket-timeout-millis**

If an outgoing request does not receive data for this amount of time, timeout the connection.

### **connection-pool-size**

How many connections can be in the pool (128 by default).

### **max-pooled-per-route**

How many connections can be pooled per host (64 by default).

### **connection-ttl-millis**

Maximum connection time to live in milliseconds. Not set by default.

### **max-connection-idle-time-millis**

Maximum time the connection might stay idle in the connection pool (900 seconds by default). Will start background cleaner thread of Apache HTTP client. Set to `-1` to disable this checking and the background thread.

### **disable-cookies**

`true` by default. When set to true, this will disable any cookie

caching.

### **client-keystore**

This is the file path to a Java keystore file. This keystore contains client certificate for two-way SSL.

### **client-keystore-password**

Password for the client keystore. This is *REQUIRED* if `client-keystore` is set.

### **client-key-password**

Password for the client's key. This is *REQUIRED* if `client-keystore` is set.

### **proxy-mappings**

Denotes proxy configurations for outgoing HTTP requests. See the section on [Proxy Mappings for Outgoing HTTP Requests](#) for more details.

#### **7.4.1. Proxy Mappings for Outgoing HTTP Requests**

Outgoing HTTP requests sent by Keycloak can optionally use a proxy server based on a comma delimited list of proxy-mappings. A proxy-mapping denotes the combination of a regex based hostname pattern and a proxy-uri in the form of `hostnamePattern;proxyUri`, e.g.:

```
.*\.(google|googleapis)\.com;http://www-proxy.acme.com:8080
```

To determine the proxy for an outgoing HTTP request the target hostname is matched against the configured hostname patterns. The first

matching pattern determines the proxy-uri to use. If none of the configured patterns match for the given hostname then no proxy is used.

The special value `NO_PROXY` for the proxy-uri can be used to indicate that no proxy should be used for hosts matching the associated hostname pattern. It is possible to specify a catch-all pattern at the end of the proxy-mappings to define a default proxy for all outgoing requests.

The following example demonstrates the proxy-mapping configuration.

```
# All requests to Google APIs should use http://www-proxy.acme.com:8080 as proxy
.*\.(google|googleapis)\.com;http://www-proxy.acme.com:8080

# All requests to internal systems should use no proxy
.*\.\.acme\.com;NO_PROXY

# All other requests should use http://fallback:8080 as proxy
.*;http://fallback:8080
```

This can be configured via the following `jboss-cli` command. Note that you need to properly escape the regex-pattern as shown below.

```
echo SETUP: Configure proxy routes for HttpClient SPI

# In case there is no connectionsHttpClient definition yet
/subsystem=keycloak-server/spi=connectionsHttpClient/provider=default:add(enabled=true)

# Configure the proxy-mappings
/subsystem=keycloak-server/spi=connectionsHttpClient/provider=default:write-attribute(name=properties.proxy-mappings,value=[".*\\".(google|googleapis)\\.com;http://www-proxy.acme.com:8080",".*\\.\.acme\\.com;NO_PROXY",".*;http://fallback:8080"])
```

The `jboss-cli` command results in the following subsystem configuration. Note that one needs to encode " characters with &quot; .

```
<spi name="connectionsHttpClient">
  <provider name="default" enabled="true">
    <properties>
      <property
        name="proxy-mappings"
        value="[".*\.(google|googleapis)\\.com;http://www-proxy.acme.com:8080",".*\\.acme\\.com;NO_PROXY;.*/http://fallback:8080]">
    </properties>
  </provider>
</spi>
```

#### 7.4.2. Outgoing HTTPS Request Truststore

When Keycloak invokes on remote HTTPS endpoints, it has to validate the remote server's certificate in order to ensure it is connecting to a trusted server. This is necessary in order to prevent man-in-the-middle attacks. The certificates of these remote server's or the CA that signed these certificates must be put in a truststore. This truststore is managed by the Keycloak server.

The truststore is used when connecting securely to identity brokers, LDAP identity providers, when sending emails, and for backchannel communication with client applications.

By default, a truststore provider is not configured, and any https connections fall back to standard java truststore configuration as described in [Java's JSSE Reference](#).

[Reference Guide](#). If there is no trust established, then these outgoing HTTPS requests will fail.

You can use *keytool* to create a new truststore file or add trusted host certificates to an existing one:

```
$ keytool -import -alias HOSTDOMAIN -keystore truststore.jks -file host-certificate.cer
```

The truststore is configured within the `standalone.xml`, `standalone-ha.xml`, or `domain.xml` file in your distribution. The location of this file depends on your [operating mode](#). You can add your truststore configuration by using the following template:

```
<spi name="truststore">
    <provider name="file" enabled="true">
        <properties>
            <property name="file" value="path to your .jks
file containing public certificates"/>
            <property name="password" value="password"/>
            <property name="hostname-verification-policy"
value="WILDCARD"/>
            <property name="disabled" value="false"/>
        </properties>
    </provider>
</spi>
```

Possible configuration options for this setting are:

### file

The path to a Java keystore file. HTTPS requests need a way to verify the host of the server they are talking to. This is what the trustore

does. The keystore contains one or more trusted host certificates or certificate authorities. This truststore file should only contain public certificates of your secured hosts. This is *REQUIRED* if `disabled` is not true.

## **password**

Password for the truststore. This is *REQUIRED* if `disabled` is not true.

## **hostname-verification-policy**

`WILDCARD` by default. For HTTPS requests, this verifies the hostname of the server's certificate. `ANY` means that the hostname is not verified. `WILDCARD` Allows wildcards in subdomain names i.e. \*.foo.com. `STRICT` CN must match hostname exactly.

## **disabled**

If true (default value), truststore configuration will be ignored, and certificate checking will fall back to JSSE configuration as described. If set to false, you must configure `file`, and `password` for the truststore.

---

## 8. Clustering

This section covers configuring Keycloak to run in a cluster. There's a number of things you have to do when setting up a cluster, specifically:

- [Pick an operation mode](#)
- [Configure a shared external database](#)
- Set up a load balancer
- Supplying a private network that supports IP multicast

Picking an operation mode and configuring a shared database have been discussed earlier in this guide. In this chapter we'll discuss setting up a load balancer and supplying a private network. We'll also discuss some issues that you need to be aware of when booting up a host in the cluster.

It is possible to cluster Keycloak without IP Multicast, but this topic is beyond the scope of this guide. For more information, see [JGroups](#) chapter of the `{appserver_jgroups_name}`.

### 8.1. Recommended Network Architecture

The recommended network architecture for deploying Keycloak is to set up an HTTP/HTTPS load balancer on a public IP address that routes requests to Keycloak servers sitting on a private network. This isolates all clustering connections and provides a nice means of protecting the

servers.

By default, there is nothing to prevent unauthorized nodes from joining the cluster and broadcasting multi-cast messages. This is why cluster nodes should be in a private network, with a firewall protecting them from outside attacks.

## 8.2. Clustering Example

KeyCloak does come with an out of the box clustering demo that leverages domain mode. Review the [Clustered Domain Example](#) chapter for more details.

## 8.3. Setting Up a Load Balancer or Proxy

This section discusses a number of things you need to configure before you can put a reverse proxy or load balancer in front of your clustered KeyCloak deployment. It also covers configuring the built in load balancer that was [Clustered Domain Example](#).

### 8.3.1. Identifying Client IP Addresses

A few features in KeyCloak rely on the fact that the remote address of the HTTP client connecting to the authentication server is the real IP address of the client machine. Examples include:

- Event logs - a failed login attempt would be logged with the wrong source IP address
- SSL required - if the SSL required is set to external (the default) it should require SSL for all external requests

- Authentication flows - a custom authentication flow that uses the IP address to for example show OTP only for external requests
- Dynamic Client Registration

This can be problematic when you have a reverse proxy or loadbalancer in front of your KeyCloak authentication server. The usual setup is that you have a frontend proxy sitting on a public network that load balances and forwards requests to backend KeyCloak server instances located in a private network. There is some extra configuration you have to do in this scenario so that the actual client IP address is forwarded to and processed by the KeyCloak server instances. Specifically:

- Configure your reverse proxy or loadbalancer to properly set `X-Forwarded-For` and `X-Forwarded-Proto` HTTP headers.
- Configure your reverse proxy or loadbalancer to preserve the original 'Host' HTTP header.
- Configure the authentication server to read the client's IP address from `X-Forwarded-For` header.

Configuring your proxy to generate the `X-Forwarded-For` and `X-Forwarded-Proto` HTTP headers and preserving the original `Host` HTTP header is beyond the scope of this guide. Take extra precautions to ensure that the `X-Forwarded-For` header is set by your proxy. If your proxy isn't configured correctly, then *rogue* clients can set this header themselves and trick KeyCloak into thinking the client is connecting from a different IP address than it actually is. This becomes really important if you are doing any black or white listing of IP addresses.

Beyond the proxy itself, there are a few things you need to configure on the Keycloak side of things. If your proxy is forwarding requests via the HTTP protocol, then you need to configure Keycloak to pull the client's IP address from the `X-Forwarded-For` header rather than from the network packet. To do this, open up the profile configuration file (`standalone.xml`, `standalone-ha.xml`, or `domain.xml` depending on your [operating mode](#)) and look for the `{subsystem_undertow_xml_urn}` XML block.

#### X-Forwarded-For *HTTP Config*

```
<subsystem xmlns="{subsystem_undertow_xml_urn}">
    <buffer-cache name="default"/>
    <server name="default-server">
        <ajp-listener name="ajp" socket-binding="ajp"/>
        <http-listener name="default" socket-binding="http"
            redirect-socket="https"
            proxy-address-forwarding="true"/>
        ...
    </server>
    ...
</subsystem>
```

Add the `proxy-address-forwarding` attribute to the `http-listener` element. Set the value to `true`.

If your proxy is using the AJP protocol instead of HTTP to forward requests (i.e. Apache HTTPD + mod-cluster), then you have to configure things a little differently. Instead of modifying the `http-listener`, you need to add a filter to pull this information from the AJP packets.

#### X-Forwarded-For *AJP Config*

```
<subsystem xmlns="{subsystem_undertow_xml_urn}">
```

```

<buffer-cache name="default"/>
<server name="default-server">
    <ajp-listener name="ajp" socket-binding="ajp"/>
    <http-listener name="default" socket-binding="http" redirect-socket="https"/>
        <host name="default-host" alias="localhost">
            ...
                <filter-ref name="proxy-peer"/>
            </host>
        </server>
        ...
    <filters>
        ...
        <filter name="proxy-peer"
            class-name="io.undertow.server.handlers.
ProxyPeerAddressHandler"
            module="io.undertow.core" />
    </filters>
</subsystem>

```

### 8.3.2. Enable HTTPS/SSL with a Reverse Proxy

Assuming that your reverse proxy doesn't use port 8443 for SSL you also need to configure what port HTTPS traffic is redirected to.

```

<subsystem xmlns="{subsystem_undertow_xml_urn}">
    ...
    <http-listener name="default" socket-binding="http"
        proxy-address-forwarding="true" redirect-so-
        cket="proxy-https"/>
    ...
</subsystem>

```

Add the `redirect-socket` attribute to the `http-listener` element. The value should be `proxy-https` which points to a socket binding you also need to define.

Then add a new `socket-binding` element to the `socket-binding-`

group element:

```
<socket-binding-group name="standard-sockets" default-interface="public"
    port-offset="${jboss.socket.binding.port-offset:0}">
    ...
    <socket-binding name="proxy-https" port="443"/>
    ...
</socket-binding-group>
```

### 8.3.3. Verify Configuration

You can verify the reverse proxy or load balancer configuration by opening the path `/auth/realms/master/.well-known/openid-configuration` through the reverse proxy. For example if the reverse proxy address is `https://acme.com/` then open the URL `https://acme.com/auth/realms/master/.well-known/openid-configuration`. This will show a JSON document listing a number of endpoints for KeyCloak. Make sure the endpoints starts with the address (scheme, domain and port) of your reverse proxy or load balancer. By doing this you make sure that KeyCloak is using the correct endpoint.

You should also verify that KeyCloak sees the correct source IP address for requests. Do check this you can try to login to the admin console with an invalid username and/or password. This should show a warning in the server log something like this:

```
08:14:21,287 WARN  XNIO-1 task-45 [org.keycloak.events]
type=LOGIN_ERROR, realmId=master, clientId=security-admin-console, userId=8f20d7ba-4974-4811-a695-242c8fb1bf8, ipAddress=X.X.X.X, error=invalid_user_credentials, auth_method=openid-connect, auth_type=code, redirect_uri=http://localhost:8080/auth/admin/master/console/?redirect_fragment=%2Frealms%2Fmaster%2Fevents-settings,
```

```
code_id=a3d48b67-a439-4546-b992-e93311d6493e, username=admin
```

Check that the value of `ipAddress` is the IP address of the machine you tried to login with and not the IP address of the reverse proxy or load balancer.

#### 8.3.4. Using the Built-In Load Balancer

This section covers configuring the built in load balancer that is discussed in the [Clustered Domain Example](#).

The [Clustered Domain Example](#) is only designed to run on one machine. To bring up a slave on another host, you'll need to

1. Edit the `domain.xml` file to point to your new host slave
2. Copy the server distribution. You don't need the `domain.xml`, `host.xml`, or `host-master.xml` files. Nor do you need the `standalone/` directory.
3. Edit the `host-slave.xml` file to change the bind addresses used or override them on the command line

#### Register a New Host With Load Balancer

Let's look first at registering the new host slave with the load balancer configuration in `domain.xml`. Open this file and go to the undertow configuration in the `load-balancer` profile. Add a new `host` definition called `remote-host3` within the `reverse-proxy` XML block.

#### `domain.xml reverse-proxy config`

```
<subsystem xmlns="{subsystem_undertow_xml_urn}">
```

```

...
<handlers>
    <reverse-proxy name="lb-handler">
        <host name="host1" outbound-socket-binding="remote-host1" scheme="ajp" path="/" instance-id="myroute1"/>
        <host name="host2" outbound-socket-binding="remote-host2" scheme="ajp" path="/" instance-id="myroute2"/>
        <host name="remote-host3" outbound-socket-binding="remote-host3" scheme="ajp" path="/" instance-id="myroute3"/>
    </reverse-proxy>
</handlers>
...
</subsystem>
```

The `output-socket-binding` is a logical name pointing to a `socket-binding` configured later in the `domain.xml` file. The `instance-id` attribute must also be unique to the new host as this value is used by a cookie to enable sticky sessions when load balancing.

Next go down to the `load-balancer-sockets` `socket-binding-group` and add the `outbound-socket-binding` for `remote-host3`. This new binding needs to point to the host and port of the new host.

### *domain.xml outbound-socket-binding*

```

<socket-binding-group name="load-balancer-sockets" default-interface="public">
    ...
    <outbound-socket-binding name="remote-host1">
        <remote-destination host="localhost" port="8159"/>
    </outbound-socket-binding>
    <outbound-socket-binding name="remote-host2">
        <remote-destination host="localhost" port="8259"/>
    </outbound-socket-binding>
    <outbound-socket-binding name="remote-host3">
        <remote-destination host="192.168.0.5"
```

```
port="8259"/>
</outbound-socket-binding>
</socket-binding-group>
```

## Master Bind Addresses

Next thing you'll have to do is to change the `public` and `management` bind addresses for the master host. Either edit the `domain.xml` file as discussed in the [Bind Addresses](#) chapter or specify these bind addresses on the command line as follows:

```
$ domain.sh --host-config=host-master.xml -Djboss.bind.address=192.168.0.2 -Djboss.bind.address.management=192.168.0.2
```

## Host Slave Bind Addresses

Next you'll have to change the `public`, `management`, and `domain controller` bind addresses (`jboss.domain.master-address`). Either edit the `host-slave.xml` file or specify them on the command line as follows:

```
$ domain.sh --host-config=host-slave.xml
-Djboss.bind.address=192.168.0.5
-Djboss.bind.address.management=192.168.0.5
-Djboss.domain.master.address=192.168.0.2
```

The values of `jboss.bind.address` and `jboss.bind.address.management` pertain to the host slave's IP address. The value of `jboss.domain.master.address` need to be the IP address of the domain controller which is the management address of the master host.

### 8.3.5. Configuring Other Load Balancers

See [the load balancing](#) section in the `{appserver_loadbalancer_name}` for information how to use other software-based load balancers.

## 8.4. Sticky sessions

Typical cluster deployment consists of the load balancer (reverse proxy) and 2 or more KeyCloak servers on private network. For performance purposes, it may be useful if load balancer forwards all requests related to particular browser session to the same KeyCloak backend node.

The reason is, that KeyCloak is using Infinispan distributed cache under the covers for save data related to current authentication session and user session. The Infinispan distributed caches are configured with one owner by default. That means that particular session is saved just on one cluster node and the other nodes need to lookup the session remotely if they want to access it.

For example if authentication session with ID `123` is saved in the Infinispan cache on `node1`, and then `node2` needs to lookup this session, it needs to send the request to `node1` over the network to return the particular session entity.

It is beneficial if particular session entity is always available locally, which can be done with the help of sticky sessions. The workflow in the cluster environment with the public frontend load balancer and two backend KeyCloak nodes can be like this:

- User sends initial request to see the KeyCloak login screen
- This request is served by the frontend load balancer, which forwards it to some random node (eg. `node1`). Strictly said, the node doesn't

need to be random, but can be chosen according to some other criteria (client IP address etc). It all depends on the implementation and configuration of underlying load balancer (reverse proxy).

- Keycloak creates authentication session with random ID (eg. 123) and saves it to the Infinispan cache.
- Infinispan distributed cache assigns the primary owner of the session based on the hash of session ID. See [Infinispan documentation](#) for more details around this. Let's assume that Infinispan assigned `node2` to be the owner of this session.
- Keycloak creates the cookie `AUTH_SESSION_ID` with the format like `<session-id>.<owner-node-id>`. In our example case, it will be `123.node2`.
- Response is returned to the user with the Keycloak login screen and the `AUTH_SESSION_ID` cookie in the browser

From this point, it is beneficial if load balancer forwards all the next requests to the `node2` as this is the node, who is owner of the authentication session with ID `123` and hence Infinispan can lookup this session locally. After authentication is finished, the authentication session is converted to user session, which will be also saved on `node2` because it has same ID `123`.

The sticky session is not mandatory for the cluster setup, however it is good for performance for the reasons mentioned above. You need to configure your loadbalancer to sticky over the `AUTH_SESSION_ID` cookie. How exactly do this is dependent on your loadbalancer.

It is recommended on the Keycloak side to use the system property

`jboss.node.name` during startup, with the value corresponding to the name of your route. For example, `-Djboss.node.name=node1` will use `node1` to identify the route. This route will be used by Infinispan caches and will be attached to the AUTH\_SESSION\_ID cookie when the node is the owner of the particular key. Here is an example of the start up command using this system property:

```
cd $RHSSO_NODE1  
. ./standalone.sh -c standalone-ha.xml -Djboss.socket.  
binding.port-offset=100 -Djboss.node.name=node1
```

Typically in production environment the route name should use the same name as your backend host, but it is not required. You can use a different route name. For example, if you want to hide the host name of your KeyCloak server inside your private network.

#### 8.4.1. Disable adding the route

Some load balancers can be configured to add the route information by themselves instead of relying on the back end KeyCloak node. However, as described above, adding the route by the KeyCloak is recommended. This is because when done this way performance improves, since KeyCloak is aware of the entity that is the owner of particular session and can route to that node, which is not necessarily the local node.

You are permitted to disable adding route information to the AUTH\_SESSION\_ID cookie by KeyCloak, if you prefer, by adding the following into your `RHSSO_HOME/standalone/configuration/standalone-ha.xml` file in the KeyCloak subsystem configuration:

```
<subsystem xmlns="urn:jboss:domain:keycloak-server:1.1">
```

```

...
    <spi name="stickySessionEncoder">
        <provider name="infinispan" enabled="true">
            <properties>
                <property name="shouldAttachRoute"
value="false"/>
            </properties>
        </provider>
    </spi>

</subsystem>

```

## 8.5. Multicast Network Setup

Out of the box clustering support needs IP Multicast. Multicast is a network broadcast protocol. This protocol is used at boot time to discover and join the cluster. It is also used to broadcast messages for the replication and invalidation of distributed caches used by Keycloak.

The clustering subsystem for Keycloak runs on the JGroups stack. Out of the box, the bind addresses for clustering are bound to a private network interface with 127.0.0.1 as default IP address. You have to edit your the *standalone-ha.xml* or *domain.xml* sections discussed in the [Bind Address](#) chapter.

### *private network config*

```

<interfaces>
    ...
    <interface name="private">
        <inet-address value="${jboss.bind.address.
private:127.0.0.1}"/>
    </interface>
</interfaces>
<socket-binding-group name="standard-sockets" default-
interface="public" port-offset="${jboss.socket.binding.
port-offset:0}">

```

```

    ...
        <socket-binding name="jgroups-mping" interface="private" port="0" multicast-address="${jboss.default.multicast.address:230.0.0.4}" multicast-port="45700"/>
            <socket-binding name="jgroups-tcp" interface="private" port="7600"/>
            <socket-binding name="jgroups-tcp-fd" interface="private" port="57600"/>
            <socket-binding name="jgroups-udp" interface="private" port="55200" multicast-address="${jboss.default.multicast.address:230.0.0.4}" multicast-port="45688"/>
            <socket-binding name="jgroups-udp-fd" interface="private" port="54200"/>
            <socket-binding name="modcluster" port="0" multicast-address="224.0.1.105" multicast-port="23364"/>
        ...
    </socket-binding-group>

```

Things you'll want to configure are the `jboss.bind.address`, `private` and `jboss.default.multicast.address` as well as the ports of the services on the clustering stack.

It is possible to cluster KeyCloak without IP Multicast, but this topic is beyond the scope of this guide. For more information, see [JGroups](#) in the `{appserver-jgroups_name}`.

## 8.6. Securing Cluster Communication

When cluster nodes are isolated on a private network it requires access to the private network to be able to join a cluster or to view communication in the cluster. In addition you can also enable authentication and encryption for cluster communication. As long as your private network is secure it is not necessary to enable authentication and encryption. KeyCloak does not send very sensitive information on the cluster in either

case.

If you want to enable authentication and encryption for clustering communication see [Securing a Cluster](#) in the *JBoss EAP Configuration Guide*.

## 8.7. Serialized Cluster Startup

Keycloak cluster nodes are allowed to boot concurrently. When Keycloak server instance boots up it may do some database migration, importing, or first time initializations. A DB lock is used to prevent start actions from conflicting with one another when cluster nodes boot up concurrently.

By default, the maximum timeout for this lock is 900 seconds. If a node is waiting on this lock for more than the timeout it will fail to boot. Typically you won't need to increase/decrease the default value, but just in case it's possible to configure it in `standalone.xml`, `standalone-ha.xml`, or `domain.xml` file in your distribution. The location of this file depends on your [operating mode](#).

```
<spi name="dblock">
    <provider name="jpa" enabled="true">
        <properties>
            <property name="lockWaitTimeout" value="900"/>
        </properties>
    </provider>
</spi>
```

## 8.8. Booting the Cluster

Booting Keycloak in a cluster depends on your [operating mode](#)

## *Standalone Mode*

```
$ bin/standalone.sh --server-config=standalone-ha.xml
```

## *Domain Mode*

```
$ bin/domain.sh --host-config=host-master.xml  
$ bin/domain.sh --host-config=host-slave.xml
```

You may need to use additional parameters or system properties. For example, the parameter `-b` for the binding host or the system property `jboss.node.name` to specify the name of the route, as described in [Sticky Sessions](#) section.

## 8.9. Troubleshooting

- Note that when you run a cluster, you should see message similar to this in the log of both cluster nodes:

```
INFO [org.infinispan.remoting.transport.jgroups.  
JGroupsTransport] (Incoming-10,shared=udp)  
ISPN000094: Received new cluster view: [node1/keyclo-  
ak|1] (2) [node1/keycloak, node2/keycloak]
```

If you see just one node mentioned, it's possible that your cluster hosts are not joined together.

Usually it's best practice to have your cluster nodes on private network without firewall for communication among them. Firewall could be enabled just on public access point to your network instead. If for some reason you still need to have firewall enabled on cluster nodes, you will need to open some ports. Default values are UDP port 55200 and multicast port 45688 with multicast address

230.0.0.4. Note that you may need more ports opened if you want to enable additional features like diagnostics for your JGroups stack. Keycloak delegates most of the clustering work to Infinispan/JGroups. For more information, see [JGroups](#) in the `{appserver-jgroups_name}`.

- If you are interested in failover support (high availability), evictions, expiration and cache tuning, see [Server Cache Configuration](#).

---

## 9. Server Cache Configuration

Keycloak has two types of caches. One type of cache sits in front of the database to decrease load on the DB and to decrease overall response times by keeping data in memory. Realm, client, role, and user metadata is kept in this type of cache. This cache is a local cache. Local caches do not use replication even if you are in the cluster with more Keycloak servers. Instead, they only keep copies locally and if the entry is updated an invalidation message is sent to the rest of the cluster and the entry is evicted. There is separate replicated cache `work`, which task is to send the invalidation messages to the whole cluster about what entries should be evicted from local caches. This greatly reduces network traffic, makes things efficient, and avoids transmitting sensitive metadata over the wire.

The second type of cache handles managing user sessions, offline tokens, and keeping track of login failures so that the server can detect password phishing and other attacks. The data held in these caches is temporary, in memory only, but is possibly replicated across the cluster.

This chapter discusses some configuration options for these caches for both clustered and non-clustered deployments.

More advanced configuration of these caches can be found in the [Infinispan](#) section of the `{appserver_-caching_name}`.

## 9.1. Eviction and Expiration

There are multiple different caches configured for Keycloak. There is a realm cache that holds information about secured applications, general security data, and configuration options. There is also a user cache that contains user metadata. Both caches default to a maximum of 10000 entries and use a least recently used eviction strategy. Each of them is also tied to an object revisions cache that controls eviction in a clustered setup. This cache is created implicitly and has twice the configured size. The same applies for the `authorization` cache, which holds the authorization data. The `keys` cache holds data about external keys and does not need to have dedicated revisions cache. Rather it has `expiration` explicitly declared on it, so the keys are periodically expired and forced to be periodically downloaded from external clients or identity providers.

The eviction policy and max entries for these caches can be configured in the `standalone.xml`, `standalone-ha.xml`, or `domain.xml` depending on your [operating mode](#). In the configuration file, there is the part with infinispan subsystem, which looks similar to this:

```
<subsystem xmlns="{subsystem_infinispan_xml_urn}">
    <cache-container name="keycloak">
        <local-cache name="realms">
            <object-memory size="10000"/>
        </local-cache>
        <local-cache name="users">
            <object-memory size="10000"/>
        </local-cache>
        ...
        <local-cache name="keys">
            <object-memory size="1000"/>
            <expiration max-idle="3600000"/>
        </local-cache>
```

```
...  
</cache-container>
```

To limit or expand the number of allowed entries simply add or edit the `object` element or the `expiration` element of particular cache configuration.

In addition, there are also separate caches `sessions`, `clientSessions`, `offlineSessions`, `offlineClientSessions`, `loginFailures` and `actionTokens`. These caches are distributed in cluster environment and they are unbounded in size by default. If they are bounded, it would then be possible that some sessions will be lost. Expired sessions are cleared internally by KeyCloak itself to avoid growing the size of these caches without limit. If you see memory issues due to a large number of sessions, you can try to:

- Increase the size of cluster (more nodes in cluster means that sessions are spread more equally among nodes)
- Increase the memory for KeyCloak server process
- Decrease the number of owners to ensure that caches are saved in one single place. See [Replication and Failover](#) for more details
- Disable l1-lifespan for distributed caches. See Infinispan documentation for more details
- Decrease session timeouts, which could be done individually for each realm in KeyCloak admin console. But this could affect usability for end users. See [`{adminguide\_timeouts\_name}`](#) for more details.

There is an additional replicated cache, `work`, which is mostly used to send messages among cluster nodes; it is also unbounded by default. However, this cache should not cause any memory issues as entries in this cache are very short-lived.

## 9.2. Replication and Failover

There are caches like `sessions`, `authenticationSessions`, `offlineSessions`, `loginFailures` and a few others (See [Eviction and Expiration](#) for more details), which are configured as distributed caches when using a clustered setup. Entries are not replicated to every single node, but instead one or more nodes is chosen as an owner of that data. If a node is not the owner of a specific cache entry it queries the cluster to obtain it. What this means for failover is that if all the nodes that own a piece of data go down, that data is lost forever. By default, Keycloak only specifies one owner for data. So if that one node goes down that data is lost. This usually means that users will be logged out and will have to login again.

You can change the number of nodes that replicate a piece of data by change the `owners` attribute in the `distributed-cache` declaration.

### *owners*

```
<subsystem xmlns="{subsystem_infinispan_xml_urn}">
    <cache-container name="keycloak">
        <distributed-cache name="sessions" owners="2"/>
    ...

```

Here we've changed it so at least two nodes will replicate one specific user login session.

The number of owners recommended is really dependent on your deployment. If you do not care if users are logged out when a node goes down, then one owner is good enough and you will avoid replication.

It is generally wise to configure your environment to use loadbalancer with sticky sessions. It is beneficial for performance as KeyCloak server, where the particular request is served, will be usually the owner of the data from the distributed cache and will therefore be able to look up the data locally. See [Sticky sessions](#) for more details.

### 9.3. Disabling Caching

To disable the realm or user cache, you must edit the `standalone.xml`, `standalone-ha.xml`, or `domain.xml` file in your distribution. The location of this file depends on your [operating mode](#). Here's what the config looks like initially.

```
<spi name="userCache">
    <provider name="default" enabled="true"/>
</spi>

<spi name="realmCache">
    <provider name="default" enabled="true"/>
</spi>
```

To disable the cache set the `enabled` attribute to false for the cache you want to disable. You must reboot your server for this change to take

effect.

## 9.4. Clearing Caches at Runtime

To clear the realm or user cache, go to the Keycloak admin console Realm Settings → Cache Config page. On this page you can clear the realm cache, the user cache or cache of external public keys.

The cache will be cleared for all realms!

- 
1. Tracked as <https://issues.jboss.org/browse/KEYCLOAK-3873>

Last updated 2019-06-13 12:48:57 MESZ

# Table of Contents

{adAPTERguide!}

## 1. Overview

### 1.1. What are Client Adapters?

### 1.2. Supported Platforms

#### 1.2.1. OpenID Connect

#### 1.2.2. SAML

### 1.3. Supported Protocols

#### 1.3.1. OpenID Connect

#### 1.3.2. SAML 2.0

#### 1.3.3. OpenID Connect vs. SAML

## 2. OpenID Connect

### 2.1. Java Adapters

#### 2.1.1. Java Adapter Config

#### 2.1.2. Installing JBoss EAP Adapter from an RPM

#### 2.1.3. JBoss Fuse 6 Adapter

#### 2.1.4. JBoss Fuse 7 Adapter

#### 2.1.5. Spring Boot Adapter

#### 2.1.6. Java Servlet Filter Adapter

#### 2.1.7. Security Context

#### 2.1.8. Error Handling

#### 2.1.9. Logout

#### 2.1.10. Parameters Forwarding

#### 2.1.11. Client Authentication

#### 2.1.12. Multi Tenancy

#### 2.1.13. Application Clustering

### 2.2. JavaScript Adapter

- 2.2.1. Session Status iframe
- 2.2.2. Implicit and Hybrid Flow
- 2.2.3. Hybrid Apps with Cordova
- 2.2.4. Earlier Browsers
- 2.2.5. JavaScript Adapter Reference
- 2.3. Node.js Adapter
  - 2.3.1. Installation
  - 2.3.2. Usage
  - 2.3.3. Installing Middleware
  - 2.3.4. Checking Authentication
  - 2.3.5. Protecting Resources
  - 2.3.6. Additional URLs
- 2.4. Other OpenID Connect Libraries
  - 2.4.1. Endpoints
  - 2.4.2. Validating Access Tokens
  - 2.4.3. Flows
  - 2.4.4. Redirect URLs

### 3. SAML

- 3.1. Java Adapters
  - 3.1.1. General Adapter Config
  - 3.1.2. Installing JBoss EAP Adapter from an RPM
  - 3.1.3. Java Servlet Filter Adapter
  - 3.1.4. Registering with an Identity Provider
  - 3.1.5. Logout
  - 3.1.6. Obtaining Assertion Attributes
  - 3.1.7. Error Handling
  - 3.1.8. Troubleshooting
  - 3.1.9. Multi Tenancy

### 3.2. mod\_auth\_mellon Apache HTTPD Module

#### 3.2.1. Configuring mod\_auth\_mellon with Keycloak

## 4. Docker Registry Configuration

### 4.1. Docker Registry Configuration File Installation

### 4.2. Docker Registry Environment Variable Override Installation

### 4.3. Docker Compose YAML File

## 5. Client Registration

### 5.1. Authentication

#### 5.1.1. Bearer Token

#### 5.1.2. Initial Access Token

#### 5.1.3. Registration Access Token

### 5.2. Keycloak Representations

### 5.3. Keycloak Adapter Configuration

### 5.4. OpenID Connect Dynamic Client Registration

### 5.5. SAML Entity Descriptors

### 5.6. Example using CURL

### 5.7. Example using Java Client Registration API

### 5.8. Client Registration Policies

## 6. Client Registration CLI

### 6.1. Configuring a new regular user for use with Client Registration CLI

### 6.2. Configuring a client for use with the Client Registration CLI

### 6.3. Installing the Client Registration CLI

### 6.4. Using the Client Registration CLI

#### 6.4.1. Logging in

#### 6.4.2. Working with alternative configurations

#### 6.4.3. Initial Access and Registration Access Tokens

- 6.4.4. Creating a client configuration
  - 6.4.5. Retrieving a client configuration
  - 6.4.6. Modifying a client configuration
  - 6.4.7. Deleting a client configuration
  - 6.4.8. Refreshing invalid Registration Access Tokens
  - 6.5. Troubleshooting
7. Token Exchange
- 7.1. Internal Token to Internal Token Exchange
    - 7.1.1. Granting Permission for the Exchange
    - 7.1.2. Making the Request
  - 7.2. Internal Token to External Token Exchange
    - 7.2.1. Granting Permission for the Exchange
    - 7.2.2. Making the Request
  - 7.3. External Token to Internal Token Exchange
    - 7.3.1. Granting Permission for the Exchange
    - 7.3.2. Making the Request
  - 7.4. Impersonation
    - 7.4.1. Granting Permission for the Exchange
    - 7.4.2. Making the Request
  - 7.5. Direct Naked Impersonation
    - 7.5.1. Granting Permission for the Exchange
    - 7.5.2. Making the Request
  - 7.6. Expand Permission Model With Service Accounts
  - 7.7. Exchange Vulnerabilities

# {adapterguide!}

# 1. Overview

KeyCloak supports both OpenID Connect (an extension to OAuth 2.0) and SAML 2.0. When securing clients and services the first thing you need to decide is which of the two you are going to use. If you want you can also choose to secure some with OpenID Connect and others with SAML.

To secure clients and services you are also going to need an adapter or library for the protocol you've selected. KeyCloak comes with its own adapters for selected platforms, but it is also possible to use generic OpenID Connect Relying Party and SAML Service Provider libraries.

## 1.1. What are Client Adapters?

KeyCloak client adapters are libraries that makes it very easy to secure applications and services with KeyCloak. We call them adapters rather than libraries as they provide a tight integration to the underlying platform and framework. This makes our adapters easy to use and they require less boilerplate code than what is typically required by a library.

## 1.2. Supported Platforms

### 1.2.1. OpenID Connect

#### Java

- [JBoss EAP](#)
- [Fuse](#)
- [Servlet Filter](#)

- [Spring Boot](#)

## JavaScript (client-side)

- [JavaScript](#)

## Node.js (server-side)

- [Node.js](#)

### 1.2.2. SAML

#### Java

- [JBoss EAP](#)

#### Apache HTTP Server

- [mod\\_auth\\_mellon](#)

## 1.3. Supported Protocols

### 1.3.1. OpenID Connect

[OpenID Connect](#) (OIDC) is an authentication protocol that is an extension of [OAuth 2.0](#). While OAuth 2.0 is only a framework for building authorization protocols and is mainly incomplete, OIDC is a full-fledged authentication and authorization protocol. OIDC also makes heavy use of the [Json Web Token](#) (JWT) set of standards. These standards define an identity token JSON format and ways to digitally sign and encrypt that data in a compact and web-friendly way.

There are really two types of use cases when using OIDC. The first is an application that asks the KeyCloak server to authenticate a user for them. After a successful login, the application will receive an *identity token* and an *access token*. The *identity token* contains information

about the user such as username, email, and other profile information. The *access token* is digitally signed by the realm and contains access information (like user role mappings) that the application can use to determine what resources the user is allowed to access on the application.

The second type of use cases is that of a client that wants to gain access to remote services. In this case, the client asks KeyCloak to obtain an *access token* it can use to invoke on other remote services on behalf of the user. KeyCloak authenticates the user then asks the user for consent to grant access to the client requesting it. The client then receives the *access token*. This *access token* is digitally signed by the realm. The client can make REST invocations on remote services using this *access token*. The REST service extracts the *access token*, verifies the signature of the token, then decides based on access information within the token whether or not to process the request.

### 1.3.2. SAML 2.0

[SAML 2.0](#) is a similar specification to OIDC but a lot older and more mature. It has its roots in SOAP and the plethora of WS-\* specifications so it tends to be a bit more verbose than OIDC. SAML 2.0 is primarily an authentication protocol that works by exchanging XML documents between the authentication server and the application. XML signatures and encryption are used to verify requests and responses.

In KeyCloak SAML serves two types of use cases: browser applications and REST invocations.

There are really two types of use cases when using SAML. The first is an application that asks the KeyCloak server to authenticate a user for

them. After a successful login, the application will receive an XML document that contains something called a SAML assertion that specifies various attributes about the user. This XML document is digitally signed by the realm and contains access information (like user role mappings) that the application can use to determine what resources the user is allowed to access on the application.

The second type of use cases is that of a client that wants to gain access to remote services. In this case, the client asks KeyCloak to obtain a SAML assertion it can use to invoke on other remote services on behalf of the user.

### 1.3.3. OpenID Connect vs. SAML

Choosing between OpenID Connect and SAML is not just a matter of using a newer protocol (OIDC) instead of the older more mature protocol (SAML).

In most cases KeyCloak recommends using OIDC.

SAML tends to be a bit more verbose than OIDC.

Beyond verbosity of exchanged data, if you compare the specifications you'll find that OIDC was designed to work with the web while SAML was retrofitted to work on top of the web. For example, OIDC is also more suited for HTML5/JavaScript applications because it is easier to implement on the client side than SAML. As tokens are in the JSON format, they are easier to consume by JavaScript. You will also find several nice features that make implementing security in your web applications easier. For example, check out the [iframe trick](#) that the specification uses to easily determine if a user is still logged in or not.

SAML has its uses though. As you see the OIDC specifications evolve you see they implement more and more features that SAML has had for years. What we often see is that people pick SAML over OIDC because of the perception that it is more mature and also because they already have existing applications that are secured with it.

## 2. OpenID Connect

This section describes how you can secure applications and services with OpenID Connect using either KeyCloak adapters or generic OpenID Connect Relying Party libraries.

### 2.1. Java Adapters

KeyCloak comes with a range of different adapters for Java application. Selecting the correct adapter depends on the target platform.

All Java adapters share a set of common configuration options described in the [Java Adapters Config](#) chapter.

#### 2.1.1. Java Adapter Config

Each Java adapter supported by KeyCloak can be configured by a simple JSON file. This is what one might look like:

```
{  
  "realm" : "demo",  
  "resource" : "customer-portal",  
  "realm-public-key" : "MIGfMA0GCSqGSIb3D...31LwIDAQAB",  
  "auth-server-url" : "https://localhost:8443/auth",  
  "ssl-required" : "external",  
  "use-resource-role-mappings" : false,  
  "enable-cors" : true,  
  "cors-max-age" : 1000,  
  "cors-allowed-methods" : "POST, PUT, DELETE, GET",  
  "cors-exposed-headers" : "WWW-Authenticate, My-custom-ex-  
    posed-Header",  
  "bearer-only" : false,  
  "enable-basic-auth" : false,  
  "expose-token" : true,
```

```

"verify-token-audience" : true,
"credentials" : {
    "secret" : "234234-234234-234234"
},

"connection-pool-size" : 20,
"disable-trust-manager": false,
"allow-any-hostname" : false,
"truststore" : "path/to/truststore.jks",
"truststore-password" : "geheim",
"client-keystore" : "path/to/client-keystore.jks",
"client-keystore-password" : "geheim",
"client-key-password" : "geheim",
"token-minimum-time-to-live" : 10,
"min-time-between-jwks-requests" : 10,
"public-key-cache-ttl": 86400,
"redirect-rewrite-rules" : {
    "^/wsmaster/api/(.*)$" : "/api/$1"
}
}

```

You can use  `${...}`  enclosure for system property replacement. For example  `${jboss.server.config.dir}`  would be replaced by  `/path/to/KeyCloak` . Replacement of environment variables is also supported via the  `env`  prefix, e.g.  `${env.MY_ENVIRONMENT_VARIABLE}` .

The initial config file can be obtained from the admin console. This can be done by opening the admin console, select  `Clients`  from the menu and clicking on the corresponding client. Once the page for the client is opened click on the  `Installation`  tab and select  `Keycloak OIDC JSON` .

Here is a description of each configuration option:

## **realm**

Name of the realm. This is *REQUIRED*.

### **resource**

The client-id of the application. Each application has a client-id that is used to identify the application. This is *REQUIRED*.

### **realm-public-key**

PEM format of the realm public key. You can obtain this from the administration console. This is *OPTIONAL* and it's not recommended to set it. If not set, the adapter will download this from KeyCloak and it will always re-download it when needed (eg. KeyCloak rotate its keys). However if realm-public-key is set, then adapter will never download new keys from KeyCloak, so when KeyCloak rotate its keys, adapter will break.

### **auth-server-url**

The base URL of the KeyCloak server. All other KeyCloak pages and REST service endpoints are derived from this. It is usually of the form `https://host:port/auth`. This is *REQUIRED*.

### **ssl-required**

Ensures that all communication to and from the KeyCloak server is over HTTPS. In production this should be set to `all`. This is *OPTIONAL*. The default value is *external* meaning that HTTPS is required by default for external requests. Valid values are 'all', 'external' and 'none'.

### **confidential-port**

The confidential port used by the KeyCloak server for secure con-

nections over SSL/TLS. This is *OPTIONAL*. The default value is `8443`.

### **use-resource-role-mappings**

If set to true, the adapter will look inside the token for application level role mappings for the user. If false, it will look at the realm level for user role mappings. This is *OPTIONAL*. The default value is `false`.

### **public-client**

If set to true, the adapter will not send credentials for the client to Keycloak. This is *OPTIONAL*. The default value is `false`.

### **enable-cors**

This enables CORS support. It will handle CORS preflight requests. It will also look into the access token to determine valid origins. This is *OPTIONAL*. The default value is `false`.

### **cors-max-age**

If CORS is enabled, this sets the value of the `Access-Control-Max-Age` header. This is *OPTIONAL*. If not set, this header is not returned in CORS responses.

### **cors-allowed-methods**

If CORS is enabled, this sets the value of the `Access-Control-Allow-Methods` header. This should be a comma-separated string. This is *OPTIONAL*. If not set, this header is not returned in CORS responses.

## **cors-allowed-headers**

If CORS is enabled, this sets the value of the `Access-Control-Allow-Headers` header. This should be a comma-separated string. This is *OPTIONAL*. If not set, this header is not returned in CORS responses.

## **cors-exposed-headers**

If CORS is enabled, this sets the value of the `Access-Control-Expose-Headers` header. This should be a comma-separated string. This is *OPTIONAL*. If not set, this header is not returned in CORS responses.

## **bearer-only**

This should be set to *true* for services. If enabled the adapter will not attempt to authenticate users, but only verify bearer tokens. This is *OPTIONAL*. The default value is *false*.

## **autodetect-bearer-only**

This should be set to *true* if your application serves both a web application and web services (e.g. SOAP or REST). It allows you to redirect unauthenticated users of the web application to the Keycloak login page, but send an HTTP `401` status code to unauthenticated SOAP or REST clients instead as they would not understand a redirect to the login page. Keycloak auto-detects SOAP or REST clients based on typical headers like `X-Requested-With`, `SOAPAction` or `Accept`. The default value is *false*.

## **enable-basic-auth**

This tells the adapter to also support basic authentication. If this option is enabled, then *secret* must also be provided. This is *OPTIONAL*. The default value is *false*.

### **expose-token**

If `true`, an authenticated browser client (via a JavaScript HTTP invocation) can obtain the signed access token via the URL `root/k_query_bearer_token`. This is *OPTIONAL*. The default value is *false*.

### **credentials**

Specify the credentials of the application. This is an object notation where the key is the credential type and the value is the value of the credential type. Currently password and jwt is supported. This is *REQUIRED* only for clients with 'Confidential' access type.

### **connection-pool-size**

Adapters will make separate HTTP invocations to the KeyCloak server to turn an access code into an access token. This config option defines how many connections to the KeyCloak server should be pooled. This is *OPTIONAL*. The default value is `20`.

### **disable-trust-manager**

If the KeyCloak server requires HTTPS and this config option is set to `true` you do not have to specify a truststore. This setting should only be used during development and **never** in production as it will disable verification of SSL certificates. This is *OPTIONAL*. The default value is `false`.

## **allow-any-hostname**

If the Keycloak server requires HTTPS and this config option is set to `true` the Keycloak server's certificate is validated via the truststore, but host name validation is not done. This setting should only be used during development and **never** in production as it will disable verification of SSL certificates. This setting may be useful in test environments. This is *OPTIONAL*. The default value is `false`.

## **proxy-url**

The URL for the HTTP proxy if one is used.

## **truststore**

The value is the file path to a truststore file. If you prefix the path with `classpath:`, then the truststore will be obtained from the deployment's classpath instead. Used for outgoing HTTPS communications to the Keycloak server. Client making HTTPS requests need a way to verify the host of the server they are talking to. This is what the truststore does. The keystore contains one or more trusted host certificates or certificate authorities. You can create this truststore by extracting the public certificate of the Keycloak server's SSL keystore. This is *REQUIRED* unless `ssl-required` is `none` or `disable-trust-manager` is `true`.

## **truststore-password**

Password for the truststore. This is *REQUIRED* if `truststore` is set and the truststore requires a password.

## **client-keystore**

This is the file path to a keystore file. This keystore contains client certificate for two-way SSL when the adapter makes HTTPS requests to the KeyCloak server. This is *OPTIONAL*.

### **client-keystore-password**

Password for the client keystore. This is *REQUIRED* if `client-keystore` is set.

### **client-key-password**

Password for the client's key. This is *REQUIRED* if `client-keystore` is set.

### **always-refresh-token**

If *true*, the adapter will refresh token in every request.

### **register-node-at-startup**

If *true*, then adapter will send registration request to KeyCloak. It's *false* by default and useful only when application is clustered. See [Application Clustering](#) for details

### **register-node-period**

Period for re-registration adapter to KeyCloak. Useful when application is clustered. See [Application Clustering](#) for details

### **token-store**

Possible values are *session* and *cookie*. Default is *session*, which means that adapter stores account info in HTTP Session. Alternative *cookie* means storage of info in cookie. See [Application Clustering](#) for details

## **token-cookie-path**

When using a cookie store, this option sets the path of the cookie used to store account info. If it's a relative path, then it is assumed that the application is running in a context root, and is interpreted relative to that context root. If it's an absolute path, then the absolute path is used to set the cookie path. Defaults to use paths relative to the context root.

## **principal-attribute**

OpenID Connect ID Token attribute to populate the UserPrincipal name with. If token attribute is null, defaults to `sub`. Possible values are `sub`, `preferred_username`, `email`, `name`, `nickname`, `given_name`, `family_name`.

## **turn-off-change-session-id-on-login**

The session id is changed by default on a successful login on some platforms to plug a security attack vector. Change this to true if you want to turn this off This is *OPTIONAL*. The default value is *false*.

## **token-minimum-time-to-live**

Amount of time, in seconds, to preemptively refresh an active access token with the KeyCloak server before it expires. This is especially useful when the access token is sent to another REST client where it could expire before being evaluated. This value should never exceed the realm's access token lifespan. This is *OPTIONAL*. The default value is `0` seconds, so adapter will refresh access token just if it's expired.

## **min-time-between-jwks-requests**

Amount of time, in seconds, specifying minimum interval between two requests to KeyCloak to retrieve new public keys. It is 10 seconds by default. Adapter will always try to download new public key when it recognize token with unknown `kid`. However it won't try it more than once per 10 seconds (by default). This is to avoid DoS when attacker sends lots of tokens with bad `kid` forcing adapter to send lots of requests to KeyCloak.

### **public-key-cache-ttl**

Amount of time, in seconds, specifying maximum interval between two requests to KeyCloak to retrieve new public keys. It is 86400 seconds (1 day) by default. Adapter will always try to download new public key when it recognize token with unknown `kid`. If it recognize token with known `kid`, it will just use the public key downloaded previously. However at least once per this configured interval (1 day by default) will be new public key always downloaded even if the `kid` of token is already known.

### **ignore-oauth-query-parameter**

Defaults to `false`, if set to `true` will turn off processing of the `access_token` query parameter for bearer token processing. Users will not be able to authenticate if they only pass in an `access_token`

### **redirect-rewrite-rules**

If needed, specify the Redirect URI rewrite rule. This is an object notation where the key is the regular expression to which the Redirect URI is to be matched and the value is the replacement String. `$` character can be used for backreferences in the replacement String.

## **verify-token-audience**

If set to `true`, then during authentication with the bearer token, the adapter will verify whether the token contains this client name (resource) as an audience. The option is especially useful for services, which primarily serve requests authenticated by the bearer token.

This is set to `false` by default, however for improved security, it is recommended to enable this. See [Audience Support](#) for more details about audience support.

You can provide an adapter config file in your WAR and change the auth-method to KEYCLOAK within web.xml.

Alternatively, you don't have to modify your WAR at all and you can secure it via the KeyCloak adapter subsystem configuration in the configuration file, such as `standalone.xml`. Both methods are described in this section.

### **Installing the adapter**

Adapters are available as a separate archive depending on what server version you are using.

This ZIP archive contains JBoss Modules specific to the KeyCloak adapter. It also contains JBoss CLI scripts to configure the adapter subsystem.

To configure the adapter subsystem if the server is not running execute:

Alternatively, you can specify the `server.config` property while installing adapters from the command

line to install adapters using a different config, for example: `-Dserver.config=standalone-ha.xml`.

Alternatively, if the server is running execute:

## JBoss SSO

WildFly has built-in support for single sign-on for web applications deployed to the same WildFly instance. This should not be enabled when using KeyCloak.

## Required Per WAR Configuration

This section describes how to secure a WAR directly by adding configuration and editing files within your WAR package.

The first thing you must do is create a `keycloak.json` adapter configuration file within the `WEB-INF` directory of your WAR.

The format of this configuration file is described in the [Java adapter configuration](#) section.

Next you must set the `auth-method` to `KEYCLOAK` in `web.xml`. You also have to use standard servlet security to specify role-base constraints on your URLs.

Here's an example:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0">
```

```

<module-name>application</module-name>

<security-constraint>
    <web-resource-collection>
        <web-resource-name>Admins</web-resource-name>
        <url-pattern>/admin/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admin</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-
guarantee>
    </user-data-constraint>
</security-constraint>
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Customers</web-resource-na-
me>
        <url-pattern>/customers/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>user</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-
guarantee>
    </user-data-constraint>
</security-constraint>

<login-config>
    <auth-method>KEYCLOAK</auth-method>
    <realm-name>this is ignored currently</realm-name>
</login-config>

<security-role>
    <role-name>admin</role-name>
</security-role>
<security-role>
    <role-name>user</role-name>
</security-role>
</web-app>

```

## Securing WARs via Adapter Subsystem

You do not have to modify your WAR to secure it with Keycloak. Instead you can externally secure it via the Keycloak Adapter Subsystem. While you don't have to specify KEYCLOAK as an `auth-method`, you still have to define the `security-constraints` in `web.xml`. You do not, however, have to create a `WEB-INF/keycloak.json` file. This metadata is instead defined within server configuration (i.e. `standalone.xml`) in the Keycloak subsystem definition.

```
<extensions>
    <extension module="org.keycloak.keycloak-adapter-subsystem"/>
</extensions>

<profile>
    <subsystem xmlns="urn:jboss:domain:keycloak:1.1">
        <secure-deployment name="WAR MODULE NAME.war">
            <realm>demo</realm>
            <auth-server-url>http://localhost:8081/auth</auth-server-url>
            <ssl-required>external</ssl-required>
            <resource>customer-portal</resource>
            <credential name="secret">password</credential>
        </secure-deployment>
    </subsystem>
</profile>
```

The `secure-deployment name` attribute identifies the WAR you want to secure. Its value is the `module-name` defined in `web.xml` with `.war` appended. The rest of the configuration corresponds pretty much one to one with the `keycloak.json` configuration options defined in [Java adapter configuration](#).

The exception is the `credential` element.

To make it easier for you, you can go to the KeyCloak Administration Console and go to the Client/Installation tab of the application this WAR is aligned with. It provides an example XML file you can cut and paste.

If you have multiple deployments secured by the same realm you can share the realm configuration in a separate element. For example:

```
<subsystem xmlns="urn:jboss:domain:keycloak:1.1">
    <realm name="demo">
        <auth-server-url>http://localhost:8080/auth</auth-
server-url>
        <ssl-required>external</ssl-required>
    </realm>
    <secure-deployment name="customer-portal.war">
        <realm>demo</realm>
        <resource>customer-portal</resource>
        <credential name="secret">password</credential>
    </secure-deployment>
    <secure-deployment name="product-portal.war">
        <realm>demo</realm>
        <resource>product-portal</resource>
        <credential name="secret">password</credential>
    </secure-deployment>
    <secure-deployment name="database.war">
        <realm>demo</realm>
        <resource>database-service</resource>
        <bearer-only>true</bearer-only>
    </secure-deployment>
</subsystem>
```

## Security Domain

To propagate the security context to the EJB tier you need to configure it to use the "keycloak" security domain. This can be achieved with the @SecurityDomain annotation:

```
import org.jboss.ejb3.annotation.SecurityDomain;
...
@Stateless
@SecurityDomain("keycloak")
public class CustomerService {

    @RolesAllowed("user")
    public List<String> getCustomers() {
        return db.getCustomers();
    }
}
```

## 2.1.2. Installing JBoss EAP Adapter from an RPM

Install the EAP 7 Adapters from an RPM:

With Red Hat Enterprise Linux 7, the term channel was replaced with the term repository. In these instructions only the term repository is used.

You must subscribe to the WildFly {appserver\_version} repository before you can install the WildFly 7 adapters from an RPM.

### *Prerequisites*

1. Ensure that your Red Hat Enterprise Linux system is registered to your account using Red Hat Subscription Manager. For more information see the [Red Hat Subscription Management documentation](#).
2. If you are already subscribed to another JBoss EAP repository, you must unsubscribe from that repository first.

For Red Hat Enterprise Linux 6, 7: Using Red Hat Subscription Manager, subscribe to the WildFly {appserver\_version} repository using the

following command. Replace <RHEL\_VERSION> with either 6 or 7 depending on your Red Hat Enterprise Linux version.

```
$ sudo subscription-manager repos --enable=jb-eap-7-for-rhel-<RHEL_VERSION>-server-rpms
```

For Red Hat Enterprise Linux 8: Using Red Hat Subscription Manager, subscribe to the WildFly {appserver\_version} repository using the following command:

```
$ sudo subscription-manager repos --enable=jb-eap-{appserver_version}-for-rhel-8-x86_64-rpms --enable=rhel-8-for-x86_64-baseos-rpms --enable=rhel-8-for-x86_64-appstream-rpms
```

Install the WildFly 7 adapters for OIDC using the following command at Red Hat Enterprise Linux 6, 7:

```
$ sudo yum install eap7-keycloak-adapter-sso7_3
```

or use following one for Red Hat Red Hat Enterprise Linux 8:

```
$ sudo dnf install eap7-keycloak-adapter-sso7_3
```

The default EAP\_HOME path for the RPM installation is /opt/rh/eap7/root/usr/share/wildfly.

Run the appropriate module installation script.

For the OIDC module, enter the following command:

```
$ $EAP_HOME/bin/jboss-cli.sh -c --file=$EAP_HOME/bin/adap-  
ter-install.cli
```

Your installation is complete.

Install the EAP 6 Adapters from an RPM:

With Red Hat Enterprise Linux 7, the term channel was replaced with the term repository. In these instructions only the term repository is used.

You must subscribe to the JBoss EAP 6 repository before you can install the EAP 6 adapters from an RPM.

### *Prerequisites*

1. Ensure that your Red Hat Enterprise Linux system is registered to your account using Red Hat Subscription Manager. For more information see the [Red Hat Subscription Management documentation](#).
2. If you are already subscribed to another JBoss EAP repository, you must unsubscribe from that repository first.

Using Red Hat Subscription Manager, subscribe to the JBoss EAP 6 repository using the following command. Replace <RHEL\_VERSION> with either 6 or 7 depending on your Red Hat Enterprise Linux version.

```
$ sudo subscription-manager repos --enable=jb-eap-6-for-  
rhel-<RHEL_VERSION>-server-rpms
```

Install the EAP 6 adapters for OIDC using the following command:

```
$ sudo yum install keycloak-adapter-sso7_3-eap6
```

The default EAP\_HOME path for the RPM installation is /opt/rh/eap6/root/usr/share/wildfly.

Run the appropriate module installation script.

For the OIDC module, enter the following command:

```
$ $EAP_HOME/bin/jboss-cli.sh -c --file=$EAP_HOME/bin/adapter-install.cli
```

Your installation is complete.

### 2.1.3. JBoss Fuse 6 Adapter

KeyCloak supports securing your web applications running inside [JBoss Fuse 6](#).

The only supported version of Fuse 6 is the latest release. If you use earlier versions of Fuse 6, it is possible that some functions will not work correctly. In particular, the [Hawtio](#) integration will not work with earlier versions of Fuse 6.

Security for the following items is supported for Fuse:

- Classic WAR applications deployed on Fuse with Pax Web War Ex-

tender

- Servlets deployed on Fuse as OSGI services with Pax Web Whiteboard Extender
- [Apache Camel](#) Jetty endpoints running with the [Camel Jetty](#) component
- [Apache CXF](#) endpoints running on their own separate [Jetty engine](#)
- [Apache CXF](#) endpoints running on the default engine provided by the CXF servlet
- SSH and JMX admin access
- [Hawtio administration console](#)

## Securing Your Web Applications Inside Fuse 6

You must first install the KeyCloak Karaf feature. Next you will need to perform the steps according to the type of application you want to secure. All referenced web applications require injecting the KeyCloak Jetty authenticator into the underlying Jetty server. The steps to achieve this depend on the application type. The details are described below.

### Installing the Keycloak Feature

You must first install the `keycloak` feature in the JBoss Fuse environment. The keycloak feature includes the Fuse adapter and all third-party dependencies. You can install it either from the Maven repository or from an archive.

### Installing from the Maven Repository

As a prerequisite, you must be online and have access to the Maven repository.

To install the keycloak feature using the Maven repository, complete the following steps:

1. Start {fuseVersion}; then in the Karaf terminal type:

```
features:addurl mvn:org.keycloak/keycloak-osgi-features/{project_versionMvn}/xml/features  
features:install keycloak
```

2. You might also need to install the Jetty 9 feature:

```
features:install keycloak-jetty9-adapter
```

3. Ensure that the features were installed:

```
features:list | grep keycloak
```

### Installing from the ZIP bundle

This is useful if you are offline or do not want to use Maven to obtain the JAR files and other artifacts.

To install the Fuse adapter from the ZIP archive, complete the following steps:

1. Download the KeyCloak Fuse adapter ZIP archive.
2. Unzip it into the root directory of JBoss Fuse. The dependencies are then installed under the `system` directory. You can overwrite all existing jar files.

Use this for {fuseVersion}:

```
cd /path-to-fuse/jboss-fuse-6.3.0.redhat-254
```

3. Start Fuse and run these commands in the fuse/karaf terminal:

```
features:addurl mvn:org.keycloak/keycloak-osgi-features/{project_versionMvn}/xml/features  
features:install keycloak
```

4. Install the corresponding Jetty adapter. Since the artifacts are available directly in the JBoss Fuse `system` directory, you do not need to use the Maven repository.

## Securing a Classic WAR Application

The needed steps to secure your WAR application are:

1. In the `/WEB-INF/web.xml` file, declare the necessary:
  - security constraints in the `<security-constraint>` element
  - login configuration in the `<login-config>` element
  - security roles in the `<security-role>` element.

For example:

```
<?xml version="1.0" encoding="UTF-8"?>  
<web-app xmlns="http://java.sun.com/xml/ns/javaee"  
         xmlns:xsi="http://www.w3.org/2001/  
         XMLSchema-instance"  
         xsi:schemaLocation="http://java.sun.com/  
         xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-  
         app_3_0.xsd"  
         version="3.0">  
  
    <module-name>customer-portal</module-name>
```

```

<welcome-file-list>
    <welcome-file>index.html</welcome-file>
</welcome-file-list>

<security-constraint>
    <web-resource-collection>
        <web-resource-name>Customers</web-re-
source-name>
        <url-pattern>/customers/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>user</role-name>
    </auth-constraint>
</security-constraint>

<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>does-not-matter</realm-name>
</login-config>

<security-role>
    <role-name>admin</role-name>
</security-role>
<security-role>
    <role-name>user</role-name>
</security-role>
</web-app>

```

2. Add the `jetty-web.xml` file with the authenticator to the `/WEB-INF/jetty-web.xml` file.

For example:

```

<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD
Configure//EN"
"http://www.eclipse.org/jetty/configure_9_0.dtd">
<Configure class="org.eclipse.jetty.webapp.WebAppCon-
text">
    <Get name="securityHandler">
        <Set name="authenticator">

```

```
<New class="org.keycloak.adapters.jetty.  
KeycloakJettyAuthenticator">  
    </New>  
    </Set>  
    </Get>  
</Configure>
```

3. Within the `/WEB-INF/` directory of your WAR, create a new file, `keycloak.json`. The format of this configuration file is described in the [Java Adapters Config](#) section. It is also possible to make this file available externally as described in [Configuring the External Adapter](#).
4. Ensure your WAR application imports `org.keycloak.adapters.jetty` and maybe some more packages in the `META-INF/MANIFEST.MF` file, under the `Import-Package` header. Using `maven-bundle-plugin` in your project properly generates OSGI headers in manifest. Note that "\*" resolution for the package does not import the `org.keycloak.adapters.jetty` package, since it is not used by the application or the Blueprint or Spring descriptor, but is rather used in the `jetty-web.xml` file.

The list of the packages to import might look like this:

```
org.keycloak.adapters.jetty;version="{project_versionMvn}",  
org.keycloak.adapters;version="{project_versionMvn}",  
org.keycloak.constants;version="{project_versionMvn}",  
org.keycloak.util;version="{project_versionMvn}",  
org.keycloak.*;version="{project_versionMvn}",  
*;resolution:=optional
```

## Configuring the External Adapter

If you do not want the `keycloak.json` adapter configuration file to be

bundled inside your WAR application, but instead made available externally and loaded based on naming conventions, use this configuration method.

To enable the functionality, add this section to your `/WEB_INF/web.xml` file:

```
<context-param>
    <param-name>keycloak.config.resolver</param-name>
    <param-value>org.keycloak.adapters.osgi.PathBasedKey-
cloakConfigResolver</param-value>
</context-param>
```

That component uses `keycloak.config` or `karaf.etc` java properties to search for a base folder to locate the configuration. Then inside one of those folders it searches for a file called `<your_web_context>-keycloak.json`.

So, for example, if your web application has context `my-portal`, then your adapter configuration is loaded from the `$FUSE_HOME/etc/my-portal-keycloak.json` file.

## Securing a Servlet Deployed as an OSGI Service

You can use this method if you have a servlet class inside your OSGI bundled project that is not deployed as a classic WAR application. Fuse uses Pax Web Whiteboard Extender to deploy such servlets as web applications.

To secure your servlet with KeyCloak, complete the following steps:

1. KeyCloak provides PaxWebIntegrationService, which allows inject-

ting jetty-web.xml and configuring security constraints for your application. You need to declare such services in the `OSGI-INF/blueprint/blueprint.xml` file inside your application. Note that your servlet needs to depend on it. An example configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0
                               http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

    <!-- Using jetty bean just for the compatibility
        with other fuse services -->
    <bean id="servletConstraintMapping" class="org.eclipse.jetty.security.ConstraintMapping">
        <property name="constraint">
            <bean class="org.eclipse.jetty.util.security.Constraint">
                <property name="name" value="cst1"/>
                <property name="roles">
                    <list>
                        <value>user</value>
                    </list>
                </property>
                <property name="authenticate" value="true"/>
                <property name="dataConstraint" value="0"/>
            </bean>
        </property>
        <property name="pathSpec" value="/product-portal/*"/>
    </bean>

    <bean id="keycloakPaxWebIntegration" class="org.keycloak.adapters.osgi.PaxWebIntegrationService">
```

```

        init-method="start" destroy-method="stop">
        <property name="jettyWebXmlLocation"
value="/WEB-INF/jetty-web.xml" />
        <property name="bundleContext" ref="blueprint-
BundleContext" />
        <property name="constraintMappings">
            <list>
                <ref component-id="servletConstraint-
Mapping" />
            </list>
        </property>
    </bean>

    <bean id="productServlet" class="org.keycloak.
example.ProductPortalServlet" depends-on="keycloakPax-
WebIntegration">
    </bean>

    <service ref="productServlet" interface="javax.
servlet.Servlet">
        <service-properties>
            <entry key="alias" value="/product-portal"
/>
            <entry key="servlet-name" value="Product-
Servlet" />
            <entry key="keycloak.config.file"
value="/keycloak.json" />
        </service-properties>
    </service>

</blueprint>

```

- You might need to have the `WEB-INF` directory inside your project (even if your project is not a web application) and create the `/WEB-INF/jetty-web.xml` and `/WEB-INF/keycloak.json` files as in the [Classic WAR application](#) section. Note you don't need the `web.xml` file as the security-constraints are declared in the blueprint configuration file.

2. The `Import-Package` in `META-INF/MANIFEST.MF` must con-

tain at least these imports:

```
org.keycloak.adapters.jetty;version="{project_versionMvn}",
org.keycloak.adapters;version="{project_versionMvn}",
org.keycloak.constants;version="{project_versionMvn}",
org.keycloak.util;version="{project_versionMvn}",
org.keycloak.*;version="{project_versionMvn}",
*;resolution:=optional
```

## Securing an Apache Camel Application

You can secure Apache Camel endpoints implemented with the [camel-jetty](#) component by adding securityHandler with `KeycloakJettyAuthenticator` and the proper security constraints injected. You can add the `OSGI-INF/blueprint/blueprint.xml` file to your Camel application with a similar configuration as below. The roles, security constraint mappings, and KeyCloak adapter configuration might differ slightly depending on your environment and needs.

For example:

```
<?xml version="1.0" encoding="UTF-8"?>

<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:camel="http://camel.apache.org/schema/blueprint"
           xsi:schemaLocation="
               http://www.osgi.org/xmlns/blueprint/v1.0.0 http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
               http://camel.apache.org/schema/blueprint http://camel.apache.org/schema/blueprint/camel-blueprint.xsd">

    <bean id="kcAdapterConfig" class="org.keycloak.represe
```

```

ntations.adapters.config.AdapterConfig">
    <property name="realm" value="demo"/>
    <property name="resource" value="admin-camel-end-
point"/>
        <property name="bearerOnly" value="true"/>
        <property name="authServerUrl" value="http://local-
host:8080/auth" />
        <property name="sslRequired" value="EXTERNAL"/>
</bean>

    <bean id="keycloakAuthenticator" class="org.keycloak.
adapters.jetty.KeycloakJettyAuthenticator">
        <property name="adapterConfig" ref="kcAdapterCon-
fig"/>
    </bean>

    <bean id="constraint" class="org.eclipse.jetty.util.
security.Constraint">
        <property name="name" value="Customers"/>
        <property name="roles">
            <list>
                <value>admin</value>
            </list>
        </property>
        <property name="authenticate" value="true"/>
        <property name="dataConstraint" value="0"/>
    </bean>

    <bean id="constraintMapping" class="org.eclipse.jetty.
security.ConstraintMapping">
        <property name="constraint" ref="constraint"/>
        <property name="pathSpec" value="/*"/>
    </bean>

    <bean id="securityHandler" class="org.eclipse.jetty.
security.ConstraintSecurityHandler">
        <property name="authenticator" ref="keycloakAuthen-
ticator" />
        <property name="constraintMappings">
            <list>
                <ref component-id="constraintMapping" />
            </list>
        </property>
        <property name="authMethod" value="BASIC"/>

```

```

        <property name="realmName" value="does-not-mat-
ter"/>
    </bean>

    <bean id="sessionHandler" class="org.keycloak.adapters.
jetty.spi.WrappingSessionHandler">
        <property name="handler" ref="securityHandler" />
    </bean>

    <bean id="helloProcessor" class="org.keycloak.example.
CamelHelloProcessor" />

<camelContext id="blueprintContext"
              trace="false"
              xmlns="http://camel.apache.org/schema/
blueprint">
    <route id="httpBridge">
        <from uri="jetty:http://0.0.0.0:8383/admin-
camel-endpoint?handlers=sessionHandler&matchOnUriPre-
fix=true" />
        <process ref="helloProcessor" />
        <log message="The message from camel endpoint
contains ${body}" />
    </route>
</camelContext>

</blueprint>

```

- The `Import-Package` in `META-INF/MANIFEST.MF` needs to contain these imports:

```

javax.servlet;version="[3,4)",
javax.servlet.http;version="[3,4)",
org.apache.camel.*,
org.apache.camel;version="[2.13,3)",
org.eclipse.jetty.security;version="[9,10)",
org.eclipse.jetty.server.nio;version="[9,10)",
org.eclipse.jetty.util.security;version="[9,10)",
org.keycloak.*;version="{project_versionMvn}",
org.osgi.service.blueprint,
org.osgi.service.blueprint.container,

```

```
org.osgi.service.event,
```

## Camel RestDSL

Camel RestDSL is a Camel feature used to define your REST endpoints in a fluent way. But you must still use specific implementation classes and provide instructions on how to integrate with KeyCloak.

The way to configure the integration mechanism depends on the Camel component for which you configure your RestDSL-defined routes.

The following example shows how to configure integration using the Jetty component, with references to some of the beans defined in previous Blueprint example.

```
<bean id="securityHandlerRest" class="org.eclipse.jetty.  
security.ConstraintSecurityHandler">  
    <property name="authenticator" ref="keycloakAuthentica-  
tor" />  
    <property name="constraintMappings">  
        <list>  
            <ref component-id="constraintMapping" />  
        </list>  
    </property>  
    <property name="authMethod" value="BASIC"/>  
    <property name="realmName" value="does-not-matter"/>  
</bean>  
  
<bean id="sessionHandlerRest" class="org.keycloak.adapters.  
jetty.spi.WrappingSessionHandler">  
    <property name="handler" ref="securityHandlerRest" />  
</bean>  
  
<camelContext id="blueprintContext"  
    trace="false"  
    xmlns="http://camel.apache.org/schema/  
blueprint">
```

```

<restConfiguration component="jetty" context-
Path="/restdsl"
                      port="8484">
    <!--the link with Keycloak security handlers hap-
pens here-->
    <endpointProperty key="handlers" value="sessi-
onHandlerRest"></endpointProperty>
    <endpointProperty key="matchOnUriPrefix"
value="true"></endpointProperty>
</restConfiguration>

<rest path="/hello" >
    <description>Hello rest service</description>
    <get uri="/{id}" outType="java.lang.String">
        <description>Just an helllo</description>
        <to uri="direct:justDirect" />
    </get>

</rest>

<route id="justDirect">
    <from uri="direct:justDirect"/>
    <process ref="helloProcessor" />
    <log message="RestDSL correctly invoked ${body}"/>
    <setBody>
        <constant>(__This second sentence is returned
from a Camel RestDSL endpoint__)</constant>
    </setBody>
</route>

</camelContext>

```

## Securing an Apache CXF Endpoint on a Separate Jetty Engine

To run your CXF endpoints secured by KeyCloak on separate Jetty engines, complete the following steps:

1. Add `META-INF/spring/beans.xml` to your application, and in it, declare `httpj:engine-factory` with Jetty SecurityHandler with injected `KeycloakJettyAuthenticator`. The configuration

for a CFX JAX-WS application might resemble this one:

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/
beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:httpj="http://cxf.apache.org/transports/
http-jetty/configuration"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-
beans.xsd
           http://cxf.apache.org/jaxws http://cxf.apache.
org/schemas/jaxws.xsd
           http://www.springframework.org/schema/osgi
           http://www.springframework.org/schema/osgi/spring-osgi.
xsd
           http://cxf.apache.org/transports/http-jetty/con
figuration http://cxf.apache.org/schemas/configuration/
http-jetty.xsd">

    <import resource="classpath: META-INF/cxf/cxf.xml"
/>

    <bean id="kcAdapterConfig" class="org.keycloak.re
presentations.adapters.config.AdapterConfig">
        <property name="realm" value="demo"/>
        <property name="resource" value="custom-cxf-
endpoint"/>
        <property name="bearerOnly" value="true"/>
        <property name="authServerUrl"
value="http://localhost:8080/auth" />
        <property name="sslRequired" value="EXTERNAL"/>
    </bean>

    <bean id="keycloakAuthenticator" class="org.
keycloak.adapters.jetty.KeycloakJettyAuthenticator">
        <property name="adapterConfig">
            <ref local="kcAdapterConfig" />
        </property>
    </bean>
```

```

        </bean>

        <bean id="constraint" class="org.eclipse.jetty.
util.security.Constraint">
            <property name="name" value="Customers"/>
            <property name="roles">
                <list>
                    <value>user</value>
                </list>
            </property>
            <property name="authenticate" value="true"/>
            <property name="dataConstraint" value="0"/>
        </bean>

        <bean id="constraintMapping" class="org.eclipse.
jetty.security.ConstraintMapping">
            <property name="constraint" ref="constraint"/>
            <property name="pathSpec" value="/*"/>
        </bean>

        <bean id="securityHandler" class="org.eclipse.
jetty.security.ConstraintSecurityHandler">
            <property name="authenticator" ref="keycloakAu-
thenticator" />
            <property name="constraintMappings">
                <list>
                    <ref local="constraintMapping" />
                </list>
            </property>
            <property name="authMethod" value="BASIC"/>
            <property name="realmName" value="does-not-mat-
ter"/>
        </bean>

        <httpj:engine-factory bus="cxf" id="kc-cxf-end-
point">
            <httpj:engine port="8282">
                <httpj:handlers>
                    <ref local="securityHandler" />
                </httpj:handlers>
                <httpj:sessionSupport>true</httpj:sessionS-
upport>
            </httpj:engine>
        </httpj:engine-factory>
    
```

```

<jaxws:endpoint
    implementor="org.keycloak.example.
    ws.ProductImpl"
    address="http://localhost:8282/Pro-
    ductServiceCF" depends-on="kc-cxf-endpoint" />

</beans>

```

For the CXF JAX-RS application, the only difference might be in the configuration of the endpoint dependent on engine-factory:

```

<jaxrs:server serviceClass="org.keycloak.example.rs.
CustomerService" address="http://localhost:8282/rest"
depends-on="kc-cxf-endpoint">
<jaxrs:providers>
    <bean class="com.fasterxml.jackson.jaxrs.json.
JacksonJsonProvider" />
</jaxrs:providers>
</jaxrs:server>

```

2. The `Import-Package` in `META-INF/MANIFEST.MF` must contain those imports:

```

META-INF.cxf;version="[2.7,3.2)",
META-INF.cxf.osgi;version="[2.7,3.2)";resolution:=optional,
org.apache.cxf.bus;version="[2.7,3.2)",
org.apache.cxf.bus.spring;version="[2.7,3.2)",
org.apache.cxf.bus.resource;version="[2.7,3.2)",
org.apache.cxf.transport.http;version="[2.7,3.2)",
org.apache.cxf.*;version="[2.7,3.2)",
org.springframework.beans.factory.config,
org.eclipse.jetty.security;version="[9,10)",
org.eclipse.jetty.util.security;version="[9,10)",
org.keycloak.*;version="{project_versionMvn}"

```

## Securing an Apache CXF Endpoint on the Default Jetty Engine

Some services automatically come with deployed servlets on startup. One such service is the CXF servlet running in the http://localhost:8181/cxf context. Securing such endpoints can be complicated. One approach, which Keycloak is currently using, is ServletReregistrationService, which undeploys a built-in servlet at startup, enabling you to redeploy it on a context secured by Keycloak.

The configuration file `OSGI-INF/blueprint/blueprint.xml` inside your application might resemble the one below. Note that it adds the JAX-RS `customerservice` endpoint, which is endpoint-specific to your application, but more importantly, secures the entire `/cxf` context.

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxrs="http://cxf.apache.org/blueprint/jaxrs"
    xsi:schemaLocation="
        http://www.osgi.org/xmlns/blueprint/v1.0.0
        http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
        http://cxf.apache.org/blueprint/jaxrs
        http://cxf.apache.org/schemas/blueprint/jaxrs.xsd">

    <!-- JAXRS Application -->

    <bean id="customerBean" class="org.keycloak.example.rs.CxfCustomerService" />

    <jaxrs:server id="cxfJaxrsServer" address="/customerservice">
        <jaxrs:providers>
            <bean class="com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider" />
        </jaxrs:providers>
    </jaxrs:server>

```

```

<jaxrs:serviceBeans>
    <ref component-id="customerBean" />
</jaxrs:serviceBeans>
</jaxrs:server>

<!-- Securing of whole /cxf context by unregister de-
fault cxf servlet from paxweb and re-register with applied
security constraints -->

<bean id="cxfConstraintMapping" class="org.eclipse.
jetty.security.ConstraintMapping">
    <property name="constraint">
        <bean class="org.eclipse.jetty.util.security.
Constraint">
            <property name="name" value="cst1"/>
            <property name="roles">
                <list>
                    <value>user</value>
                </list>
            </property>
            <property name="authenticate"
value="true"/>
                <property name="dataConstraint" value="0"/>
            </bean>
        </property>
        <property name="pathSpec" value="/cxf/*"/>
    </bean>

    <bean id="cxfKeycloakPaxWebIntegration" class="org.
keycloak.adapters.osgi.PaxWebIntegrationService"
        init-method="start" destroy-method="stop">
        <property name="bundleContext" ref="blueprintBund-
leContext" />
        <property name="jettyWebXmlLocation" value="/WEB-
INF/jetty-web.xml" />
        <property name="constraintMappings">
            <list>
                <ref component-id="cxfConstraintMapping" />
            </list>
        </property>
    </bean>

    <bean id="defaultCxfReregistration" class="org.

```

```

keycloak.adapters.osgi.ServletReregistrationService" de-
pends-on="cxfKeycloakPaxWebIntegration"
    init-method="start" destroy-method="stop">
        <property name="bundleContext" ref="blueprintBund-
leContext" />
        <property name="managedServiceReference">
            <reference interface="org.osgi.service.cm.
ManagedService" filter="(service.pid=org.apache.cxf osgi)"
timeout="5000" />
        </property>
    </bean>

</blueprint>

```

As a result, all other CXF services running on the default CXF HTTP destination are also secured. Similarly, when the application is undeployed, the entire `/cxf` context becomes unsecured as well. For this reason, using your own Jetty engine for your applications as described in [Secure CXF Application on separate Jetty Engine](#) then gives you more control over security for each individual application.

- The `WEB-INF` directory might need to be inside your project (even if your project is not a web application). You might also need to edit the `/WEB-INF/jetty-web.xml` and `/WEB-INF/keycloak.json` files in a similar way as in [Classic WAR application](#). Note that you do not need the `web.xml` file as the security constraints are declared in the blueprint configuration file.
- The `Import-Package` in `META-INF/MANIFEST.MF` must contain these imports:

```

META-INF.cxf;version="[2.7,3.2)",
META-INF.cxf.osgi;version="[2.7,3.2)";resolution:=optional,
org.apache.cxf.transport.http;version="[2.7,3.2)",
org.apache.cxf.*;version="[2.7,3.2)",

```

```
com.fasterxml.jackson.jaxrs.json;version="[2.5,3)",  
org.eclipse.jetty.security;version="[9,10)",  
org.eclipse.jetty.util.security;version="[9,10)",  
org.keycloak.*;version="{project_versionMvn}",  
org.keycloak.adapters.jetty;version="{project_versionMvn}",  
*;resolution:=optional
```

## Securing Fuse Administration Services

### Using SSH Authentication to Fuse Terminal

KeyCloak mainly addresses use cases for authentication of web applications; however, if your other web services and applications are protected with KeyCloak, protecting non-web administration services such as SSH with KeyCloak credentials is a best practice. You can do this using the JAAS login module, which allows remote connection to KeyCloak and verifies credentials based on [Resource Owner Password Credentials](#).

To enable SSH authentication, complete the following steps:

1. In KeyCloak create a client (for example, `ssh-jmx-admin-client`), which will be used for SSH authentication. This client needs to have `Direct Access Grants Enabled` selected to `On`.
2. In the `$FUSE_HOME/etc/org.apache.karaf.shell.cfg` file, update or specify this property:

```
sshRealm=keycloak
```

3. Add the `$FUSE_HOME/etc/keycloak-direct-access.json` file with content similar to the following (based on your environment and KeyCloak client settings):

```
{  
    "realm": "demo",  
    "resource": "ssh-jmx-admin-client",  
    "ssl-required" : "external",  
    "auth-server-url" : "http://localhost:8080/auth",  
    "credentials": {  
        "secret": "password"  
    }  
}
```

This file specifies the client application configuration, which is used by JAAS DirectAccessGrantsLoginModule from the `keycloak` JAAS realm for SSH authentication.

4. Start Fuse and install the `keycloak` JAAS realm. The easiest way is to install the `keycloak-jaas` feature, which has the JAAS realm predefined. You can override the feature's predefined realm by using your own `keycloak` JAAS realm with higher ranking. For details see the [JBoss Fuse documentation](#).

Use these commands in the Fuse terminal:

```
features:addurl mvn:org.keycloak/keycloak-osgi-features/{project_versionMvn}/xml/features  
features:install keycloak-jaas
```

5. Log in using SSH as `admin` user by typing the following in the terminal:

```
ssh -o PubkeyAuthentication=no -p 8101 admin@localhost
```

6. Log in with password `password`.

On some later operating systems, you might also need

to use the SSH command's -o option `-o HostKeyAlgorithms=+ssh-dss` because later SSH clients do not allow use of the `ssh-dss` algorithm, by default. However, by default, it is currently used in {fuseVersion}.

Note that the user needs to have realm role `admin` to perform all operations or another role to perform a subset of operations (for example, the **viewer** role that restricts the user to run only read-only Karaf commands). The available roles are configured in `$FUSE_HOME/etc/org.apache.karaf.shell.cfg` or `$FUSE_HOME/etc/system.properties`.

## Using JMX Authentication

JMX authentication might be necessary if you want to use jconsole or another external tool to remotely connect to JMX through RMI. Otherwise it might be better to use hawt.io/jolokia, since the jolokia agent is installed in hawt.io by default. For more details see [Hawtio Admin Console](#).

To use JMX authentication, complete the following steps:

1. In the `$FUSE_HOME/etc/org.apache.karaf.management.cfg` file, change the `jmxRealm` property to:

```
jmxRealm=keycloak
```

2. Install the `keycloak-jaas` feature and configure the `$FUSE_HOME/etc/keycloak-direct-access.json` file as described in

the SSH section above.

3. In jconsole you can use a URL such as:

```
service:jmx:rmi://localhost:4444/jndi/rmi://localhost:1099/karaf-root
```

and credentials: admin/password (based on the user with admin privileges according to your environment).

## Securing the Hawtio Administration Console

To secure the Hawtio Administration Console with KeyCloak, complete the following steps:

1. Add these properties to the `$FUSE_HOME/etc/system.properties` file:

```
hawtio.keycloakEnabled=true
hawtio.realm=keycloak
hawtio.keycloakClientConfig=file://${karaf.base}/etc/keycloak-hawtio-client.json
hawtio.rolePrincipalClasses=org.keycloak.adapters.jaas.RolePrincipal,org.apache.karaf.jaas.boot.principal.RolePrincipal
```

2. Create a client in the KeyCloak administration console in your realm. For example, in the KeyCloak `demo` realm, create a client `hawtio-client`, specify `public` as the Access Type, and specify a redirect URI pointing to Hawtio: `http://localhost:8181/hawtio/*`. You must also have a corresponding Web Origin configured (in this case, `http://localhost:8181`).
3. Create the `keycloak-hawtio-client.json` file in the `$FU-`

`SE_HOME/etc` directory using content similar to that shown in the example below. Change the `realm`, `resource`, and `auth-server-url` properties according to your Keycloak environment. The `resource` property must point to the client created in the previous step. This file is used by the client (Hawtio JavaScript application) side.

```
{  
  "realm" : "demo",  
  "resource" : "hawtio-client",  
  "auth-server-url" : "http://localhost:8080/auth",  
  "ssl-required" : "external",  
  "public-client" : true  
}
```

4. Create the `keycloak-hawtio.json` file in the `$FUSE_HOME/etc` directory using content similar to that shown in the example below. Change the `realm` and `auth-server-url` properties according to your Keycloak environment. This file is used by the adapters on the server (JAAS Login module) side.

```
{  
  "realm" : "demo",  
  "resource" : "jaas",  
  "bearer-only" : true,  
  "auth-server-url" : "http://localhost:8080/auth",  
  "ssl-required" : "external",  
  "use-resource-role-mappings": false,  
  "principal-attribute": "preferred_username"  
}
```

5. Start `{fuseVersion}` and install the keycloak feature if you have not already done so. The commands in Karaf terminal are similar to this example:

```
features:addurl mvn:org.keycloak/keycloak-osgi-features/{project_versionMvn}/xml/features  
features:install keycloak
```

6. Go to <http://localhost:8181/hawtio> and log in as a user from your KeyCloak realm.

Note that the user needs to have the proper realm role to successfully authenticate to Hawtio. The available roles are configured in the `$FUSE_HOME/etc/system.properties` file in `hawtio.roles`.

## Securing Hawtio on {fuseHawtioEAPVersion}

To run Hawtio on the {fuseHawtioEAPVersion} server, complete the following steps:

1. Set up KeyCloak as described in the previous section, Securing the Hawtio Administration Console. It is assumed that:
  - you have a KeyCloak realm `demo` and client `hawtio-client`
  - your KeyCloak is running on `localhost:8080`
  - the {fuseHawtioEAPVersion} server with deployed Hawtio will be running on `localhost:8181`. The directory with this server is referred in next steps as `$EAP_HOME`.
2. Copy the {fuseHawtioWARVersion} archive to the `$EAP_HOME/standalone/configuration` directory. For more details about deploying Hawtio see the [Fuse Hawtio documentation](#).
3. Copy the `keycloak-hawtio.json` and `keycloak-hawtio-client.json` files with the above content to the `$EAP_HOME/standalone/configuration` directory.

4. Install the KeyCloak adapter subsystem to your {fuseHawtioEAP-Version} server as described in the [JBoss adapter documentation](#).
5. In the `$EAP_HOME/standalone/configuration/standalone.xml` file configure the system properties as in this example:

```

<extensions>
  ...
</extensions>

<system-properties>
  <property name="hawtio.authenticationEnabled" value="true" />
  <property name="hawtio.realm" value="hawtio" />
  <property name="hawtio.roles" value="admin,viewer" />
  <property name="hawtio.rolePrincipalClasses" value="org.keycloak.adapters.jaas.BeaconRolePrincipal" />
  <property name="hawtio.keycloakEnabled" value="true" />
  <property name="hawtio.keycloakClientConfig" value="$KEYCLOAK_CLIENT_CONFIG" />
  <property name="hawtio.keycloakServerConfig" value="$KEYCLOAK_SERVER_CONFIG" />
</system-properties>

```

6. Add the Hawtio realm to the same file in the `security-domains` section:

```

<security-domain name="hawtio" cache-type="default">
  <authentication>
    <login-module code="org.keycloak.adapters.jaas.BeaconLoginModule" optional="true">
      <module-option name="keycloak-config-file" value="keycloak.json" />
    </login-module>
  </authentication>
</security-domain>

```

7. Add the `secure-deployment` section `hawtio` to the adapter subsystem. This ensures that the Hawtio WAR is able to find the JAAS login module classes.

```
<subsystem xmlns="urn:jboss:domain:keycloak:1.1">
    <secure-deployment name="{fuseHawtioWARVersion}" />
</subsystem>
```

8. Restart the {fuseHawtioEAPVersion} server with Hawtio:

```
cd $EAP_HOME/bin
./standalone.sh -Djboss.socket.binding.port-offset=101
```

9. Access Hawtio at <http://localhost:8181/hawtio>. It is secured by KeyCloak.

#### 2.1.4. JBoss Fuse 7 Adapter

KeyCloak supports securing your web applications running inside [JBoss Fuse 7](#).

JBoss Fuse 7 leverages Undertow adapter which is essentially the same as {fuse7Version} is bundled with [Undertow HTTP engine](#) under the covers and Undertow is used for running various kinds of web applications.

The only supported version of Fuse 7 is the latest release. If you use earlier versions of Fuse 7, it is possible that some functions will not work correctly. In particular, integration will not work at all for versions of Fuse 7 lower than 7.0.1.

Security for the following items is supported for Fuse:

- Classic WAR applications deployed on Fuse with Pax Web War Ex-

tender

- Servlets deployed on Fuse as OSGI services with Pax Web Whiteboard Extender and additionally servlets registered through org.osgi.service.http.HttpService#registerServlet() which is standard OSGI Enterprise HTTP Service
- [Apache Camel](#) Undertow endpoints running with the [Camel Undertow](#) component
- [Apache CXF](#) endpoints running on their own separate Undertow engine
- [Apache CXF](#) endpoints running on the default engine provided by the CXF servlet
- SSH and JMX admin access
- [Hawtio administration console](#)

## Securing Your Web Applications Inside Fuse 7

You must first install the Keycloak Karaf feature. Next you will need to perform the steps according to the type of application you want to secure. All referenced web applications require injecting the Keycloak Undertow authentication mechanism into the underlying web server. The steps to achieve this depend on the application type. The details are described below.

### Installing the Keycloak Feature

You must first install the `keycloak-pax-http-undertow` and `keycloak-jaas` features in the JBoss Fuse environment. The `keycloak-pax-http-undertow` feature includes the Fuse adapter and all third-party dependencies. The `keycloak-jaas` contains JAAS module used

in realm for SSH and JMX authentication. You can install it either from the Maven repository or from an archive.

## Installing from the Maven Repository

As a prerequisite, you must be online and have access to the Maven repository.

To install the keycloak feature using the Maven repository, complete the following steps:

1. Start {fuse7Version}; then in the Karaf terminal type:

```
feature:repo-add mvn:org.keycloak/keycloak-osgi-features/{project_versionMvn}/xml/features  
feature:install keycloak-pax-http-undertow keycloak-jaas
```

2. You might also need to install the Undertow feature:

```
feature:install pax-http-undertow
```

3. Ensure that the features were installed:

```
feature:list | grep keycloak
```

## Installing from the ZIP bundle

This is useful if you are offline or do not want to use Maven to obtain the JAR files and other artifacts.

To install the Fuse adapter from the ZIP archive, complete the following steps:

1. Download the KeyCloak Fuse adapter ZIP archive.
2. Unzip it into the root directory of JBoss Fuse. The dependencies are then installed under the `system` directory. You can overwrite all existing jar files.

Use this for {fuse7Version}:

```
cd /path-to-fuse/fuse-karaf-7.z
```

3. Start Fuse and run these commands in the fuse/karaf terminal:

```
feature:repo-add mvn:org.keycloak/keycloak-osgi-features/{project_versionMvn}/xml/features  
feature:install keycloak-pax-http-undertow keycloak-jaas
```

4. Install the corresponding Undertow adapter. Since the artifacts are available directly in the JBoss Fuse `system` directory, you do not need to use the Maven repository.

## Securing a Classic WAR Application

The needed steps to secure your WAR application are:

1. In the `/WEB-INF/web.xml` file, declare the necessary:
  - security constraints in the `<security-constraint>` element
  - login configuration in the `<login-config>` element. Make sure that the `<auth-method>` is `KEYCLOAK`.
  - security roles in the `<security-role>` element

For example:

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/
xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_3_0.xsd"
          version="3.0">

    <module-name>customer-portal</module-name>

    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
    </welcome-file-list>

    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Customers</web-re-
source-name>
            <url-pattern>/customers/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>user</role-name>
        </auth-constraint>
    </security-constraint>

    <login-config>
        <auth-method>KEYCLOAK</auth-method>
        <realm-name>does-not-matter</realm-name>
    </login-config>

    <security-role>
        <role-name>admin</role-name>
    </security-role>
    <security-role>
        <role-name>user</role-name>
    </security-role>
</web-app>

```

2. Within the `/WEB-INF/` directory of your WAR, create a new file, `keycloak.json`. The format of this configuration file is described in

the [Java Adapters Config](#) section. It is also possible to make this file available externally as described in [Configuring the External Adapter](#).

For example:

```
{  
    "realm": "demo",  
    "resource": "customer-portal",  
    "auth-server-url": "http://localhost:8080/auth",  
    "ssl-required" : "external",  
    "credentials": {  
        "secret": "password"  
    }  
}
```

3. Contrary to the Fuse 6 adapter, there are no special OSGi imports needed in MANIFEST.MF.

## Configuration Resolvers

The `keycloak.json` adapter configuration file can be stored inside a bundle, which is default behaviour, or in a directory on a filesystem. To specify the actual source of the configuration file, set the `keycloak.config.resolver` deployment parameter to the desired configuration resolver class. For example, in a classic WAR application, set the `keycloak.config.resolver` context parameter in `web.xml` file like this:

```
<context-param>  
    <param-name>keycloak.config.resolver</param-name>  
    <param-value>org.keycloak.adapters.osgi.PathBasedKey-  
cloakConfigResolver</param-value>  
</context-param>
```

The following resolvers are available for `keycloak.config`.

`resolver`:

### **org.keycloak.adapters.osgi.BundleBasedKeycloakConfigResolver**

This is the default resolver. The configuration file is expected inside the OSGi bundle that is being secured. By default, it loads file named `WEB-INF/keycloak.json` but this file name can be configured via `configLocation` property.

### **org.keycloak.adapters.osgi.PathBasedKeycloakConfigResolver**

This resolver searches for a file called `<your_web_context>-keycloak.json` inside a folder that is specified by `keycloak.config` system property. If `keycloak.config` is not set, `karaf.etc` system property is used instead.

For example, if your web application is deployed into context `my-portal`, then your adapter configuration would be loaded either from the  `${keycloak.config}/my-portal-keycloak.json` file, or from  `${karaf.etc}/my-portal-keycloak.json`.

### **org.keycloak.adapters.osgi.HierarchicalPathBasedKeycloakConfigResolver**

This resolver is similar to `PathBasedKeycloakConfigResolver` above, where for given URI path, configuration locations are checked from most to least specific.

For example, for `/my/web-app/context` URI, the following configuration locations are searched for existence until the first one exists:

- \${karaf.etc}/my-web-app-context-keycloak.json
- \${karaf.etc}/my-web-app-keycloak.json
- \${karaf.etc}/my-keycloak.json
- \${karaf.etc}/keycloak.json

## Securing a Servlet Deployed as an OSGI Service

You can use this method if you have a servlet class inside your OSGI bundled project that is not deployed as a classic WAR application. Fuse uses Pax Web Whiteboard Extender to deploy such servlets as web applications.

To secure your servlet with KeyCloak, complete the following steps:

1. KeyCloak provides `org.keycloak.adapters.osgi.undertow.PaxWebIntegrationService`, which allows configuring authentication method and security constraints for your application. You need to declare such services in the `OSGI-INF/blueprint/blueprint.xml` file inside your application. Note that your servlet needs to depend on it. An example configuration:

```

<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0 http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

    <bean id="servletConstraintMapping" class="org.keycloak.adapters.osgi.PaxWebSecurityConstraintMapping">
        <property name="roles">

```

```

        <list>
            <value>user</value>
        </list>
    </property>
    <property name="authentication" value="true"/>
    <property name="url" value="/product-por-
tal/*"/>
</bean>

        <!-- This handles the integration and setting the
login-config and security-constraints parameters -->
<bean id="keycloakPaxWebIntegration" class="org.
keycloak.adapters.osgi.undertow.PaxWebIntegrationSer-
vice"
        init-method="start" destroy-method="stop">
    <property name="bundleContext" ref="blueprint-
BundleContext" />
    <property name="constraintMappings">
        <list>
            <ref component-id="servletConstraint-
Mapping" />
        </list>
    </property>
</bean>

<bean id="productServlet" class="org.keycloak.
example.ProductPortalServlet" depends-on="keycloakPax-
WebIntegration" />

        <service ref="productServlet" interface="javax.
servlet.Servlet">
            <service-properties>
                <entry key="alias" value="/product-portal"
/>
                <entry key="servlet-name" value="Product-
Servlet" />
                <entry key="keycloak.config.file"
value="/keycloak.json" />
            </service-properties>
        </service>
</blueprint>
```

- You might need to have the `WEB-INF` directory inside your project (even if your project is not a web application) and create the `/WEB-INF/keycloak.json` file as described in the [Classic WAR application](#) section. Note you don't need the `web.xml` file as the security-constraints are declared in the blueprint configuration file.
2. Contrary to the Fuse 6 adapter, there are no special OSGi imports needed in `MANIFEST.MF`.

## Securing an Apache Camel Application

You can secure Apache Camel endpoints implemented with the [camel-undertow](#) component by injecting the proper security constraints via blueprint and updating the used component to `undertow-keycloak`. You have to add the `OSGI-INF/blueprint/blueprint.xml` file to your Camel application with a similar configuration as below. The roles and security constraint mappings, and adapter configuration might differ slightly depending on your environment and needs.

Compared to the standard `undertow` component, `undertow-keycloak` component adds two new properties:

- `configResolver` is a resolver bean that supplies Keycloak adapter configuration. Available resolvers are listed in [Configuration Resolvers](#) section.
- `allowedRoles` is a comma-separated list of roles. User accessing the service has to have at least one role to be permitted the access.

For example:

```

<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.
0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
           xmlns:camel="http://camel.apache.org/schema/
blueprint"
           xsi:schemaLocation="
               http://www.osgi.org/xmlns/blueprint/v1.0.0 http://
www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
               http://camel.apache.org/schema/blueprint http://
camel.apache.org/schema/blueprint/camel-blueprint-2.17.1.
xsd">

    <bean id="keycloakConfigResolver" class="org.keycloak.
adapters.osgi.BundleBasedKeycloakConfigResolver" >
        <property name="bundleContext" ref="blueprintBund-
leContext" />
    </bean>

    <bean id="helloProcessor" class="org.keycloak.example.
CamelHelloProcessor" />

    <camelContext id="blueprintContext"
                  trace="false"
                  xmlns="http://camel.apache.org/schema/
blueprint">

        <route id="httpBridge">
            <from uri="undertow-keycloak:http://0.0.0.0:
8383/admin-camel-endpoint?matchOnUriPrefix=true&config-
Resolver=#keycloakConfigResolver&allowedRoles=admin" />
            <process ref="helloProcessor" />
            <log message="The message from camel endpoint
contains ${body}"/>
        </route>

    </camelContext>

</blueprint>

```

- The `Import-Package` in `META-INF/MANIFEST.MF` needs to con-

tain these imports:

```
javax.servlet;version="[3,4)",
javax.servlet.http;version="[3,4)",
javax.net.ssl,
org.apache.camel.*,
org.apache.camel;version="[2.13,3)",
io.undertow.*,
org.keycloak.*;version="{project_versionMvn}",
org.osgi.service.blueprint,
org.osgi.service.blueprint.container
```

## Camel RestDSL

Camel RestDSL is a Camel feature used to define your REST endpoints in a fluent way. But you must still use specific implementation classes and provide instructions on how to integrate with KeyCloak.

The way to configure the integration mechanism depends on the Camel component for which you configure your RestDSL-defined routes.

The following example shows how to configure integration using the `undertow-keycloak` component, with references to some of the beans defined in previous Blueprint example.

```
<camelContext id="blueprintContext"
              trace="false"
              xmlns="http://camel.apache.org/schema/
blueprint">

    <!--the link with Keycloak security handlers happens by
using undertow-keycloak component -->
    <restConfiguration apiComponent="undertow-keycloak"
contextPath="/restdsl" port="8484">
        <endpointProperty key="configResolver" value="#key-
cloakConfigResolver" />
```

```

        <endpointProperty key="allowedRoles"
value="admin,superadmin" />
    </restConfiguration>

    <rest path="/hello" >
        <description>Hello rest service</description>
        <get uri="/{id}" outType="java.lang.String">
            <description>Just a hello</description>
            <to uri="direct:justDirect" />
        </get>

    </rest>

    <route id="justDirect">
        <from uri="direct:justDirect"/>
        <process ref="helloProcessor" />
        <log message="RestDSL correctly invoked ${body}"/>
        <setBody>
            <constant>(__This second sentence is returned
from a Camel RestDSL endpoint__)</constant>
        </setBody>
    </route>

</camelContext>

```

## Securing an Apache CXF Endpoint on a Separate Undertow Engine

To run your CXF endpoints secured by Keycloak on a separate Undertow engine, complete the following steps:

1. Add `OSGI-INF/blueprint/blueprint.xml` to your application, and in it, add the proper configuration resolver bean similarly to [Camel configuration](#). In the `httpu:engine-factory` declare `org.keycloak.adapters.osgi.undertow.CxfKeycloakAuthHandler` handler using that camel configuration. The configuration for a CXF JAX-WS application might resemble this one:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:jaxws="http://cxf.apache.org/blueprint/jaxws"
           xmlns:cxf="http://cxf.apache.org/blueprint/core"
           xmlns:httpu="http://cxf.apache.org/transports/http-undertow/configuration".
           xsi:schemaLocation="
               http://cxf.apache.org/transports/http-undertow/configuration
               http://cxf.apache.org/schemas/configuration/http-undertow.xsd
               http://cxf.apache.org/blueprint/core http://cxf.
               apache.org/schemas/blueprint/core.xsd
               http://cxf.apache.org/blueprint/jaxws http://cxf.
               apache.org/schemas/blueprint/jaxws.xsd">

    <bean id="keycloakConfigResolver" class="org.
    keycloak.adapters.osgi.BundleBasedKeycloakConfigResol-
    ver" >
        <property name="bundleContext" ref="blueprint-
    BundleContext" />
    </bean>

    <httpu:engine-factory bus="cxf" id="kc-cxf-end-
    point">
        <httpu:engine port="8282">
            <httpu:handlers>
                <bean class="org.keycloak.adapters.
    osgi.undertow.CxfKeycloakAuthHandler">
                    <property name="configResolver"
ref="keycloakConfigResolver" />
                </bean>
            </httpu:handlers>
        </httpu:engine>
    </httpu:engine-factory>

    <jaxws:endpoint implementor="org.keycloak.example.
    ws.ProductImpl"
        address="http://localhost:8282/Pro-
    ductServiceCF" depends-on="kc-cxf-endpoint"/>

```

```
</blueprint>
```

For the CXF JAX-RS application, the only difference might be in the configuration of the endpoint dependent on engine-factory:

```
<jaxrs:server serviceClass="org.keycloak.example.rs.CustomerService" address="http://localhost:8282/rest"
depends-on="kc-cxf-endpoint">
<jaxrs:providers>
<bean class="com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider" />
</jaxrs:providers>
</jaxrs:server>
```

2. The `Import-Package` in `META-INF/MANIFEST.MF` must contain those imports:

```
META-INF.cxf;version="[2.7,3.3)",
META-INF.cxf.osgi;version="[2.7,3.3)";resolution:=optional,
org.apache.cxf.bus;version="[2.7,3.3)",
org.apache.cxf.bus.spring;version="[2.7,3.3)",
org.apache.cxf.bus.resource;version="[2.7,3.3)",
org.apache.cxf.transport.http;version="[2.7,3.3)",
org.apache.cxf.*;version="[2.7,3.3)",
org.springframework.beans.factory.config,
org.keycloak.*;version="{project_versionMvn}"
```

## Securing an Apache CXF Endpoint on the Default Undertow Engine

Some services automatically come with deployed servlets on startup. One such service is the CXF servlet running in the `http://localhost:8181/cxf` context. Fuse's Pax Web supports altering existing contexts via configuration admin. This can be used to secure endpoints by KeyCloak.

The configuration file `OSGI-INF/blueprint/blueprint.xml` inside your application might resemble the one below. Note that it adds the JAX-RS `customerservice` endpoint, which is endpoint-specific to your application.

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:jaxrs="http://cxf.apache.org/blueprint/jaxrs"
           xsi:schemaLocation="
               http://www.osgi.org/xmlns/blueprint/v1.0.0
               http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
               http://cxf.apache.org/blueprint/jaxrs
               http://cxf.apache.org/schemas/blueprint/jaxrs.xsd">

    <!-- JAXRS Application -->
    <bean id="customerBean" class="org.keycloak.example.rs.CxfCustomerService" />

    <jaxrs:server id="cxfJaxrsServer" address="/customerservice">
        <jaxrs:providers>
            <bean class="com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider" />
        </jaxrs:providers>
        <jaxrs:serviceBeans>
            <ref component-id="customerBean" />
        </jaxrs:serviceBeans>
    </jaxrs:server>
</blueprint>
```

Furthermore, you have to create  `${karaf.etc}/org.ops4j.pax.web.context-anyName.cfg` file. It will be treated as factory PID configuration that is tracked by `pax-web-runtime` bundle. Such configuration may contain the following properties that correspond to some

of the properties of standard `web.xml`:

```
bundle.symbolicName = org.apache.cxf.cxf-rt-transports-http
context.id = default

context.param.keycloak.config.resolver = org.keycloak.
adapters.osgi.HierarchicalPathBasedKeycloakConfigResolver

login.config.authMethod = KEYCLOAK

security.cxf.url = /cxf/customerservice/*
security.cxf.roles = admin, user
```

For full description of available properties in configuration admin file, please refer to Fuse documentation. The properties above have the following meaning:

`bundle.symbolicName` and `context.id`

Identification of the bundle and its deployment context within `org.ops4j.pax.web.service.WebContainer`.

`context.param.keycloak.config.resolver`

Provides value of `keycloak.config.resolver` context parameter to the bundle just the same as in `web.xml` for classic WARs. Available resolvers are described in [Configuration Resolvers](#) section.

`login.config.authMethod`

Authentication method. Must be `KEYCLOAK`.

`security.anyName.url` and `security.anyName.roles`

Values of properties of individual security constraints just as they would be set in `security-constraint/web-resource-collec-`

`tion/url-pattern` and `security-constraint/auth-constraint/role-name` in `web.xml`, respectively. Roles are separated by comma and whitespace around it. The `anyName` identifier can be arbitrary but must match for individual properties of the same security constraint.

Some Fuse versions contain a bug that requires roles to be separated by `", "` (comma and single space). Make sure you use precisely this notation for separating the roles.

The `Import-Package` in `META-INF/MANIFEST.MF` must contain at least these imports:

```
javax.ws.rs;version="[2,3)",
META-INF.cxf;version="[2.7,3.3)",
META-INF.cxf.osgi;version="[2.7,3.3)";resolution:=optional,
org.apache.cxf.transport.http;version="[2.7,3.3)",
org.apache.cxf.*;version="[2.7,3.3)",
com.fasterxml.jackson.jaxrs.json;version="${jackson.version}"
```

## Securing Fuse Administration Services

### Using SSH Authentication to Fuse Terminal

Keycloak mainly addresses use cases for authentication of web applications; however, if your other web services and applications are protected with Keycloak, protecting non-web administration services such as SSH with Keycloak credentials is a best practice. You can do this using the JAAS login module, which allows remote connection to Keycloak and verifies credentials based on [Resource Owner Password Credentials](#).

[als.](#)

To enable SSH authentication, complete the following steps:

1. In KeyCloak create a client (for example, `ssh-jmx-admin-client`), which will be used for SSH authentication. This client needs to have `Direct Access Grants Enabled` selected to `On`.
2. In the `$FUSE_HOME/etc/org.apache.karaf.shell.cfg` file, update or specify this property:

```
sshRealm=keycloak
```

3. Add the `$FUSE_HOME/etc/keycloak-direct-access.json` file with content similar to the following (based on your environment and KeyCloak client settings):

```
{
    "realm": "demo",
    "resource": "ssh-jmx-admin-client",
    "ssl-required" : "external",
    "auth-server-url" : "http://localhost:8080/auth",
    "credentials": {
        "secret": "password"
    }
}
```

This file specifies the client application configuration, which is used by JAAS DirectAccessGrantsLoginModule from the `keycloak` JAAS realm for SSH authentication.

4. Start Fuse and install the `keycloak` JAAS realm. The easiest way is to install the `keycloak-jaas` feature, which has the JAAS

realm predefined. You can override the feature's predefined realm by using your own `keycloak` JAAS realm with higher ranking. For details see the [JBoss Fuse documentation](#).

Use these commands in the Fuse terminal:

```
features:addurl mvn:org.keycloak/keycloak-osgi-features/{project_versionMvn}/xml/features  
features:install keycloak-jaas
```

5. Log in using SSH as `admin` user by typing the following in the terminal:

```
ssh -o PubkeyAuthentication=no -p 8101 admin@localhost
```

6. Log in with password `password`.

On some later operating systems, you might also need to use the SSH command's `-o` option `-o HostKeyAlgorithms=+ssh-dss` because later SSH clients do not allow use of the `ssh-dss` algorithm, by default. However, by default, it is currently used in {fuse7Version}.

Note that the user needs to have realm role `admin` to perform all operations or another role to perform a subset of operations (for example, the `viewer` role that restricts the user to run only read-only Karaf commands). The available roles are configured in `$FUSE_HOME/etc/org.apache.karaf.shell.cfg` or `$FUSE_HOME/etc/system.properties`.

## Using JMX Authentication

JMX authentication might be necessary if you want to use jconsole or another external tool to remotely connect to JMX through RMI. Otherwise it might be better to use hawt.io/jolokia, since the jolokia agent is installed in hawt.io by default. For more details see [Hawtio Admin Console](#).

To use JMX authentication, complete the following steps:

1. In the `$FUSE_HOME/etc/org.apache.karaf.management.cfg` file, change the `jmxRealm` property to:

```
jmxRealm=keycloak
```

2. Install the `keycloak-jaas` feature and configure the `$FUSE_HOME/etc/keycloak-direct-access.json` file as described in the SSH section above.
3. In jconsole you can use a URL such as:

```
service:jmx:rmi://localhost:4444/jndi/rmi://localhost:1099/karaf-root
```

and credentials: admin/password (based on the user with admin privileges according to your environment).

## Securing the Hawtio Administration Console

To secure the Hawtio Administration Console with KeyCloak, complete the following steps:

1. Create a client in the KeyCloak administration console in your realm. For example, in the KeyCloak `demo` realm, create a client `hawtio-client`, specify `public` as the Access Type, and specify a redirect URI pointing to Hawtio: `http://localhost:8181/hawtio/*`. Configure corresponding Web Origin (in this case, `http://localhost:8181`). Setup client scope mapping to include *view-profile* client role of *account* client in *Scope* tab in `hawtio-client` client detail.
2. Create the `keycloak-hawtio-client.json` file in the `$FUSION_HOME/etc` directory using content similar to that shown in the example below. Change the `realm`, `resource`, and `auth-server-url` properties according to your KeyCloak environment. The `resource` property must point to the client created in the previous step. This file is used by the client (Hawtio JavaScript application) side.

```
{  
  "realm" : "demo",  
  "clientId" : "hawtio-client",  
  "url" : "http://localhost:8080/auth",  
  "ssl-required" : "external",  
  "public-client" : true  
}
```

3. Create the `keycloak-direct-access.json` file in the `$FUSION_HOME/etc` directory using content similar to that shown in the example below. Change the `realm` and `url` properties according to your KeyCloak environment. This file is used by JavaScript client.

```
{
```

```

"realm" : "demo",
"resource" : "ssh-jmx-admin-client",
"auth-server-url" : "http://localhost:8080/auth",
"ssl-required" : "external",
"credentials": {
    "secret": "password"
}
}

```

4. Create the `keycloak-hawtio.json` file in the `$FUSE_HOME/etc` directory using content similar to that shown in the example below. Change the `realm` and `auth-server-url` properties according to your Keycloak environment. This file is used by the adapters on the server (JAAS Login module) side.

```

{
    "realm" : "demo",
    "resource" : "jaas",
    "bearer-only" : true,
    "auth-server-url" : "http://localhost:8080/auth",
    "ssl-required" : "external",
    "use-resource-role-mappings": false,
    "principal-attribute": "preferred_username"
}

```

5. Start {fuse7Version}, [install the Keycloak feature](#). Then type in the Karaf terminal:

```

system:property -p hawtio.keycloakEnabled true
system:property -p hawtio.realm keycloak
system:property -p hawtio.keycloakClientConfig
file:///${karaf.base}/etc/keycloak-hawtio-client.json
system:property -p hawtio.rolePrincipalClasses org.
keycloak.adapters.jaas.RolePrincipal,org.apache.karaf.
jaas.boot.principal.RolePrincipal
restart io.hawt.hawtio-war

```

6. Go to <http://localhost:8181/hawtio> and log in as a user from your Keycloak realm.

Note that the user needs to have the proper realm role to successfully authenticate to Hawtio. The available roles are configured in the `$FUSE_HOME/etc/system.properties` file in `hawtio.roles`.

### 2.1.5. Spring Boot Adapter

To be able to secure Spring Boot apps you must add the Keycloak Spring Boot adapter JAR to your app. You then have to provide some extra configuration via normal Spring Boot configuration (`application.properties`). Let's go over these steps.

#### Adapter Installation

The Keycloak Spring Boot adapter takes advantage of Spring Boot's autoconfiguration so all you need to do is add the Keycloak Spring Boot starter to your project. The Keycloak Spring Boot Starter is also directly available from the [Spring Start Page](#). To add it manually using Maven, add the following to your dependencies:

```
<dependency>
    <groupId>org.keycloak</groupId>
    <artifactId>keycloak-spring-boot-starter</artifactId>
</dependency>
```

Add the Adapter BOM dependency:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.keycloak.bom</groupId>
            <artifactId>keycloak-adapter-bom</artifactId>
```

```
<version>{project_versionMvn}</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

Currently the following embedded containers are supported and do not require any extra dependencies if using the Starter:

- Tomcat
- Undertow
- Jetty

## Required Spring Boot Adapter Configuration

This section describes how to configure your Spring Boot app to use Keycloak.

Instead of a `keycloak.json` file, you configure the realm for the Spring Boot Keycloak adapter via the normal Spring Boot configuration. For example:

```
keycloak.realm = demorealm
keycloak.auth-server-url = http://127.0.0.1:8080/auth
keycloak.ssl-required = external
keycloak.resource = demoapp
keycloak.credentials.secret = 11111111-1111-1111-1111-
111111111111
keycloak.use-resource-role-mappings = true
```

You can disable the Keycloak Spring Boot Adapter (for example in tests) by setting `keycloak.enabled = false`.

To configure a Policy Enforcer, unlike `keycloak.json`, `policy-enforcer-config` must be used instead of just `policy-enforcer`.

You also need to specify the Java EE security config that would normally go in the `web.xml`. The Spring Boot Adapter will set the `login-method` to `KEYCLOAK` and configure the `security-constraints` at startup time. Here's an example configuration:

```
keycloak.securityConstraints[0].authRoles[0] = admin  
keycloak.securityConstraints[0].authRoles[1] = user  
keycloak.securityConstraints[0].securityCollections[0].name  
= insecure stuff  
keycloak.securityConstraints[0].securityCollections[0].pat-  
terns[0] = /insecure  
  
keycloak.securityConstraints[1].authRoles[0] = admin  
keycloak.securityConstraints[1].securityCollections[0].name  
= admin stuff  
keycloak.securityConstraints[1].securityCollections[0].pat-  
terns[0] = /admin
```

If you plan to deploy your Spring Application as a WAR then you should not use the Spring Boot Adapter and use the dedicated adapter for the application server or servlet container you are using. Your Spring Boot should also contain a `web.xml` file.

### 2.1.6. Java Servlet Filter Adapter

If you are deploying your Java Servlet application on a platform where there is no KeyCloak adapter you opt to use the servlet filter adapter. This adapter works a bit differently than the other adapters. You do not define security constraints in `web.xml`. Instead you define a filter map-

ping using the KeyCloak servlet filter adapter to secure the url patterns you want to secure.

Backchannel logout works a bit differently than the standard adapters. Instead of invalidating the HTTP session it marks the session id as logged out. There's no standard way to invalidate an HTTP session based on a session id.

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">

    <module-name>application</module-name>

    <filter>
        <filter-name>Keycloak Filter</filter-name>
        <filter-class>org.keycloak.adapters.servlet.
KeycloakOIDCFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>Keycloak Filter</filter-name>
        <url-pattern>/keycloak/*</url-pattern>
        <url-pattern>/protected/*</url-pattern>
    </filter-mapping>
</web-app>
```

In the snippet above there are two url-patterns. `/protected/*` are the files we want protected, while the `/keycloak/*` url-pattern handles callbacks from the KeyCloak server.

If you need to exclude some paths beneath the configured `url-pat-`

terns you can use the Filter init-param `keycloak.config.skipPattern` to configure a regular expression that describes a path-pattern for which the keycloak filter should immediately delegate to the filter-chain. By default no skipPattern is configured.

Patterns are matched against the `requestURI` without the `context-path`. Given the context-path `/myapp` a request for `/myapp/index.html` will be matched with `/index.html` against the skip pattern.

```
<init-param>
    <param-name>keycloak.config.skipPattern</param-name>
    <param-value>^/(path1|path2|path3).*</param-value>
</init-param>
```

Note that you should configure your client in the KeyCloak Admin Console with an Admin URL that points to a secured section covered by the filter's url-pattern.

The Admin URL will make callbacks to the Admin URL to do things like backchannel logout. So, the Admin URL in this example should be `http[s]://hostname/{context-root}/keycloak`.

The KeyCloak filter has the same configuration parameters as the other adapters except you must define them as filter init params instead of context params.

To use this filter, include this maven artifact in your WAR poms:

```
<dependency>
    <groupId>org.keycloak</groupId>
    <artifactId>keycloak-servlet-filter-adapter</artifactId>
```

```
tId>
<version>{project_versionMvn}</version>
</dependency>
```

## 2.1.7. Security Context

The `KeycloakSecurityContext` interface is available if you need to access to the tokens directly. This could be useful if you want to retrieve additional details from the token (such as user profile information) or you want to invoke a RESTful service that is protected by Keycloak.

In servlet environments it is available in secured invocations as an attribute in `HttpServletRequest`:

```
httpServletRequest
    .getAttribute(KeycloakSecurityContext.class.getName());
```

Or, it is available in insecured requests in the `HttpSession`:

```
httpServletRequest.getSession()
    .getAttribute(KeycloakSecurityContext.class.getName());
```

## 2.1.8. Error Handling

Keycloak has some error handling facilities for servlet based client adapters. When an error is encountered in authentication, Keycloak will call `HttpServletResponse.sendError()`. You can set up an error-page within your `web.xml` file to handle the error however you want. Keycloak can throw 400, 401, 403, and 500 errors.

```
<error-page>
    <error-code>403</error-code>
    <location>/ErrorHandler</location>
```

```
</error-page>
```

Keycloak also sets a `HttpServletRequest` attribute that you can retrieve. The attribute name is `org.keycloak.adapters.spi.AuthenticationError`, which should be casted to `org.keycloak.adapters.OIDCAuthenticationError`.

For example:

```
import org.keycloak.adapters.OIDCAuthenticationError;
import org.keycloak.adapters.OIDCAuthenticationError.
Reason;
...
OIDCAuthenticationError error = (OIDCAuthenticationError)
httpServletRequest
    .getAttribute('org.keycloak.adapters.spi.Authentication-
nError');

Reason reason = error.getReason();
System.out.println(reason.name());
```

## 2.1.9. Logout

You can log out of a web application in multiple ways. For Java EE servlet containers, you can call `HttpServletRequest.logout()`. For other browser applications, you can redirect the browser to `http://auth-server/auth/realms/{realm-name}/proto-col/openid-connect/logout?redirect_uri=encodedRedirec-tUri`, which logs you out if you have an SSO session with your browser.

When using the `HttpServletRequest.logout()` option the adapter executes a back-channel POST call against the Keycloak server passing

the refresh token. If the method is executed from an unprotected page (a page that does not check for a valid token) the refresh token can be unavailable and, in that case, the adapter skips the call. For this reason, using a protected page to execute `HttpServletRequest.logout()` is recommended so that current tokens are always taken into account and an interaction with the Keycloak server is performed if needed.

If you want to avoid logging out of an external identity provider as part of the logout process, you can supply the parameter `initiating_idp`, with the value being the identity (alias) of the identity provider in question. This is useful when the logout endpoint is invoked as part of single logout initiated by the external identity provider.

### 2.1.10. Parameters Forwarding

The Keycloak initial authorization endpoint request has support for various parameters. Most of the parameters are described in [OIDC specification](#). Some parameters are added automatically by the adapter based on the adapter configuration. However, there are also a few parameters that can be added on a per-invocation basis. When you open the secured application URI, the particular parameter will be forwarded to the Keycloak authorization endpoint.

For example, if you request an offline token, then you can open the secured application URI with the `scope` parameter like:

```
http://myappserver/mysecuredapp?scope=offline_access
```

and the parameter `scope=offline_access` will be automatically forwarded to the Keycloak authorization endpoint.

The supported parameters are:

- scope - Use a space-delimited list of scopes. A space-delimited list typically references [Client scopes](#) defined on particular client. Note that the scope `openid` will be always be added to the list of scopes by the adapter. For example, if you enter the scope options `address phone`, then the request to KeyCloak will contain the scope parameter `scope=openid address phone`.
- prompt - KeyCloak supports these settings:
  - `login` - SSO will be ignored and the KeyCloak login page will be always shown, even if the user is already authenticated
  - `consent` - Applicable only for the clients with `Consent Required`. If it is used, the Consent page will always be displayed, even if the user previously granted consent to this client.
  - `none` - The login page will never be shown; instead the user will be redirected to the application, with an error if the user is not yet authenticated. This setting allows you to create a filter/interceptor on the application side and show a custom error page to the user. See more details in the specification.
- `max_age` - Used only if a user is already authenticated. Specifies maximum permitted time for the authentication to persist, measured from when the user authenticated. If user is authenticated longer than `maxAge`, the SSO is ignored and he must re-authenticate.
- `login_hint` - Used to pre-fill the username/email field on the login form.
- `kc_idp_hint` - Used to tell KeyCloak to skip showing login page and

automatically redirect to specified identity provider instead. More info in the [Identity Provider documentation](#).

Most of the parameters are described in the [OIDC specification](#). The only exception is parameter `kc_idp_hint`, which is specific to KeyCloak and contains the name of the identity provider to automatically use. For more information see the `Identity Brokering` section in [{adminguide!}](#).

If you open the URL using the attached parameters, the adapter will not redirect you to KeyCloak if you are already authenticated in the application. For example, opening `http://myappserver/mysecuredapp?prompt=login` will not automatically redirect you to the KeyCloak login page if you are already authenticated to the application `mysecuredapp`. This behavior may be changed in the future.

### 2.1.11. Client Authentication

When a confidential OIDC client needs to send a backchannel request (for example, to exchange code for the token, or to refresh the token) it needs to authenticate against the KeyCloak server. By default, there are three ways to authenticate the client: client ID and client secret, client authentication with signed JWT, or client authentication with signed JWT using client secret.

#### Client ID and Client Secret

This is the traditional method described in the OAuth2 specification. The client has a secret, which needs to be known to both the adapter

(application) and the KeyCloak server. You can generate the secret for a particular client in the KeyCloak administration console, and then paste this secret into the `keycloak.json` file on the application side:

```
"credentials": {  
    "secret": "19666a4f-32dd-4049-b082-684c74115f28"  
}
```

## Client Authentication with Signed JWT

This is based on the [RFC7523](#) specification. It works this way:

- The client must have the private key and certificate. For KeyCloak this is available through the traditional `keystore` file, which is either available on the client application's classpath or somewhere on the file system.
- Once the client application is started, it allows to download its public key in [JWKS](#) format using a URL such as `http://myhost.com/myapp/k_jwks`, assuming that `http://myhost.com/myapp` is the base URL of your client application. This URL can be used by KeyCloak (see below).
- During authentication, the client generates a JWT token and signs it with its private key and sends it to KeyCloak in the particular back-channel request (for example, code-to-token request) in the `client_assertion` parameter.
- KeyCloak must have the public key or certificate of the client so that it can verify the signature on JWT. In KeyCloak you need to configure client credentials for your client. First you need to choose `Signed JWT` as the method of authenticating your client in the tab

`Credentials` in administration console. Then you can choose to either:

- Configure the JWKS URL where KeyCloak can download the client’s public keys. This can be a URL such as `http://myhost.com/myapp/k_jwks` (see details above). This option is the most flexible, since the client can rotate its keys anytime and KeyCloak then always downloads new keys when needed without needing to change the configuration. More accurately, KeyCloak downloads new keys when it sees the token signed by an unknown `kid` (Key ID).
- Upload the client’s public key or certificate, either in PEM format, in JWK format, or from the keystore. With this option, the public key is hardcoded and must be changed when the client generates a new key pair. You can even generate your own keystore from the KeyCloak administration console if you don’t have your own available. For more details on how to set up the KeyCloak administration console see `{adminguide_link}`[`{adminguide!}`].

For set up on the adapter side you need to have something like this in your `keycloak.json` file:

```
"credentials": {  
    "jwt": {  
        "client-keystore-file": "classpath:keystore-client.jks",  
        "client-keystore-type": "JKS",  
        "client-keystore-password": "storepass",  
        "client-key-password": "keypass",  
        "client-key-alias": "clientkey",  
        "token-expiration": 10  
    }  
}
```

```
    }  
}
```

With this configuration, the keystore file `keystore-client.jks` must be available on classpath in your WAR. If you do not use the prefix `classpath:` you can point to any file on the file system where the client application is running.

### 2.1.12. Multi Tenancy

Multi Tenancy, in our context, means that a single target application (WAR) can be secured with multiple Keycloak realms. The realms can be located one the same Keycloak instance or on different instances.

In practice, this means that the application needs to have multiple `keycloak.json` adapter configuration files.

You could have multiple instances of your WAR with different adapter configuration files deployed to different context-paths. However, this may be inconvenient and you may also want to select the realm based on something else than context-path.

Keycloak makes it possible to have a custom config resolver so you can choose what adapter config is used for each request.

To achieve this first you need to create an implementation of `org.keycloak.adapters.KeycloakConfigResolver`. For example:

```
package example;  
  
import org.keycloak.adapters.KeycloakConfigResolver;  
import org.keycloak.adapters.KeycloakDeployment;  
import org.keycloak.adapters.KeycloakDeploymentBuilder;
```

```

public class PathBasedKeycloakConfigResolver implements
KeycloakConfigResolver {

    @Override
    public KeycloakDeployment resolve(OIDCHttpFacade.Re-
quest request) {
        if (path.startsWith("alternative")) {
            KeycloakDeployment deployment = cache.
get(realm);
            if (null == deployment) {
                InputStream is = getClass().getResourceAs-
Stream("/tenant1-keycloak.json");
                return KeycloakDeploymentBuilder.build(is);
            }
        } else {
            InputStream is = getClass().getResourceAs-
Stream("/default-keycloak.json");
            return KeycloakDeploymentBuilder.build(is);
        }
    }

}

```

You also need to configure which `KeycloakConfigResolver` implementation to use with the `keycloak.config.resolver` context-param in your `web.xml`:

```

<web-app>
    ...
    <context-param>
        <param-name>keycloak.config.resolver</param-name>
        <param-value>example.PathBasedKeycloakConfigResol-
ver</param-value>
    </context-param>
</web-app>

```

## 2.1.13. Application Clustering

There are a few options available depending on whether your application is:

- Stateless or stateful
- Distributable (replicated http session) or non-distributable
- Relying on sticky sessions provided by load balancer
- Hosted on same domain as KeyCloak

Dealing with clustering is not quite as simple as for a regular application. Mainly due to the fact that both the browser and the server-side application sends requests to KeyCloak, so it's not as simple as enabling sticky sessions on your load balancer.

### Stateless token store

By default, the web application secured by KeyCloak uses the HTTP session to store security context. This means that you either have to enable sticky sessions or replicate the HTTP session.

As an alternative to storing the security context in the HTTP session the adapter can be configured to store this in a cookie instead. This is useful if you want to make your application stateless or if you don't want to store the security context in the HTTP session.

To use the cookie store for saving the security context, edit your applications `WEB-INF/keycloak.json` and add:

```
"token-store": "cookie"
```

The default value for `token-store` is `session`, which stores the security context in the HTTP session.

One limitation of using the cookie store is that the whole security context is passed in the cookie for every HTTP request. This may impact performance.

Another small limitation is limited support for Single-Sign Out. It works without issues if you init servlet logout (`HttpServletRequest.logout`) from the application itself as the adapter will delete the `KEYCLOAK_ADAPTER_STATE` cookie. However, back-channel logout initialized from a different application isn't propagated by Keycloak to applications using cookie store. Hence it's recommended to use a short value for the access token timeout (for example 1 minute).

Some load balancers do not allow any configuration of the sticky session cookie name or contents, such as Amazon ALB. For these, it is recommended to set the `shouldAttachRoute` option to `false`.

## Relative URI optimization

In deployment scenarios where Keycloak and the application is hosted on the same domain (through a reverse proxy or load balancer) it can be convenient to use relative URI options in your client configuration.

With relative URIs the URI is resolved as relative to the URL used to access Keycloak.

For example if the URL to your application is `https://acme.org/`

`myapp` and the URL to KeyCloak is `https://acme.org/auth`, then you can use the redirect-uri `/myapp` instead of `https://acme.org/myapp`.

## Admin URL configuration

Admin URL for a particular client can be configured in the KeyCloak Administration Console. It's used by the KeyCloak server to send backend requests to the application for various tasks, like logout users or push revocation policies.

For example the way backchannel logout works is:

1. User sends logout request from one application
2. The application sends logout request to KeyCloak
3. The KeyCloak server invalidates the user session
4. The KeyCloak server then sends a backchannel request to application with an admin url that are associated with the session
5. When an application receives the logout request it invalidates the corresponding HTTP session

If admin URL contains  `${application.session.host}` it will be replaced with the URL to the node associated with the HTTP session.

## Registration of application nodes

The previous section describes how KeyCloak can send logout request to node associated with a specific HTTP session. However, in some cases admin may want to propagate admin tasks to all registered cluster nodes, not just one of them. For example to push a new not before poli-

cy to the application or to logout all users from the application.

In this case KeyCloak needs to be aware of all application cluster nodes, so it can send the event to all of them. To achieve this, we support auto-discovery mechanism:

1. When a new application node joins the cluster, it sends a registration request to the KeyCloak server
2. The request may be re-sent to KeyCloak in configured periodic intervals
3. If the KeyCloak server doesn't receive a re-registration request within a specified timeout then it automatically unregisters the specific node
4. The node is also unregistered in KeyCloak when it sends an unregistration request, which is usually during node shutdown or application undeployment. This may not work properly for forced shutdown when undeployment listeners are not invoked, which results in the need for automatic unregistration

Sending startup registrations and periodic re-registration is disabled by default as it's only required for some clustered applications.

To enable the feature edit the `WEB-INF/keycloak.json` file for your application and add:

```
"register-node-at-startup": true,  
"register-node-period": 600,
```

This means the adapter will send the registration request on startup and re-register every 10 minutes.

In the KeyCloak Administration Console you can specify the maximum node re-registration timeout (should be larger than *register-node-period* from the adapter configuration). You can also manually add and remove cluster nodes in through the Adminstration Console, which is useful if you don't want to rely on the automatic registration feature or if you want to remove stale application nodes in the event your not using the automatic unregistration feature.

## Refresh token in each request

By default the application adapter will only refresh the access token when it's expired. However, you can also configure the adapter to refresh the token on every request. This may have a performance impact as your application will send more requests to the KeyCloak server.

To enable the feature edit the `WEB-INF/keycloak.json` file for your application and add:

```
"always-refresh-token": true
```

This may have a significant impact on performance. Only enable this feature if you can't rely on backchannel messages to propagate logout and not before policies. Another thing to consider is that by default access tokens has a short expiration so even if logout is not propagated the token will expire within minutes of the logout.

## 2.2. JavaScript Adapter

KeyCloak comes with a client-side JavaScript library that can be used to secure HTML5/JavaScript applications. The JavaScript adapter has built-in support for Cordova applications.

The library can be retrieved directly from the KeyCloak server at `/auth/js/keycloak.js` and is also distributed as a ZIP archive.

A best practice is to load the JavaScript adapter directly from KeyCloak Server as it will automatically be updated when you upgrade the server. If you copy the adapter to your web application instead, make sure you upgrade the adapter only after you have upgraded the server.

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

To use the JavaScript adapter you must first create a client for your application in the KeyCloak Administration Console. Make sure `public` is selected for `Access Type`.

You also need to configure valid redirect URIs and valid web origins. Be as specific as possible as failing to do so may result in a security vulnerability.

Once the client is created click on the `Installation` tab select `Keycloak OIDC JSON` for `Format Option` then click `Download`. The

downloaded `keycloak.json` file should be hosted on your web server at the same location as your HTML pages.

Alternatively, you can skip the configuration file and manually configure the adapter.

The following example shows how to initialize the JavaScript adapter:

```
<head>
  <script src="keycloak.js"></script>
  <script>
    var keycloak = Keycloak();
    keycloak.init().success(function(authenticated) {
      alert(authenticated ? 'authenticated' : 'not
authenticated');
    }).error(function() {
      alert('failed to initialize');
    });
  </script>
</head>
```

If the `keycloak.json` file is in a different location you can specify it:

```
var keycloak = Keycloak('http://localhost:8080/myapp/key-
cloak.json');
```

Alternatively, you can pass in a JavaScript object with the required configuration instead:

```
var keycloak = Keycloak({
  url: 'http://keycloak-server/auth',
  realm: 'myrealm',
  clientId: 'myapp'
});
```

By default to authenticate you need to call the `login` function. However, there are two options available to make the adapter automatically authenticate. You can pass `login-required` or `check-sso` to the `init` function. `login-required` will authenticate the client if the user is logged-in to Keycloak or display the login page if not. `check-sso` will only authenticate the client if the user is already logged-in, if the user is not logged-in the browser will be redirected back to the application and remain unauthenticated.

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

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

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

```
var loadData = function () {
    document.getElementById('username').innerText = keycloak.subject;

    var url = 'http://localhost:8080/restful-service';

    var req = new XMLHttpRequest();
    req.open('GET', url, true);
    req.setRequestHeader('Accept', 'application/json');
    req.setRequestHeader('Authorization', 'Bearer ' + keycloak.token);

    req.onreadystatechange = function () {
        if (req.readyState == 4) {
            if (req.status == 200) {
                alert('Success');
            }
        }
    }
}
```

```
        } else if (req.status == 403) {
            alert('Forbidden');
        }
    }

    req.send();
};
```

One thing to keep in mind is that the access token by default has a short life expiration so you may need to refresh the access token prior to sending the request. You can do this by the `updateToken` method. The `updateToken` method returns a promise object which makes it easy to invoke the service only if the token was successfully refreshed and for example display an error to the user if it wasn't. For example:

```
keycloak.updateToken(30).success(function() {
    loadData();
}).error(function() {
    alert('Failed to refresh token');
});
```

### 2.2.1. Session Status iframe

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

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

## 2.2.2. Implicit and Hybrid Flow

By default, the JavaScript adapter uses the [Authorization Code](#) flow.

With this flow the KeyCloak server returns an authorization code, not an authentication token, to the application. The JavaScript adapter exchanges the `code` for an access token and a refresh token after the browser is redirected back to the application.

KeyCloak also supports the [Implicit](#) flow where an access token is sent immediately after successful authentication with KeyCloak. This may have better performance than standard flow, as there is no additional request to exchange the code for tokens, but it has implications when the access token expires.

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

To enable implicit flow, you need to enable the `Implicit Flow Enabled` flag for the client in the KeyCloak Administration Console. You also need to pass the parameter `flow` with value `implicit` to `init` method:

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

One thing to note is that only an access token is provided and there is no refresh token. This means that once the access token has expired the application has to do the redirect to the KeyCloak again to obtain a new access token.

KeyCloak also supports the [Hybrid](#) flow.

This requires the client to have both the `Standard Flow Enabled` and `Implicit Flow Enabled` flags enabled in the admin console. The KeyCloak server will then send both the code and tokens to your application. The access token can be used immediately while the code can be exchanged for access and refresh tokens. Similar to the implicit flow, the hybrid flow is good for performance because the access token is available immediately. But, the token is still sent in the URL, and the security vulnerability mentioned earlier may still apply.

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

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

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

### 2.2.3. Hybrid Apps with Cordova

Keycloak support hybrid mobile apps developed with [Apache Cordova](#). The JavaScript adapter has two modes for this: `cordova` and `cordova-native`:

The default is `cordova`, which the adapter will automatically select if no adapter type has been configured and `window.cordova` is present. When logging in, it will open an [InApp Browser](#) that lets the user interact with KeyCloak and afterwards returns to the app by redirecting to <http://localhost>. Because of this, you must whitelist this URL as

a valid redirect-uri in the client configuration section of the Administration Console.

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

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

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

The alternative mode `cordova-native` takes a different approach. It opens the login page using the system's browser. After the user has authenticated, the browser redirects back into the app using a special URL. From there, the KeyCloak adapter can finish the login by reading the code or token from the URL.

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

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

This adapter required two additional plugins:

- [cordova-plugin-browsertab](#): allows the app to open webpages in the system's browser
- [cordova-plugin-deeplinks](#): allow the browser to redirect back to your app by special URLs

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

There are different kinds of links for opening apps: custom schemes (i.e. `myapp://login` or `android-app://com.example.myapp/https/example.com/login`) and [Universal Links \(iOS\)](#) / [Deep Links \(Android\)](#). While the former are easier to setup and tend to work more reliably, the later offer extra security as they are unique and only the owner of a domain can register them. Custom-URLs are deprecated on iOS. We recommend that you use universal links, combined with a fallback site with a custom-url link on it for best reliability.

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

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

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

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

#### 2.2.4. Earlier Browsers

The JavaScript adapter depends on Base64 (window.btoa and window.atob), HTML5 History API and optionally the Promise API. If you need to support browsers that do not have these available (for example, IE9) you need to add polyfillers.

Example polyfill libraries:

- Base64 - <https://github.com/davidchambers/Base64.js>
- HTML5 History - <https://github.com/devote/HTML5-History-API>
- Promise - <https://github.com/stefanpenner/es6-promise>

#### 2.2.5. JavaScript Adapter Reference

##### Constructor

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

##### Properties

###### **authenticated**

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

###### **token**

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

## **tokenParsed**

The parsed token as a JavaScript object.

## **subject**

The user id.

## **idToken**

The base64 encoded ID token.

## **idTokenParsed**

The parsed id token as a JavaScript object.

## **realmAccess**

The realm roles associated with the token.

## **resourceAccess**

The resource roles associated with the token.

## **refreshToken**

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

## **refreshTokenParsed**

The parsed refresh token as a JavaScript object.

## **timeSkew**

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

## **responseMode**

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

## **flow**

Flow passed in init.

## **adapter**

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

- "default" - the library uses the browser api for redirects (this is the default)
- "cordova" - the library will try to use the InAppBrowser cordova plugin to load keycloak login/registration pages (this is used automatically when the library is working in a cordova ecosystem)
- "cordova-native" - the library tries to open the login and registration page using the phone's system browser using the BrowserTabs cordova plugin. This requires extra setup for redirecting back to the app (see [Hybrid Apps with Cordova](#)).
- custom - allows you to implement a custom adapter (only for advanced use cases)

## **responseType**

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

## **pkceMethod**

The method for Proof Key Code Exchange ([PKCE](#)) to use. Configuring this value enables the PKCE mechanism. Available options:

- "S256" - The SHA256 based PKCE method

## Methods

### init(options)

Called to initialize the adapter.

Options is an Object, where:

- onLoad - Specifies an action to do on load. Supported values are 'login-required' or 'check-sso'.
- token - Set an initial value for the token.
- refreshToken - Set an initial value for the refresh token.
- idToken - Set an initial value for the id token (only together with token or refreshToken).
- timeSkew - Set an initial value for skew between local time and Keycloak server in seconds (only together with token or refreshToken).
- checkLoginIframe - Set to enable/disable monitoring login state (default is true).
- checkLoginIframeInterval - Set the interval to check login state (default is 5 seconds).
- responseMode - Set the OpenID Connect response mode send to Keycloak server at login request. Valid values are query or frag-

ment. Default value is fragment, which means that after successful authentication will Keycloak redirect to JavaScript application with OpenID Connect parameters added in URL fragment. This is generally safer and recommended over query.

- `flow` - Set the OpenID Connect flow. Valid values are standard, implicit or hybrid.
- `promiseType` - If set to `native` all methods returning a promise will return a native JavaScript promise. If not set will return Keycloak specific promise objects.

Returns promise to set functions to be invoked on success or error.

## login(options)

Redirects to login form on (options is an optional object with `redirectUri` and/or `prompt` fields).

Options is an Object, where:

- `redirectUri` - Specifies the uri to redirect to after login.
- `prompt` - This parameter allows to slightly customize the login flow on the Keycloak server side. For example enforce displaying the login screen in case of value `login`. See [Parameters Forwarding Section](#) for the details and all the possible values of the `prompt` parameter.
- `maxAge` - Used just if user is already authenticated. Specifies maximum time since the authentication of user happened. If user is already authenticated for longer time than `maxAge`, the SSO is ignored and he will need to re-authenticate again.

- loginHint - Used to pre-fill the username/email field on the login form.
- scope - Used to forward the scope parameter to the Keycloak login endpoint. Use a space-delimited list of scopes. Those typically reference [Client scopes](#) defined on particular client. Note that the scope `openid` will be always be added to the list of scopes by the adapter. For example, if you enter the scope options `address phone`, then the request to Keycloak will contain the scope parameter `scope=openid address phone`.
- idpHint - Used to tell Keycloak to skip showing the login page and automatically redirect to the specified identity provider instead. More info in the [Identity Provider documentation](#).
- action - If value is 'register' then user is redirected to registration page, otherwise to login page.
- locale - Sets the 'ui\_locales' query param in compliance with [section 3.1.2.1 of the OIDC 1.0 specification](#).
- kcLocale - Specifies the desired Keycloak locale for the UI. This differs from the locale param in that it tells the Keycloak server to set a cookie and update the user's profile to a new preferred locale.
- cordovaOptions - Specifies the arguments that are passed to the Cordova in-app-browser (if applicable). Options `hidden` and `location` are not affected by these arguments. All available options are defined at <https://cordova.apache.org/docs/en/latest/reference/cordova-plugin-inappbrowser/>. Example of use: `{ zoom: "no", hardwareback: "yes" };`

## `createLoginUrl(options)`

Returns the URL to login form on (options is an optional object with redirectUri and/or prompt fields).

Options is an Object, which supports same options like the function `login`.

### **logout(options)**

Redirects to logout.

Options is an Object, where:

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

### **createLogoutUrl(options)**

Returns the URL to logout the user.

Options is an Object, where:

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

### **register(options)**

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

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

### **createRegisterUrl(options)**

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

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

'register'

### accountManagement()

Redirects to the Account Management Console.

### createAccountUrl()

Returns the URL to the Account Management Console.

### hasRealmRole(role)

Returns true if the token has the given realm role.

### hasResourceRole(role, resource)

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

### loadUserProfile()

Loads the users profile.

Returns promise to set functions to be invoked if the profile was loaded successfully, or if the profile could not be loaded.

For example:

```
keycloak.loadUserProfile().success(function(profile) {  
    alert(JSON.stringify(profile, null, " "));  
}).error(function() {  
    alert('Failed to load user profile');  
});
```

### isTokenExpired(minValidity)

Returns true if the token has less than minValidity seconds left before it

expires (minValidity is optional, if not specified 0 is used).

### updateToken(minValidity)

If the token expires within minValidity seconds (minValidity is optional, if not specified 5 is used) the token is refreshed. If the session status iframe is enabled, the session status is also checked.

Returns promise to set functions that can be invoked if the token is still valid, or if the token is no longer valid. For example:

```
keycloak.updateToken(5).success(function(refreshed) {
    if (refreshed) {
        alert('Token was successfully refreshed');
    } else {
        alert('Token is still valid');
    }
}).error(function() {
    alert('Failed to refresh the token, or the session
has expired');
});
```

### clearToken()

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

Invoking this results in onAuthLogout callback listener being invoked.

## Callback Events

The adapter supports setting callback listeners for certain events.

For example:

```
keycloak.onAuthSuccess = function() { alert('authenticated'); }
```

The available events are:

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

## 2.3. Node.js Adapter

KeyCloak provides a Node.js adapter built on top of [Connect](#) to protect server-side JavaScript apps - the goal was to be flexible enough to integrate with frameworks like [Express.js](#).

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

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

### 2.3.1. Installation

Assuming you've already installed [Node.js](#), create a folder for your application:

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

Use `npm init` command to create a `package.json` for your application. Now add the KeyCloak connect adapter in the dependencies list:

### 2.3.2. Usage

#### Instantiate a Keycloak class

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

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

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

By default, this will locate a file named `keycloak.json` alongside the main executable of your application to initialize keycloak-specific settings (public key, realm name, various URLs). The `keycloak.json` file is obtained from the KeyCloak Admin Console.

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

```
let kcConfig = {  
    clientId: 'myclient',  
    bearerOnly: true,  
    serverUrl: 'http://localhost:8080/auth',  
    realm: 'myrealm',  
    realmPublicKey: 'MIIBIjANB...'  
};  
  
let keycloak = new Keycloak({ store: memoryStore }, kcConfig);
```

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

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

## Configuring a web session store

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

```
var session = require('express-session');  
var memoryStore = new session.MemoryStore();  
  
var keycloak = new Keycloak({ store: memoryStore });
```

## Passing a custom scope value

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

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

### 2.3.3. Installing Middleware

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

```
var app = express();

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

### 2.3.4. Checking Authentication

To check that a user is authenticated before accessing a resource, simply use `keycloak.checkSso()`. It will only authenticate if the user is already logged-in. If the user is not logged-in, the browser will be redirected back to the originally-requested URL and remain unauthenticated:

```
app.get( '/check-sso', keycloak.checkSso(), checkSsoHandler );
```

### 2.3.5. Protecting Resources

#### Simple authentication

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

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

## Role-based authorization

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

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

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

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

To secure a resource with a realm role:

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

## Resource-Based Authorization

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

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

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

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

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

```
app.get('/apis/me', keycloak.enforcer('user:profile', {response_mode: 'token'}), function (req, res) {
    var token = req.kauth.grant.access_token.content;
    var permissions = token.authorization ? token.authorization.permissions : undefined;

    // show user profile
});
```

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

If `response_mode` is set to `permissions` (default mode), the server only returns the list of granted permissions, without issuing a new access token. In addition to not issuing a new token, this method exposes

the permissions granted by the server through the `request` as follows:

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

  // show user profile
});
```

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

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

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

```
keycloak.enforcer('user:profile', {resource_server-
```

```
_id: 'my-apiserver'})
```

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

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

```
app.get('/protected/resource', keycloak.en-
forcer(['resource:view', 'resource:write'], {
  claims: function(request) {
    return {
      "http.uri": ["/protected/resource"],
      "user.agent": // get user agent from request
    }
  },
  function (req, res) {
    // access granted
  }
}), function (req, res) {
  // handle request
})
```

For more details about how to configure Keycloak to protect your application resources, please take a look at the [{authorizationguide!}](#).

## Advanced authorization

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

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

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

### 2.3.6. Additional URLs

#### Explicit user-triggered logout

By default, the middleware catches calls to `/logout` to send the user through a KeyCloak-centric logout workflow. This can be changed by specifying a `logout` configuration parameter to the `middleware()` call:

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

#### KeyCloak Admin Callbacks

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

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

## 2.4. Other OpenID Connect Libraries

KeyCloak can be secured by supplied adapters that are usually easier to use and provide better integration with KeyCloak. However, if an adapter is not available for your programming language, framework, or platform you might opt to use a generic OpenID Connect Relying Party (RP) library instead. This chapter describes details specific to KeyCloak

and does not contain specific protocol details. For more information see the [OpenID Connect specifications](#) and [OAuth2 specification](#).

#### 2.4.1. Endpoints

The most important endpoint to understand is the well-known configuration endpoint. It lists endpoints and other configuration options relevant to the OpenID Connect implementation in KeyCloak. The endpoint is:

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

To obtain the full URL, add the base URL for KeyCloak and replace {realm-name} with the name of your realm. For example:

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

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

#### Authorization Endpoint

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

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

For more details see the [Authorization Endpoint](#) section in the OpenID Connect specification.

#### Token Endpoint

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

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

For more details see the [Token Endpoint](#) section in the OpenID Connect specification.

## Userinfo Endpoint

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

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

For more details see the [Userinfo Endpoint](#) section in the OpenID Connect specification.

## Logout Endpoint

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

The logout endpoint logs out the authenticated user.

The user agent can be redirected to the endpoint, in which case the active user session is logged out. Afterward the user agent is redirected back to the application.

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

## Certificate Endpoint

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

The certificate endpoint returns the public keys enabled by the realm, encoded as a JSON Web Key (JWK). Depending on the realm settings there can be one or more keys enabled for verifying tokens. For more information see the [{adminguide!}](#) and the [JSON Web Key specification](#).

## Introspection Endpoint

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

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

For more details on how to invoke on this endpoint, see [OAuth 2.0 Token Introspection specification](#).

## Dynamic Client Registration Endpoint

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

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

For more details see the [Client Registration chapter](#) and the [OpenID Connect Dynamic Client Registration specification](#).

### 2.4.2. Validating Access Tokens

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

### 2.4.3. Flows

#### Authorization Code

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

The flow is targeted towards web applications, but is also recommended

for native applications, including mobile applications, where it is possible to embed a user agent.

For more details refer to the [Authorization Code Flow](#) in the OpenID Connect specification.

## Implicit

The Implicit flow works similarly to the Authorization Code flow, but instead of returning an Authorization Code the Access Token and ID Token is returned. This reduces the need for the extra invocation to exchange the Authorization Code for an Access Token. However, it does not include a Refresh Token. This results in the need to either permit Access Tokens with a long expiration, which is problematic as it's very hard to invalidate these. Or requires a new redirect to obtain new Access Token once the initial Access Token has expired. The Implicit flow is useful if the application only wants to authenticate the user and deals with logout itself.

There's also a Hybrid flow where both the Access Token and an Authorization Code is returned.

One thing to note is that both the Implicit flow and Hybrid flow has potential security risks as the Access Token may be leaked through web server logs and browser history. This is somewhat mitigated by using short expiration for Access Tokens.

For more details refer to the [Implicit Flow](#) in the OpenID Connect specification.

## Resource Owner Password Credentials

Resource Owner Password Credentials, referred to as Direct Grant in Keycloak, allows exchanging user credentials for tokens. It's not recommended to use this flow unless you absolutely need to. Examples where this could be useful are legacy applications and command-line interfaces.

There are a number of limitations of using this flow, including:

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

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

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

For more details refer to the [Resource Owner Password Credentials Grant](#) chapter in the OAuth 2.0 specification.

### Example using CURL

The following example shows how to obtain an access token for a user

in the realm `master` with username `user` and password `password`.

The example is using the confidential client `myclient`:

```
curl \  
  -d "client_id=myclient" \  
  -d "client_secret=40cc097b-2a57-4c17-b36a-8fdf3fc2d578" \  
  -d "username=user" \  
  -d "password=password" \  
  -d "grant_type=password" \  
  "http://localhost:8080/auth/realms/master/proto-  
  col/openid-connect/token"
```

## Client Credentials

Client Credentials is used when clients (applications and services) wants to obtain access on behalf of themselves rather than on behalf of a user. This can for example be useful for background services that applies changes to the system in general rather than for a specific user.

KeyCloak provides support for clients to authenticate either with a secret or with public/private keys.

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

For more details refer to the [Client Credentials Grant](#) chapter in the OAuth 2.0 specification.

### 2.4.4. Redirect URLs

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

- Open redirects - this can allow attackers to create spoof links that look like they are coming from your domain
- Unauthorized entry - when users are already authenticated with Keycloak an attacker can use a public client where redirect uris have not been configured correctly to gain access by redirecting the user without the users knowledge

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

There's also a few special redirect URIs:

`http://localhost`

This redirect URI is useful for native applications and allows the native application to create a web server on a random port that can be used to obtain the authorization code. This redirect uri allows any port.

`urn:ietf:wg:oauth:2.0:oob`

If its not possible to start a web server in the client (or a browser is not available) it is possible to use the special

`urn:ietf:wg:oauth:2.0:oob` redirect uri. When this redirect uri is used Keycloak displays a page with the code in the title and in a box on the page. The application can either detect that the browser title has changed, or the user can copy/paste the code manually to the application. With this redirect uri it is also possible for a user to use a different device to obtain a code to paste back to the application.

## 3. SAML

This section describes how you can secure applications and services with SAML using either KeyCloak client adapters or generic SAML provider libraries.

### 3.1. Java Adapters

KeyCloak comes with a range of different adapters for Java application. Selecting the correct adapter depends on the target platform.

#### 3.1.1. General Adapter Config

Each SAML client adapter supported by KeyCloak can be configured by a simple XML text file. This is what one might look like:

```
<keycloak-saml-adapter xmlns="urn:keycloak:saml:adapter"
    xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance"
    xsi:schemaLocation="urn:keyclo-
    ak:saml:adapter {saml_adapter_xsd_urn}">
    <SP entityID="http://localhost:8081/sales-post-sig/"
        sslPolicy="EXTERNAL"
        nameIDPolicyFormat="urn:oasis:names:tc:SAML:1.1:na-
        meid-format:unspecified"
        logoutPage="/logout.jsp"
        forceAuthentication="false"
        isPassive="false"
        turnOffChangeSessionIdOnLogin="false"
        autodetectBearerOnly="false">
        <Keys>
            <Key signing="true" >
                <KeyStore resource="/WEB-INF/keystore.jks"
password="store123">
                    <PrivateKey alias="http://local-
host:8080/sales-post-sig/" password="test123"/>
```

```

        <Certificate alias="http://local-
host:8080/sales-post-sig"/>
    </KeyStore>
</Key>
</Keys>
<PrincipalNameMapping policy="FROM_NAME_ID"/>
<RoleIdentifiers>
    <Attribute name="Role"/>
</RoleIdentifiers>
<IDP entityID="idp"
    signaturesRequired="true">
    <SingleSignOnService requestBinding="POST"
        bindingUrl="http://local-
host:8081/auth/realm/demo/protocol/saml"
    />

    <SingleLogoutService
        requestBinding="POST"
        responseBinding="POST"
        postBindingUrl="http://local-
host:8081/auth/realm/demo/protocol/saml"
        redirectBindingUrl="http://local-
host:8081/auth/realm/demo/protocol/saml"
    />
<Keys>
    <Key signing="true">
        <KeyStore resource="/WEB-INF/keysto-
re.jks" password="store123">
            <Certificate alias="demo"/>
        </KeyStore>
    </Key>
</Keys>
</IDP>
</SP>
</keycloak-saml-adapter>

```

Some of these configuration switches may be adapter specific and some are common across all adapters. For Java adapters you can use  `${...}` enclosure as System property replacement. For example  `${jboss.server.config.dir}.`

## SP Element

Here is the explanation of the SP element attributes:

```
<SP entityId="sp"
      sslPolicy="ssl"
      nameIDPolicyFormat="format"
      forceAuthentication="true"
      isPassive="false"
      autodetectBearerOnly="false">
  ...
</SP>
```

### entityID

This is the identifier for this client. The IdP needs this value to determine who the client is that is communicating with it. This setting is *REQUIRED*.

### sslPolicy

This is the SSL policy the adapter will enforce. Valid values are: `ALL`, `EXTERNAL`, and `NONE`. For `ALL`, all requests must come in via HTTPS. For `EXTERNAL`, only non-private IP addresses must come over the wire via HTTPS. For `NONE`, no requests are required to come over via HTTPS. This setting is *OPTIONAL*. Default value is `EXTERNAL`.

### nameIDPolicyFormat

SAML clients can request a specific NameID Subject format. Fill in this value if you want a specific format. It must be a standard SAML format identifier: `urn:oasis:names:tc:SAML:2.0:nameid-format:transient`. This setting is *OPTIONAL*. By default, no special format is requested.

## **forceAuthentication**

SAML clients can request that a user is re-authenticated even if they are already logged in at the IdP. Set this to `true` to enable. This setting is *OPTIONAL*. Default value is `false`.

## **isPassive**

SAML clients can request that a user is never asked to authenticate even if they are not logged in at the IdP. Set this to `true` if you want this. Do not use together with `forceAuthentication` as they are opposite. This setting is *OPTIONAL*. Default value is `false`.

## **turnOffChangeSessionIdOnLogin**

The session ID is changed by default on a successful login on some platforms to plug a security attack vector. Change this to `true` to disable this. It is recommended you do not turn it off. Default value is `false`.

## **autodetectBearerOnly**

This should be set to *true* if your application serves both a web application and web services (e.g. SOAP or REST). It allows you to redirect unauthenticated users of the web application to the Keycloak login page, but send an HTTP `401` status code to unauthenticated SOAP or REST clients instead as they would not understand a redirect to the login page. Keycloak auto-detects SOAP or REST clients based on typical headers like `X-Requested-With`, `SOAPAction` or `Accept`. The default value is *false*.

## **logoutPage**

This sets the page to display after logout. If the page is a full URL, such as `http://web.example.com/logout.html`, the user is redirected after logout to that page using the HTTP 302 status code. If a link without scheme part is specified, such as `/logout.jsp`, the page is displayed after logout, *regardless of whether it lies in a protected area according to security-constraint declarations in web.xml*, and the page is resolved relative to the deployment context root.

## Service Provider Keys and Key Elements

If the IdP requires that the client application (or SP) sign all of its requests and/or if the IdP will encrypt assertions, you must define the keys used to do this. For client-signed documents you must define both the private and public key or certificate that is used to sign documents. For encryption, you only have to define the private key that is used to decrypt it.

There are two ways to describe your keys. They can be stored within a Java KeyStore or you can copy/paste the keys directly within `keycloak-saml.xml` in the PEM format.

```
<Keys>
  <Key signing="true" >
    ...
  </Key>
</Keys>
```

The `Key` element has two optional attributes `signing` and `encryption`. When set to true these tell the adapter what the key will be used for. If both attributes are set to true, then the key will be used for both

signing documents and decrypting encrypted assertions. You must set at least one of these attributes to true.

## KeyStore element

Within the `Key` element you can load your keys and certificates from a Java Keystore. This is declared within a `KeyStore` element.

```
<Keys>
    <Key signing="true" >
        <KeyStore resource="/WEB-INF/keystore.jks"
password="store123">
            <PrivateKey alias="myPrivate" pass-
word="test123"/>
            <Certificate alias="myCertAlias"/>
        </KeyStore>
    </Key>
</Keys>
```

Here are the XML config attributes that are defined with the `KeyStore` element.

### file

File path to the key store. This option is *OPTIONAL*. The file or resource attribute must be set.

### resource

WAR resource path to the KeyStore. This is a path used in method call to `ServletContext.getResourceAsStream()`. This option is *OPTIONAL*. The file or resource attribute must be set.

### password

The password of the KeyStore. This option is *REQUIRED*.

If you are defining keys that the SP will use to sign document, you must also specify references to your private keys and certificates within the Java KeyStore. The `PrivateKey` and `Certificate` elements in the above example define an `alias` that points to the key or cert within the keystore. Keystores require an additional password to access private keys. In the `PrivateKey` element you must define this password within a `password` attribute.

## Key PEMs

Within the `Key` element you declare your keys and certificates directly using the sub elements `PrivateKeyPem`, `PublicKeyPem`, and `CertificatePem`. The values contained in these elements must conform to the PEM key format. You usually use this option if you are generating keys using `openssl` or similar command line tool.

```
<Keys>
  <Key signing="true">
    <PrivateKeyPem>
      2341251234AB31234==231BB998311222423522334
    </PrivateKeyPem>
    <CertificatePem>
      211111341251234AB31234==231BB998311222423522334
    </CertificatePem>
  </Key>
</Keys>
```

## SP PrincipalNameMapping element

This element is optional. When creating a Java Principal object that you obtain from methods such as `HttpServletRequest.getUserPrincipal()`, you can define what name is returned by the `Principal.getName()` method.

```
<SP ...>
  <PrincipalNameMapping policy="FROM_NAME_ID"/>
</SP>

<SP ...>
  <PrincipalNameMapping policy="FROM_ATTRIBUTE" attribu-
  te="email" />
</SP>
```

The `policy` attribute defines the policy used to populate this value. The possible values for this attribute are:

### **FROM\_NAME\_ID**

This policy just uses whatever the SAML subject value is. This is the default setting

### **FROM\_ATTRIBUTE**

This will pull the value from one of the attributes declared in the SAML assertion received from the server. You'll need to specify the name of the SAML assertion attribute to use within the `attribute` XML attribute.

## **RoleIdentifiers Element**

The `RoleIdentifiers` element defines what SAML attributes within the assertion received from the user should be used as role identifiers within the Java EE Security Context for the user.

```
<RoleIdentifiers>
  <Attribute name="Role"/>
  <Attribute name="member"/>
  <Attribute name="memberOf"/>
</RoleIdentifiers>
```

By default `Role` attribute values are converted to Java EE roles. Some IdPs send roles using a `member` or `memberOf` attribute assertion. You can define one or more `Attribute` elements to specify which SAML attributes must be converted into roles.

## IDP Element

Everything in the IDP element describes the settings for the identity provider (authentication server) the SP is communicating with.

```
<IDP entityID="idp"
      signaturesRequired="true"
      signatureAlgorithm="RSA_SHA1"
      signatureCanonicalizationMethod="http://www.w3.org/
2001/10/xml-exc-c14n#">
  ...
</IDP>
```

Here are the attribute config options you can specify within the `IDP` element declaration.

### **entityID**

This is the issuer ID of the IDP. This setting is *REQUIRED*.

### **signaturesRequired**

If set to `true`, the client adapter will sign every document it sends to the IDP. Also, the client will expect that the IDP will be signing any documents sent to it. This switch sets the default for all request and response types, but you will see later that you have some fine grain control over this. This setting is *OPTIONAL* and will default to `false`.

## **signatureAlgorithm**

This is the signature algorithm that the IDP expects signed documents to use. Allowed values are: `RSA_SHA1`, `RSA_SHA256`, `RSA_SHA512`, and `DSA_SHA1`. This setting is *OPTIONAL* and defaults to `RSA_SHA256`.

## **signatureCanonicalizationMethod**

This is the signature canonicalization method that the IDP expects signed documents to use. This setting is *OPTIONAL*. The default value is `http://www.w3.org/2001/10/xml-exc-c14n#` and should be good for most IDPs.

## **metadataUrl**

The URL used to retrieve the IDP metadata, currently this is only used to pick up signing and encryption keys periodically which allow cycling of these keys on the IDP without manual changes on the SP side.

## **IDP SingleSignOnService sub element**

The `SingleSignOnService` sub element defines the login SAML endpoint of the IDP. The client adapter will send requests to the IDP formatted via the settings within this element when it wants to login.

```
<SingleSignOnService signRequest="true"  
                    validateResponseSignature="true"  
                    requestBinding="post"  
                    bindingUrl="url"/>
```

Here are the config attributes you can define on this element:

## **signRequest**

Should the client sign authn requests? This setting is *OPTIONAL*. Defaults to whatever the IDP `signaturesRequired` element value is.

## **validateResponseSignature**

Should the client expect the IDP to sign the assertion response document sent back from an auhtn request? This setting *OPTIONAL*. Defaults to whatever the IDP `signaturesRequired` element value is.

## **requestBinding**

This is the SAML binding type used for communicating with the IDP. This setting is *OPTIONAL*. The default value is `POST`, but you can set it to `REDIRECT` as well.

## **responseBinding**

SAML allows the client to request what binding type it wants authn responses to use. The values of this can be `POST` or `REDIRECT`. This setting is *OPTIONAL*. The default is that the client will not request a specific binding type for responses.

## **assertionConsumerServiceUrl**

URL of the assertion consumer service (ACS) where the IDP login service should send responses to. This setting is *OPTIONAL*. By default it is unset, relying on the configuration in the IdP. When set, it must end in `/saml`, e.g. `http://sp.domain.com/my/endpoint/for/saml`. The value of this property is sent in `AssertionConsumerServiceURL` attribute of SAML `AuthnRequest` message. This

property is typically accompanied by the `responseBinding` attribute.

## bindingUrl

This is the URL for the IDP login service that the client will send requests to. This setting is *REQUIRED*.

## IDP SingleLogoutService sub element

The `SingleLogoutService` sub element defines the logout SAML endpoint of the IDP. The client adapter will send requests to the IDP formatted via the settings within this element when it wants to logout.

```
<SingleLogoutService validateRequestSignature="true"  
                     validateResponseSignature="true"  
                     signRequest="true"  
                     signResponse="true"  
                     requestBinding="redirect"  
                     responseBinding="post"  
                     postBindingUrl="posturl"  
                     redirectBindingUrl="redirecturl">
```

## signRequest

Should the client sign logout requests it makes to the IDP? This setting is *OPTIONAL*. Defaults to whatever the IDP `signaturesRequired` element value is.

## signResponse

Should the client sign logout responses it sends to the IDP requests? This setting is *OPTIONAL*. Defaults to whatever the IDP `signaturesRequired` element value is.

## **validateRequestSignature**

Should the client expect signed logout request documents from the IDP? This setting is *OPTIONAL*. Defaults to whatever the IDP `signaturesRequired` element value is.

## **validateResponseSignature**

Should the client expect signed logout response documents from the IDP? This setting is *OPTIONAL*. Defaults to whatever the IDP `signaturesRequired` element value is.

## **requestBinding**

This is the SAML binding type used for communicating SAML requests to the IDP. This setting is *OPTIONAL*. The default value is `POST`, but you can set it to `REDIRECT` as well.

## **responseBinding**

This is the SAML binding type used for communicating SAML responses to the IDP. The values of this can be `POST` or `REDIRECT`. This setting is *OPTIONAL*. The default value is `POST`, but you can set it to `REDIRECT` as well.

## **postBindingUrl**

This is the URL for the IDP's logout service when using the POST binding. This setting is *REQUIRED* if using the `POST` binding.

## **redirectBindingUrl**

This is the URL for the IDP's logout service when using the `REDIRECT` binding. This setting is *REQUIRED* if using the `REDIRECT` binding.

## IDP Keys sub element

The Keys sub element of IDP is only used to define the certificate or public key to use to verify documents signed by the IDP. It is defined in the same way as the [SP's Keys element](#). But again, you only have to define one certificate or public key reference. Note that, if both IDP and SP are realized by KeyCloak server and adapter, respectively, there is no need to specify the keys for signature validation, see below.

It is possible to configure SP to obtain public keys for IDP signature validation from published certificates automatically, provided both SP and IDP are implemented by KeyCloak. This is done by removing all declarations of signature validation keys in Keys sub element. If the Keys sub element would then remain empty, it can be omitted completely. The keys are then automatically obtained by SP from SAML descriptor, location of which is derived from SAML endpoint URL specified in the [IDP SingleSignOnService sub element](#). Settings of the HTTP client that is used for SAML descriptor retrieval usually needs no additional configuration, however it can be configured in the [IDP HttpClient sub element](#).

It is also possible to specify multiple keys for signature verification. This is done by declaring multiple Key elements within Keys sub element that have `signing` attribute set to `true`. This is useful for example in situation when the IDP signing keys are rotated: There is usually a transition period when new SAML protocol messages and assertions are signed with the new key but those signed by previous key should still be accepted.

It is not possible to configure KeyCloak to both obtain the keys for si-

gnature verification automatically and define additional static signature verification keys.

```
<IDP entityId="idp">
    ...
    <Keys>
        <Key signing="true">
            <KeyStore resource="/WEB-INF/keystore.jks" password="store123">
                <Certificate alias="demo"/>
            </KeyStore>
        </Key>
    </Keys>
</IDP>
```

## IDP HttpClient sub element

The `HttpClient` optional sub element defines the properties of HTTP client used for automatic obtaining of certificates containing public keys for IDP signature verification via SAML descriptor of the IDP when enabled.

```
<HttpClient connectionPoolSize="10"
    disableTrustManager="false"
    allowAnyHostname="false"
    clientKeystore="classpath:keystore.jks"
    clientKeystorePassword="pwd"
    truststore="classpath:truststore.jks"
    truststorePassword="pwd"
    proxyUrl="http://proxy/" />
```

## connectionPoolSize

Adapters will make separate HTTP invocations to the Keycloak server to turn an access code into an access token. This config option defines how many connections to the Keycloak server should be

pooled. This is *OPTIONAL*. The default value is `10`.

### **disableTrustManager**

If the Keycloak server requires HTTPS and this config option is set to `true` you do not have to specify a truststore. This setting should only be used during development and **never** in production as it will disable verification of SSL certificates. This is *OPTIONAL*. The default value is `false`.

### **allowAnyHostname**

If the Keycloak server requires HTTPS and this config option is set to `true` the Keycloak server's certificate is validated via the truststore, but host name validation is not done. This setting should only be used during development and **never** in production as it will partly disable verification of SSL certificates. This setting may be useful in test environments. This is *OPTIONAL*. The default value is `false`.

### **truststore**

The value is the file path to a truststore file. If you prefix the path with `classpath:`, then the truststore will be obtained from the deployment's classpath instead. Used for outgoing HTTPS communications to the Keycloak server. Client making HTTPS requests need a way to verify the host of the server they are talking to. This is what the truststore does. The keystore contains one or more trusted host certificates or certificate authorities. You can create this truststore by extracting the public certificate of the Keycloak server's SSL keystore. This is *REQUIRED* unless `disableTrustManager` is `true`.

### **truststorePassword**

Password for the truststore. This is *REQUIRED* if `truststore` is set and the truststore requires a password.

### **clientKeystore**

This is the file path to a keystore file. This keystore contains client certificate for two-way SSL when the adapter makes HTTPS requests to the KeyCloak server. This is *OPTIONAL*.

### **clientKeystorePassword**

Password for the client keystore and for the client's key. This is *REQUIRED* if `clientKeystore` is set.

### **proxyUrl**

URL to HTTP proxy to use for HTTP connections. This is *OPTIONAL*.

You then provide a keycloak config, `/WEB-INF/keycloak-saml.xml` file in your WAR and change the auth-method to KEYCLOAK-SAML within web.xml. Both methods are described in this section.

## **Adapter Installation**

Each adapter is a separate download on the KeyCloak download site.

After adding the modules, you must then enable the KeyCloak SAML Subsystem within your app server's server configuration: `domain.xml` or `standalone.xml`.

There is a CLI script that will help you modify your server configuration. Start the server and run the script from the server's bin directory:

The script will add the extension, subsystem, and optional security-domain as described below.

```
<server xmlns="urn:jboss:domain:1.4">

    <extensions>
        <extension module="org.keycloak.keycloak-saml-adapter-subsystem"/>
        ...
    </extensions>

    <profile>
        <subsystem xmlns="urn:jboss:domain:keycloak-saml:1.1"/>
        ...
    </profile>
```

The `keycloak` security domain should be used with EJBs and other components when you need the security context created in the secured web tier to be propagated to the EJBs (other EE component) you are invoking. Otherwise this configuration is optional.

```
<server xmlns="urn:jboss:domain:1.4">
    <subsystem xmlns="urn:jboss:domain:security:1.2">
        <security-domains>
            ...
            <security-domain name="keycloak">
                <authentication>
                    <login-module code="org.keycloak.adapters.jboss.KeycloakLoginModule"
                        flag="required"/>
                </authentication>
            </security-domain>
        </security-domains>
    </subsystem>
```

For example, if you have a JAX-RS service that is an EJB within your

WEB-INF/classes directory, you'll want to annotate it with the `@SecurityDomain` annotation as follows:

```
import org.jboss.ejb3.annotation.SecurityDomain;
import org.jboss.resteasy.annotations.cache.NoCache;

import javax.annotation.security.RolesAllowed;
import javax.ejb.EJB;
import javax.ejb.Stateless;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import java.util.ArrayList;
import java.util.List;

@Path("customers")
@Stateless
@SecurityDomain("keycloak")
public class CustomerService {

    @EJB
    CustomerDB db;

    @GET
    @Produces("application/json")
    @NoCache
    @RolesAllowed("db_user")
    public List<String> getCustomers() {
        return db.getCustomers();
    }
}
```

We hope to improve our integration in the future so that you don't have to specify the `@SecurityDomain` annotation when you want to propagate a keycloak security context to the EJB tier.

## JBoss SSO

WildFly has built-in support for single sign-on for web applications de-

ployed to the same WildFly instance. This should not be enabled when using KeyCloak.

### 3.1.2. Installing JBoss EAP Adapter from an RPM

Install the EAP 7 Adapters from an RPM:

With Red Hat Enterprise Linux 7, the term channel was replaced with the term repository. In these instructions only the term repository is used.

You must subscribe to the JBoss EAP 7 repository before you can install the EAP 7 adapters from an RPM.

#### *Prerequisites*

1. Ensure that your Red Hat Enterprise Linux system is registered to your account using Red Hat Subscription Manager. For more information see the [Red Hat Subscription Management documentation](#).
2. If you are already subscribed to another JBoss EAP repository, you must unsubscribe from that repository first.

For Red Hat Enterprise Linux 6, 7: Using Red Hat Subscription Manager, subscribe to the WildFly {appserver\_version} repository using the following command. Replace <RHEL\_VERSION> with either 6 or 7 depending on your Red Hat Enterprise Linux version.

```
$ sudo subscription-manager repos --enable=jb-eap-7-for-rhel-<RHEL_VERSION>-server-rpms
```

For Red Hat Enterprise Linux 8: Using Red Hat Subscription Manager,

subscribe to the WildFly {appserver\_version} repository using the following command:

```
$ sudo subscription-manager repos --enable=jb-eap-{appserver_version}-for-rhel-8-x86_64-rpms --enable=rhel-8-for-x86_64-baseos-rpms --enable=rhel-8-for-x86_64-appstream-rpms
```

Install the EAP 7 adapters for SAML using the following command:

```
$ sudo yum install eap7-keycloak-saml-adapter-sso7_3
```

or use following one for Red Hat Red Hat Enterprise Linux 8:

```
$ sudo dnf install eap7-keycloak-adapter-sso7_3
```

The default EAP\_HOME path for the RPM installation is /opt/rh/eap7/root/usr/share/wildfly.

Run the appropriate module installation script.

For the SAML module, enter the following command:

```
$ $EAP_HOME/bin/jboss-cli.sh -c --file=$EAP_HOME/bin/adapter-install-saml.cli
```

Your installation is complete.

Install the EAP 6 Adapters from an RPM:

With Red Hat Enterprise Linux 7, the term channel was replaced with the term repository. In these instructions only the term repository is used.

You must subscribe to the JBoss EAP 6 repository before you can install the EAP 6 adapters from an RPM.

### *Prerequisites*

1. Ensure that your Red Hat Enterprise Linux system is registered to your account using Red Hat Subscription Manager. For more information see the [Red Hat Subscription Management documentation](#).
2. If you are already subscribed to another JBoss EAP repository, you must unsubscribe from that repository first.

Using Red Hat Subscription Manager, subscribe to the JBoss EAP 6 repository using the following command. Replace <RHEL\_VERSION> with either 6 or 7 depending on your Red Hat Enterprise Linux version.

```
$ sudo subscription-manager repos --enable=jb-eap-6-for-rhel-<RHEL_VERSION>-server-rpms
```

Install the EAP 6 adapters for SAML using the following command:

```
$ sudo yum install keycloak-saml-adapter-sso7_3-eap6
```

The default EAP\_HOME path for the RPM installation is /opt/rh/eap6/root/usr/share/wildfly.

Run the appropriate module installation script.

For the SAML module, enter the following command:

```
$ $EAP_HOME/bin/jboss-cli.sh -c --file=$EAP_HOME/bin/adapter-install-saml.cli
```

Your installation is complete.

## Per WAR Configuration

This section describes how to secure a WAR directly by adding config and editing files within your WAR package.

The first thing you must do is create a `keycloak-saml.xml` adapter config file within the `WEB-INF` directory of your WAR. The format of this config file is described in the [General Adapter Config](#) section.

Next you must set the `auth-method` to `KEYCLOAK-SAML` in `web.xml`. You also have to use standard servlet security to specify role-base constraints on your URLs. Here's an example `web.xml` file:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0">

    <module-name>customer-portal</module-name>

    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Admins</web-resource-name>
            <url-pattern>/admin/*</url-pattern>
        </web-resource-collection>
```

```

<auth-constraint>
    <role-name>admin</role-name>
</auth-constraint>
<user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-
guarantee>
    </user-data-constraint>
</security-constraint>
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Customers</web-resource-na-
me>
        <url-pattern>/customers/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>user</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-
guarantee>
    </user-data-constraint>
</security-constraint>

<login-config>
    <auth-method>KEYCLOAK-SAML</auth-method>
    <realm-name>this is ignored currently</realm-name>
</login-config>

<security-role>
    <role-name>admin</role-name>
</security-role>
<security-role>
    <role-name>user</role-name>
</security-role>
</web-app>

```

All standard servlet settings except the `auth-method` setting.

## Securing WARs via KeyCloak SAML Subsystem

You do not have to crack open a WAR to secure it with KeyCloak. Alternatively, you can externally secure it via the KeyCloak SAML Adap-

ter Subsystem. While you don't have to specify KEYCLOAK-SAML as an `auth-method`, you still have to define the `security-constraints` in `web.xml`. You do not, however, have to create a `WEB-INF/keycloak-saml.xml` file. This metadata is instead defined within the XML in your server's `domain.xml` or `standalone.xml` subsystem configuration section.

```
<extensions>
  <extension module="org.keycloak.keycloak-saml-adapter-
subsystem"/>
</extensions>

<profile>
  <subsystem xmlns="urn:jboss:domain:keycloak-saml:1.1">
    <secure-deployment name="WAR MODULE NAME.war">
      <SP entityID="APPLICATION URL">
        ...
      </SP>
    </secure-deployment>
  </subsystem>
</profile>
```

The `secure-deployment name` attribute identifies the WAR you want to secure. Its value is the `module-name` defined in `web.xml` with `.war` appended. The rest of the configuration uses the same XML syntax as `keycloak-saml.xml` configuration defined in [General Adapter Config](#).

An example configuration:

```
<subsystem xmlns="urn:jboss:domain:keycloak-saml:1.1">
  <secure-deployment name="saml-post-encryption.war">
    <SP entityID="http://localhost:8080/sales-post-enc/">
      <sslPolicy>EXTERNAL</sslPolicy>
      <nameIDPolicyFormat>urn:oasis:names:tc:SAML:1.1:na-
```

```

meid-format:unspecified"
    logoutPage="/logout.jsp"
    forceAuthentication="false">
<Keys>
    <Key signing="true" encryption="true">
        <KeyStore resource="/WEB-INF/keystore.jks" pass-
word="store123">
            <PrivateKey alias="http://localhost:8080/sales-
post-enc/" password="test123"/>
            <Certificate alias="http://localhost:8080/sa-
les-post-enc/" />
        </KeyStore>
    </Key>
</Keys>
<PrincipalNameMapping policy="FROM_NAME_ID"/>
<RoleIdentifiers>
    <Attribute name="Role"/>
</RoleIdentifiers>
<IDP entityID="idp">
    <SingleSignOnService signRequest="true"
        validateResponseSignature="true"
        requestBinding="POST"
        bindingUrl="http://localhost:8080/auth/re-
alms/saml-demo/protocol/saml"/>

    <SingleLogoutService
        validateRequestSignature="true"
        validateResponseSignature="true"
        signRequest="true"
        signResponse="true"
        requestBinding="POST"
        responseBinding="POST"
        postBindingUrl="http://localhost:8080/auth/re-
alms/saml-demo/protocol/saml"
        redirectBindingUrl="http://local-
host:8080/auth/realms/saml-demo/protocol/saml"/>
    <Keys>
        <Key signing="true" >
            <KeyStore resource="/WEB-INF/keystore.jks"
password="store123">
                <Certificate alias="saml-demo"/>
            </KeyStore>
        </Key>
    </Keys>

```

```
</IDP>
</SP>
</secure-deployment>
</subsystem>
```

### 3.1.3. Java Servlet Filter Adapter

If you want to use SAML with a Java servlet application that doesn't have an adapter for that servlet platform, you can opt to use the servlet filter adapter that KeyCloak has. This adapter works a little differently than the other adapters. You still have to specify a `/WEB-INF/keycloak-saml.xml` file as defined in the [General Adapter Config](#) section, but you do not define security constraints in `web.xml`. Instead you define a filter mapping using the KeyCloak servlet filter adapter to secure the url patterns you want to secure.

Backchannel logout works a bit differently than the standard adapters. Instead of invalidating the http session it instead marks the session ID as logged out.

There's just no way of arbitrarily invalidating an http session based on a session ID.

Backchannel logout does not currently work when you have a clustered application that uses the SAML filter.

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0">

    <module-name>customer-portal</module-name>
```

```

<filter>
    <filter-name>Keycloak Filter</filter-name>
    <filter-class>org.keycloak.adapters.saml.servlet.
SamlFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>Keycloak Filter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
</web-app>

```

The Keycloak filter has the same configuration parameters available as the other adapters except you must define them as filter init params instead of context params.

You can define multiple filter mappings if you have various different secure and unsecure url patterns.

You must have a filter mapping that covers `/saml`.  
This mapping covers all server callbacks.

When registering SPs with an IdP, you must register

`http[s]://hostname/{context-root}/saml` as your Assert Consumer Service URL and Single Logout Service URL.

To use this filter, include this maven artifact in your WAR poms:

```

<dependency>
    <groupId>org.keycloak</groupId>
    <artifactId>keycloak-saml-servlet-filter-adapter</arti-
factId>
    <version>{project_versionMvn}</version>
</dependency>

```

In order to use [Multi Tenancy](#) the `keycloak.config.resolver` parameter should be passed as a filter parameter.

```
<filter>
    <filter-name>Keycloak Filter</filter-name>
    <filter-class>org.keycloak.adapters.saml.servlet.
SamlFilter</filter-class>
    <init-param>
        <param-name>keycloak.config.resolver</param-na-
me>
        <param-value>example.SamlMultiTenantResol-
ver</param-value>
    </init-param>
</filter>
```

### 3.1.4. Registering with an Identity Provider

For each servlet-based adapter, the endpoint you register for the assert consumer service URL and single logout service must be the base URL of your servlet application with `/saml` appended to it, that is,

`https://example.com/contextPath/saml`.

### 3.1.5. Logout

There are multiple ways you can logout from a web application. For Java EE servlet containers, you can call `HttpServletRequest.logout()`. For any other browser application, you can point the browser at any url of your web application that has a security constraint and pass in a query parameter `GLO`, i.e. `http://myapp?GLO=true`. This will log you out if you have an SSO session with your browser.

### Logout in Clustered Environment

Internally, the SAML adapter stores a mapping between the SAML session index, principal name (when known), and HTTP session ID. This

mapping can be maintained in JBoss application server family (WildFly 10/11, EAP 6/7) across cluster for distributable applications. As a pre-condition, the HTTP sessions need to be distributed across cluster (i.e. application is marked with `<distributable/>` tag in application's `web.xml`).

To enable the functionality, add the following section to your `/WEB-INF/web.xml` file:

For EAP 7, WildFly 10/11:

```
<context-param>
    <param-name>keycloak.sessionIdMapperUpdater.classes</param-name>
    <param-value>org.keycloak.adapters.saml.wildfly.infinispan.InfinispanSessionCacheIdMapperUpdater</param-value>
</context-param>
```

For EAP 6:

```
<context-param>
    <param-name>keycloak.sessionIdMapperUpdater.classes</param-name>
    <param-value>org.keycloak.adapters.saml.jbossweb.infinispan.InfinispanSessionCacheIdMapperUpdater</param-value>
</context-param>
```

If the session cache of the deployment is named `deployment-cache`, the cache used for SAML mapping will be named as `deployment-cache.ssoCache`. The name of the cache can be overridden by a context parameter `keycloak.sessionIdMapperUpdater.infinispan`.

`cacheName`. The cache container containing the cache will be the same as the one containing the deployment session cache, but can be overridden by a context parameter `keycloak.sessionIdMapperUpdater.infinispan.containerName`.

By default, the configuration of the SAML mapping cache will be derived from session cache. The configuration can be manually overridden in cache configuration section of the server just the same as other caches.

Currently, to provide reliable service, it is recommended to use replicated cache for the SAML session cache. Using distributed cache may lead to results where the SAML logout request would land to a node with no access to SAML session index to HTTP session mapping which would lead to unsuccessful logout.

## Logout in Cross DC Scenario

The cross DC scenario only applies to WildFly 10 and higher, and EAP 7 and higher.

Special handling is needed for handling sessions that span multiple data centers. Imagine the following scenario:

1. Login requests are handled within cluster in data center 1.
2. Admin issues logout request for a particular SAML session, the request lands in data center 2.

The data center 2 has to log out all sessions that are present in data center 1 (and all other data centers that share HTTP sessions).

To cover this case, the SAML session cache described [above](#) needs to be replicated not only within individual clusters but across all the data centers e.g. [via standalone Infinispan/JDG server](#):

1. A cache has to be added to the standalone Infinispan/JDG server.
2. The cache from previous item has to be added as a remote store for the respective SAML session cache.

Once remote store is found to be present on SAML session cache during deployment, it is watched for changes and the local SAML session cache is updated accordingly.

### 3.1.6. Obtaining Assertion Attributes

After a successful SAML login, your application code may want to obtain attribute values passed with the SAML assertion. `HttpServletRequest.getUserPrincipal()` returns a `Principal` object that you can typecast into a Keycloak specific class called `org.keycloak.adapters.saml.SamlPrincipal`. This object allows you to look at the raw assertion and also has convenience functions to look up attribute values.

```
package org.keycloak.adapters.saml;

public class SamlPrincipal implements Serializable, Principal {
    /**
     * Get full saml assertion
     *
     * @return
     */
    public AssertionType getAssertion() {
        ...
    }
}
```

```

    /**
     * Get SAML subject sent in assertion
     *
     * @return
     */
    public String getSam1Subject() {
        ...
    }

    /**
     * Subject nameID format
     *
     * @return
     */
    public String getNameIDFormat() {
        ...
    }

    @Override
    public String getName() {
        ...
    }

    /**
     * Convenience function that gets Attribute value by
     attribute name
     *
     * @param name
     * @return
     */
    public List<String> getAttributes(String name) {
        ...
    }

    /**
     * Convenience function that gets Attribute value by
     attribute friendly name
     *
     * @param friendlyName
     * @return
     */
    public List<String> getFriendlyAttributes(String fri-

```

```

endlyName) {
    ...
}

/**
 * Convenience function that gets first value of an
attribute by attribute name
*
* @param name
* @return
*/
public String getAttribute(String name) {
    ...
}

/**
 * Convenience function that gets first value of an
attribute by attribute name
*
*
* @param friendlyName
* @return
*/
public String getFriendlyAttribute(String friendlyName)
{
    ...
}

/**
 * Get set of all assertion attribute names
*
* @return
*/
public Set<String> getAttributeNames() {
    ...
}

/**
 * Get set of all assertion friendly attribute names
*
* @return
*/
public Set<String> getFriendlyNames() {
    ...
}

```

```
    }
}
```

### 3.1.7. Error Handling

Keycloak has some error handling facilities for servlet based client adapters. When an error is encountered in authentication, the client adapter will call `HttpServletResponse.sendError()`. You can set up an `error-page` within your `web.xml` file to handle the error however you want. The client adapter can throw 400, 401, 403, and 500 errors.

```
<error-page>
    <error-code>403</error-code>
    <location>/ErrorHandler</location>
</error-page>
```

The client adapter also sets an `HttpServletRequest` attribute that you can retrieve. The attribute name is `org.keycloak.adapters.spi.AuthenticationError`. Typecast this object to: `org.keycloak.adapters.saml.SamlAuthenticationError`. This class can tell you exactly what happened. If this attribute is not set, then the adapter was not responsible for the error code.

```
public class SamlAuthenticationError implements AuthenticationError {
    public static enum Reason {
        EXTRACTION_FAILURE,
        INVALID_SIGNATURE,
        ERROR_STATUS
    }

    public Reason getReason() {
        return reason;
    }
    public StatusResponseType getStatus() {
```

```
        return status;  
    }  
}
```

### 3.1.8. Troubleshooting

The best way to troubleshoot problems is to turn on debugging for SAML in both the client adapter and KeyCloak Server. Using your logging framework, set the log level to `DEBUG` for the `org.keycloak.saml` package. Turning this on allows you to see the SAML requests and response documents being sent to and from the server.

### 3.1.9. Multi Tenancy

SAML offers the same functionality as OIDC for [Multi Tenancy](#), meaning that a single target application (WAR) can be secured with multiple KeyCloak realms. The realms can be located on the same KeyCloak instance or on different instances.

To do this, the application must have multiple `keycloak-saml.xml` adapter configuration files.

While you could have multiple instances of your WAR with different adapter configuration files deployed to different context-paths, this may be inconvenient and you may also want to select the realm based on something other than context-path.

KeyCloak makes it possible to have a custom config resolver, so you can choose which adapter config is used for each request. In SAML, the configuration is only interesting in the login processing; once the user is logged in, the session is authenticated and it does not matter if the `keycloak-saml.xml` returned is different. For that reason, returning the

same configuration for the same session is the correct way to go.

To achieve this, create an implementation of `org.keycloak.adapters.saml.SamlConfigResolver`. The following example uses the `Host` header to locate the proper configuration and load it and the associated elements from the application's Java classpath:

```
package example;

import java.io.InputStream;
import org.keycloak.adapters.saml.SamlConfigResolver;
import org.keycloak.adapters.saml.SamlDeployment;
import org.keycloak.adapters.saml.config.parsers.DeploymentBuilder;
import org.keycloak.adapters.saml.config.parsers.ResourceLoader;
import org.keycloak.adapters.spi.HttpFacade;
import org.keycloak.common.exceptions.ParsingException;

public class SamlMultiTenantResolver implements SamlConfigResolver {

    @Override
    public SamlDeployment resolve(HttpFacade.Request request) {
        String host = request.getHeader("Host");
        String realm = null;
        if (host.contains("tenant1")) {
            realm = "tenant1";
        } else if (host.contains("tenant2")) {
            realm = "tenant2";
        } else {
            throw new IllegalStateException("Not able to
guess the keycloak-saml.xml to load");
        }

        InputStream is = getClass().getResourceAsStream("/")
+ realm + "-keycloak-saml.xml");
        if (is == null) {
```

```

        throw new IllegalStateException("Not able to
find the file /" + realm + "-keycloak-saml.xml");
    }

    ResourceLoader loader = new ResourceLoader() {
        @Override
        public InputStream getResourceAsStream(String
path) {
            return getClass().getResourceAs-
Stream(path);
        }
    };

    try {
        return new DeploymentBuilder().build(is, loader);
    } catch (ParseException e) {
        throw new IllegalStateException("Cannot load
SAML deployment", e);
    }
}
}

```

You must also configure which `SamlConfigResolver` implementation to use with the `keycloak.config.resolver` context-param in your `web.xml`:

```

<web-app>
    ...
    <context-param>
        <param-name>keycloak.config.resolver</param-name>
        <param-value>example.SamlMultiTenantResolver</pa-
ram-value>
    </context-param>
</web-app>

```

## 3.2. mod\_auth\_mellon Apache HTTPD Module

The `mod_auth_mellon` module is an Apache HTTPD plugin for SAML.

If your language/environment supports using Apache HTTPD as a proxy, then you can use `mod_auth_mellon` to secure your web application with SAML. For more details on this module see the [\*mod\\_auth\\_mellon\*](#) GitHub repo.

To configure `mod_auth_mellon` you'll need:

- An Identity Provider (IdP) entity descriptor XML file, which describes the connection to KeyCloak or another SAML IdP
- An SP entity descriptor XML file, which describes the SAML connections and configuration for the application you are securing.
- A private key PEM file, which is a text file in the PEM format that defines the private key the application uses to sign documents.
- A certificate PEM file, which is a text file that defines the certificate for your application.
- `mod_auth_mellon`-specific Apache HTTPD module configuration.

### 3.2.1. Configuring `mod_auth_mellon` with KeyCloak

There are two hosts involved:

- The host on which KeyCloak is running, which will be referred to as `$idp_host` because KeyCloak is a SAML identity provider (IdP).
- The host on which the web application is running, which will be referred to as `$sp_host`. In SAML an application using an IdP is called a service provider (SP).

All of the following steps need to be performed on `$sp_host` with root privileges.

## Installing the Packages

To install the necessary packages, you will need:

- Apache Web Server (httpd)
- Mellon SAML SP add-on module for Apache
- Tools to create X509 certificates

To install the necessary packages, run this command:

```
yum install httpd mod_auth_mellon mod_ssl openssl
```

## Creating a Configuration Directory for Apache SAML

It is advisable to keep configuration files related to Apache's use of SAML in one location.

Create a new directory named saml2 located under the Apache configuration root /etc/httpd:

```
mkdir /etc/httpd/saml2
```

## Configuring the Mellon Service Provider

Configuration files for Apache add-on modules are located in the /etc/httpd/conf.d directory and have a file name extension of .conf. You need to create the /etc/httpd/conf.d/mellon.conf file and place Mellon's configuration directives in it.

Mellan's configuration directives can roughly be broken down into two classes of information:

- Which URLs to protect with SAML authentication
- What SAML parameters will be used when a protected URL is referenced.

Apache configuration directives typically follow a hierarchical tree structure in the URL space, which are known as locations. You need to specify one or more URL locations for Mellon to protect. You have flexibility in how you add the configuration parameters that apply to each location. You can either add all the necessary parameters to the location block or you can add Mellon parameters to a common location high up in the URL location hierarchy that specific protected locations inherit (or some combination of the two). Since it is common for an SP to operate in the same way no matter which location triggers SAML actions, the example configuration used here places common Mellon configuration directives in the root of the hierarchy and then specific locations to be protected by Mellon can be defined with minimal directives. This strategy avoids duplicating the same parameters for each protected location.

This example has just one protected location: `https://$sp_host/protected`.

To configure the Mellon service provider, complete the following steps:

1. Create the file `/etc/httpd/conf.d/mellon.conf` with this content:

```
<Location />
  MellonEnable info
  MellonEndpointPath /mellon/
  MellonSPMetadataFile /etc/httpd/saml2/mellon_metadata.xml
  MellonSPPrivateKeyFile /etc/httpd/saml2/mellon.key
```

```
MellanSPCertFile /etc/httpd/saml2/mellon.crt  
MellanIdPMetadataFile /etc/httpd/saml2/idp_metadata.xml  
</Location>  
<Location /private >  
AuthType Mellon  
MellanEnable auth  
Require valid-user  
</Location>
```

Some of the files referenced in the code above are created in later steps.

## Creating the Service Provider Metadata

In SAML IdPs and SPs exchange SAML metadata, which is in XML format. The schema for the metadata is a standard, thus assuring participating SAML entities can consume each other's metadata. You need:

- Metadata for the IdP that the SP utilizes
- Metadata describing the SP provided to the IdP

One of the components of SAML metadata is X509 certificates. These certificates are used for two purposes:

- Sign SAML messages so the receiving end can prove the message originated from the expected party.
- Encrypt the message during transport (seldom used because SAML messages typically occur on TLS-protected transports)

You can use your own certificates if you already have a Certificate Authority (CA) or you can generate a self-signed certificate. For simplicity in this example a self-signed certificate is used.

Because Mellon's SP metadata must reflect the capabilities of the installed version of mod\_auth\_mellon, must be valid SP metadata XML, and must contain an X509 certificate (whose creation can be obtuse unless you are familiar with X509 certificate generation) the most expedient way to produce the SP metadata is to use a tool included in the mod\_auth\_mellon package (`mellan_create_metadata.sh`). The generated metadata can always be edited later because it is a text file. The tool also creates your X509 key and certificate.

SAML IdPs and SPs identify themselves using a unique name known as an EntityID. To use the Mellon metadata creation tool you need:

- The EntityID, which is typically the URL of the SP, and often the URL of the SP where the SP metadata can be retrieved
- The URL where SAML messages for the SP will be consumed, which Mellon calls the MellonEndPointPath.

To create the SP metadata, complete the following steps:

1. Create a few helper shell variables:

```
fqdn=`hostname`  
mellan_endpoint_url="https://\$fqdn/mellan"  
mellan_entity_id="\$mellan_endpoint_url/metadata"  
file_prefix="\$(echo \"\$mellan_entity_id\" | sed 's/[A-Za-z.]/_/g' | sed 's/_*/_/g')"
```

2. Invoke the Mellon metadata creation tool by running this command:

```
/usr/libexec/mod_auth_mellan/mellan_create_metadata.sh  
\$mellan_entity_id \$mellan_endpoint_url
```

3. Move the generated files to their destination (referenced in the /etc/httpd/conf.d/mellon.conf file created above):

```
mv ${file_prefix}.cert /etc/httpd/saml2/mellon.crt  
mv ${file_prefix}.key /etc/httpd/saml2/mellon.key  
mv ${file_prefix}.xml /etc/httpd/saml2/mellon_metadata.xml
```

## Adding the Mellon Service Provider to the KeyCloak Identity Provider

Assumption: The KeyCloak IdP has already been installed on the \$idp\_host.

KeyCloak supports multiple tenancy where all users, clients, and so on are grouped in what is called a realm. Each realm is independent of other realms. You can use an existing realm in your KeyCloak, but this example shows how to create a new realm called test\_realm and use that realm.

All these operations are performed using the KeyCloak administration web console. You must have the admin username and password for \$idp\_host.

To complete the following steps:

1. Open the Admin Console and log on by entering the admin username and password.

After logging into the administration console there will be an existing realm. When KeyCloak is first set up a root realm, master, is

created by default. Any previously created realms are listed in the upper left corner of the administration console in a drop-down list.

2. From the realm drop-down list select **Add realm**.
3. In the Name field type `test_realm` and click **Create**.

### Adding the Mellon Service Provider as a Client of the Realm

In KeyCloak SAML SPs are known as clients. To add the SP we must be in the Clients section of the realm.

1. Click the Clients menu item on the left and click **Create** in the upper right corner to create a new client.

### Adding the Mellon SP Client

To add the Mellon SP client, complete the following steps:

1. Set the client protocol to SAML. From the Client Protocol drop down list, select **saml**.
2. Provide the Mellon SP metadata file created above (`/etc/httpd/saml2/mellon_metadata.xml`). Depending on where your browser is running you might have to copy the SP metadata from `$sp_host` to the machine on which your browser is running so the browser can find the file.
3. Click **Save**.

### Editing the Mellon SP Client

There are several client configuration parameters we suggest setting:

- Ensure "Force POST Binding" is On.

- Add paosResponse to the Valid Redirect URIs list:
  1. Copy the postResponse URL in "Valid Redirect URIs" and paste it into the empty add text fields just below the "+".
  2. Change "postResponse" to "paosResponse". (The paosResponse URL is needed for SAML ECP.)
  3. Click **Save** at the bottom.

Many SAML SPs determine authorization based on a user's membership in a group. The KeyCloak IdP can manage user group information but it does not supply the user's groups unless the IdP is configured to supply it as a SAML attribute.

To configure the IdP to supply the user's groups as a SAML attribute, complete the following steps:

1. Click the Mappers tab of the client.
2. In the upper right corner of the Mappers page, click **Create**.
3. From the Mapper Type drop-down list select **Group list**.
4. Set Name to "group list".
5. Set the SAML attribute name to "groups".
6. Click **Save**.

The remaining steps are performed on \$sp\_host.

## Retrieving the Identity Provider Metadata

Now that you have created the realm on the IdP you need to retrieve the IdP metadata associated with it so the Mellon SP recognizes it. In the

/etc/httpd/conf.d/mellan.conf file created previously, the MellonIdPMe-tadataFile is specified as /etc/httpd/saml2/idp\_metadata.xml but until now that file has not existed on \$sp\_host. To get that file we will retrieve it from the IdP.

1. Retrieve the file from the IdP by substituting \$idp\_host with the correct value:

```
curl -k -o /etc/httpd/saml2/idp_metadata.xml \
https://$idp_host/auth/realms/test_realm/proto-
col/saml/descriptor
```

Mellan is now fully configured.

2. To run a syntax check for Apache configuration files:

```
apachectl configtest
```

Configtest is equivalent to the -t argument to apachectl. If the configuration test shows any errors, correct them before proceeding.

3. Restart the Apache server:

```
systemctl restart httpd.service
```

You have now set up both KeyCloak as a SAML IdP in the test\_realm and mod\_auth\_mellan as SAML SP protecting the URL \$sp\_host/protected (and everything beneath it) by authenticating against the \$idp\_host IdP.



## 4. Docker Registry Configuration

Docker authentication is disabled by default. To enable see [`{installguide\_profile\_name}`](#).

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

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

### 4.1. Docker Registry Configuration File Installation

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

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

This output can then be copied into any existing registry config file. See the [registry config file specification](#) for more information on how the

file should be set up, or start with [a basic example](#).

Don't forget to configure the `rootcertbundle` field with the location of the Keycloak realm's public certificate. The auth configuration will not work without this argument.

## 4.2. Docker Registry Environment Variable Override Installation

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

```
REGISTRY_AUTH_TOKEN_REALM: http://localhost:8080/auth/realm/master/protocol/docker-v2/auth
REGISTRY_AUTH_TOKEN_SERVICE: docker-test
REGISTRY_AUTH_TOKEN_ISSUER: http://localhost:8080/auth/realm/master
```

Don't forget to configure the `REGISTRY_AUTH_TOKEN_ROOTCERTBUNDLE` override with the location of the Keycloak realm's public certificate. The auth configuration will not work without this argument.

## 4.3. Docker Compose YAML File

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

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

1. From the desired realm, create a client configuration. At this point you won't have a Docker registry - the quickstart will take care of that part.
2. Choose the "Docker Compose YAML" option from the installation tab and download the .zip file
3. Unzip the archive to the desired location, and open the directory.
4. Start the Docker registry with `docker-compose up`

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

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

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

Login Succeeded

---

## 5. Client Registration

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

The Client Registration Service provides built-in support for Keycloak Client Representations, OpenID Connect Client Meta Data and SAML Entity Descriptors. The Client Registration Service endpoint is

```
/auth/realms/<realm>/clients-registrations/<provider>.
```

The built-in supported providers are:

- default - Keycloak Client Representation (JSON)
- install - Keycloak Adapter Configuration (JSON)
- openid-connect - OpenID Connect Client Metadata Description (JSON)
- saml2-entity-descriptor - SAML Entity Descriptor (XML)

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

### 5.1. Authentication

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

cies (see below).

### 5.1.1. Bearer Token

The bearer token can be issued on behalf of a user or a Service Account. The following permissions are required to invoke the endpoints (see [{adminguide!}](#) for more details):

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

If you are using a bearer token to create clients it's recommend to use a token from a Service Account with only the `create-client` role (see [{adminguide!}](#) for more details).

### 5.1.2. Initial Access Token

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

An initial access token can be created through the admin console. To create a new initial access token first select the realm in the admin console, then click on `Realm Settings` in the menu on the left, followed by `Client Registration` in the tabs displayed in the page. Then finally click on `Initial Access Tokens` sub-tab.

You will now be able to see any existing initial access tokens. If you have access you can delete tokens that are no longer required. You can

only retrieve the value of the token when you are creating it. To create a new token click on `Create`. You can now optionally add how long the token should be valid, also how many clients can be created using the token. After you click on `Save` the token value is displayed.

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

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

```
Authorization: bearer eyJhbGciOiJSUZ...
```

### 5.1.3. Registration Access Token

When you create a client through the Client Registration Service the response will include a registration access token. The registration access token provides access to retrieve the client configuration later, but also to update or delete the client. The registration access token is included with the request in the same way as a bearer token or initial access token. Registration access tokens are only valid once, when it's used the response will include a new token.

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

registration access token.

## 5.2. KeyCloak Representations

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

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

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

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

It will also return a new registration access token.

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

It will also return a new registration access token.

To delete the Client Representation perform an HTTP DELETE request

```
to: /auth/realms/<realm>/clients-registrations/de-  
fault/<client id>
```

## 5.3. KeyCloak Adapter Configuration

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

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

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

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

## 5.4. OpenID Connect Dynamic Client Registration

KeyCloak implements [OpenID Connect Dynamic Client Registration](#), which extends [OAuth 2.0 Dynamic Client Registration Protocol](#) and [OAuth 2.0 Dynamic Client Registration Management Protocol](#).

The endpoint to use these specifications to register clients in KeyCloak is `/auth/realms/<realm>/clients-registrations/openid-connect [<client id>]`.

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

## 5.5. SAML Entity Descriptors

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

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

## 5.6. Example using CURL

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

```
curl -X POST \
-d '{ "clientId": "myclient" }' \
-H "Content-Type:application/json" \
-H "Authorization: bearer eyJhbGciOiJSUz..." \
http://localhost:8080/auth/realms/master/clients-registrations/default
```

## 5.7. Example using Java Client Registration API

The Client Registration Java API makes it easy to use the Client Regis-

tration Service using Java. To use include the dependency `org.keycloak:keycloak-client-registration-api:>VERSION<` from Maven.

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

```
String token = "eyJhbGciOiJSUz...";  
  
ClientRepresentation client = new ClientRepresentation();  
client.setClientId(CLIENT_ID);  
  
ClientRegistration reg = ClientRegistration.create()  
    .url("http://localhost:8080/auth", "myrealm")  
    .build();  
  
reg.auth(Auth.token(token));  
  
client = reg.create(client);  
  
String registrationAccessToken = client.getRegistrationAccessToken();
```

## 5.8. Client Registration Policies

KeyCloak currently supports 2 ways how can be new clients registered through Client Registration Service.

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

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

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

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

Currently we have these policy implementations:

- Trusted Hosts Policy - You can configure list of trusted hosts and trusted domains. Request to Client Registration Service can be sent

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

- Consent Required Policy - Newly registered clients will have `Consent Allowed` switch enabled. So after successful authentication, user will always see consent screen when he needs to approve permissions (client scopes). It means that client won't have access to any personal info or permission of user unless user approves it.
- Protocol Mappers Policy - Allows to configure list of whitelisted protocol mapper implementations. New client can't be registered or updated if it contains some non-whitelisted protocol mapper. Note that this policy is used for authenticated requests as well, so even for authenticated request there are some limitations which protocol mappers can be used.
- Client Scope Policy - Allow to whitelist `Client Scopes`, which can be used with newly registered or updated clients. There are no whitelisted scopes by default; only the client scopes, which are defined as `Realm Default Client Scopes` are whitelisted by default.
- Full Scope Policy - Newly registered clients will have `Full Scope Allowed` switch disabled. This means they won't have any scoped realm roles or client roles of other clients.
- Max Clients Policy - Rejects registration if current number of clients in the realm is same or bigger than specified limit. It's 200 by default.

fault for anonymous registrations.

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

---

## 6. Client Registration CLI

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

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

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

To allow a particular user to use `Client Registration CLI` the KeyCloak administrator typically uses the Admin Console to configure a new user with proper roles or to configure a new client and client secret to grant access to the Client Registration REST API.

### 6.1. Configuring a new regular user for use with Client Registration CLI

1. Log in to the Admin Console (for example, <http://localhost:8080/auth/admin>) as `admin`.
2. Select a realm to administer.

3. If you want to use an existing user, select that user to edit; otherwise, create a new user.
4. Select **Role Mappings** > **Client Roles** > **realm-management**. If you are in the master realm, select **NAME-realm**, where **NAME** is the name of the target realm. You can grant access to any other realm to users in the master realm.
5. Select **Available Roles** > **manage-client** to grant a full set of client management permissions. Another option is to choose **view-clients** for read-only or **create-client** to create new clients.

These permissions grant the user the capability to perform operations without the use of [Initial Access Token](#) or [Registration Access Token](#).

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

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

## 6.2. Configuring a client for use with the Client Registration CLI

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

with a user name.

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

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

4. Enable service accounts if you want to use a service account associated with the client by selecting a client to edit in the **Clients** section of the `Admin Console`.
  - a. Under **Settings**, change the `Access Type` to `Confidential`, toggle the **Service Accounts Enabled** setting to **On**, and click **Save**.
  - b. Click **Service Account Roles** and select desired roles to configure the access for the service account. For the details on what roles to select, see [Configuring a new regular user for use with Client Registration CLI](#).
5. Toggle the **Direct Access Grants Enabled** setting it to **On** if you want to use a regular user account instead of a service account.
6. If the client is configured as `Confidential`, provide the configured secret when running `kcreg config credentials` by using the `--secret` option.

7. Specify which `clientId` to use (for example, `--client reg-cli`) when running `kcreg config credentials`.
8. With the service account enabled, you can omit specifying the user when running `kcreg config credentials` and only provide the client secret or keystore information.

## 6.3. Installing the Client Registration CLI

The Client Registration CLI is packaged inside the KeyCloak Server distribution. You can find execution scripts inside the `bin` directory. The Linux script is called `kcreg.sh`, and the Windows script is called `kcreg.bat`.

Add the KeyCloak server directory to your `PATH` when setting up the client for use from any location on the file system.

For example, on:

- Linux:

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

- Windows:

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

`KEYCLOAK_HOME` refers to a directory where the KeyCloak Server distribution was unpacked.

## 6.4. Using the Client Registration CLI

1. Start an authenticated session by logging in with your credentials.
2. Run commands on the `Client Registration REST` endpoint.

For example, on:

- Linux:

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

- Windows:

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

In a production environment, Keycloak has to be accessed with `https:` to avoid exposing tokens to network sniffers.

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

For example, on:

- Linux:

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

- Windows:

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

#### 6.4.1. Logging in

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

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

You cannot configure users with the Client Registration CLI. Use the Admin Console web interface or the Admin Client CLI to configure

users. See [{adminguide!}](#) for more details.

When `kcreg` successfully logs in, it receives authorization tokens and saves them in a private configuration file so the tokens can be used for subsequent invocations. See [Working with alternative configurations](#) for more information on configuration files.

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

For example, on:

- Linux:

```
$ kcreg.sh help
```

- Windows:

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

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

#### 6.4.2. Working with alternative configurations

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

gle configuration file from a single thread.

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

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

#### 6.4.3. Initial Access and Registration Access Tokens

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

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

For example, on:

- Linux:

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

or

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

- Windows:

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

or

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

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

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

See [Client Registration](#) for more information about Initial Access and Registration Access Tokens.

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

#### 6.4.4. Creating a client configuration

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

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

- Linux:

```
$ kcreg.sh create -f client-template.json -s clientId=my-client -s baseUrl=/myclient -s 'redirectUris=["/myclient/*"]' -o
```

- Windows:

```
C:\> kcreg create -f client-template.json -s clientId=my-client -s baseUrl=/myclient -s "redirectUris=[\"/myclient/*\"]" -o
```

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

You can use `kcreg attrs` to list available attributes. Keep in mind

that many configuration attributes are not checked for validity or consistency. It is up to you to specify proper values. Remember that you should not have any id fields in your template and should not specify them as arguments to the `kcreg create` command.

#### 6.4.5. Retrieving a client configuration

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

For example, on:

- Linux:

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

- Windows:

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

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

For example, on:

- Linux:

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

- Windows:

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

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

#### 6.4.6. Modifying a client configuration

There are two methods for updating a client configuration.

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

For example, on:

- Linux:

```
$ kcreg.sh get myclient > myclient.json
$ vi myclient.json
$ kcreg.sh update myclient -f myclient.json
```

- Windows:

```
C:\> kcreg get myclient > myclient.json
C:\> notepad myclient.json
C:\> kcreg update myclient -f myclient.json
```

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

For example, on:

- Linux:

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

- Windows:

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

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

For example, on:

- Linux:

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

- Windows:

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

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

#### 6.4.7. Deleting a client configuration

Use the following example to delete a client.

- Linux:

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

- Windows:

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

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

#### 6.4.8. Refreshing invalid Registration Access Tokens

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

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

the Client Registration CLI automatically handle it for you from that point on.

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

## 6.5. Troubleshooting

- Q: When logging in, I get an error: **Parameter client\_assertion-type is missing [invalid\_client]**.

A: This error means your client is configured with `Signed JWT` token credentials, which means you have to use the `--keystore` parameter when logging in.

---

## 7. Token Exchange

In KeyCloak, token exchange is the process of using a set of credentials or token to obtain an entirely different token. A client may want to invoke on a less trusted application so it may want to downgrade the current token it has. A client may want to exchange a KeyCloak token for a token stored for a linked social provider account. You may want to trust external tokens minted by other KeyCloak realms or foreign IDPs. A client may have a need to impersonate a user. Here's a short summary of the current capabilities of KeyCloak around token exchange.

- A client can exchange an existing KeyCloak token created for a specific client for a new token targeted to a different client
- A client can exchange an existing KeyCloak token for an external token, i.e. a linked Facebook account
- A client can exchange an external token for a KeyCloak token.
- A client can impersonate a user

Token exchange in KeyCloak is a very loose implementation of the [OAuth Token Exchange](#) specification at the IETF. We have extended it a little, ignored some of it, and loosely interpreted other parts of the specification. It is a simple grant type invocation on a realm's OpenID Connect token endpoint.

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

It accepts form parameters (`application/x-www-form-urlencoded`) as input and the output depends on the type of token you requested an exchange for. Token exchange is a client endpoint so requests must provide authentication information for the calling client. Public clients specify their client identifier as a form parameter. Confidential clients can also use form parameters to pass their client id and secret, Basic Auth, or however your admin has configured the client authentication flow in your realm. Here's a list of form parameters

### **client\_id**

*REQUIRED MAYBE.* This parameter is required for clients using form parameters for authentication. If you are using Basic Auth, a client JWT token, or client cert authentication, then do not specify this parameter.

### **client\_secret**

*REQUIRED MAYBE.* This parameter is required for clients using form parameters for authentication and using a client secret as a credential. Do not specify this parameter if client invocations in your realm are authenticated by a different means.

### **grant\_type**

*REQUIRED.* The value of the parameter must be `urn:ietf:parameters:oauth:grant-type:token-exchange`.

### **subject\_token**

*OPTIONAL.* A security token that represents the identity of the party on behalf of whom the request is being made. It is required if you are exchanging an existing token for a new one.

## **subject\_issuer**

*OPTIONAL.* Identifies the issuer of the `subject_token`. It can be left blank if the token comes from the current realm or if the issuer can be determined from the `subject_token_type`. Otherwise it is required to be specified. Valid values are the alias of an `Identity Provider` configured for your realm. Or an issuer claim identifier configured by a specific `Identity Provider`.

## **subject\_token\_type**

*OPTIONAL.* This parameter is the type of the token passed with the `subject_token` parameter. This defaults to `urn:ietf:params:oauth:token-type:access_token` if the `subject_token` comes from the realm and is an access token. If it is an external token, this parameter may or may not have to be specified depending on the requirements of the `subject_issuer`.

## **requested\_token\_type**

*OPTIONAL.* This parameter represents the type of token the client wants to exchange for. Currently only oauth and OpenID Connect token types are supported. The default value for this depends on whether the is `urn:ietf:params:oauth:token-type:refresh_token` in which case you will be returned both an access token and refresh token within the response. Other appropriate values are `urn:ietf:params:oauth:token-type:access_token` and `urn:ietf:params:oauth:token-type:id_token`

## **audience**

*OPTIONAL.* This parameter specifies the target client you want the new token minted for.

## **requested\_issuer**

*OPTIONAL.* This parameter specifies that the client wants a token minted by an external provider. It must be the alias of an `Identity Provider` configured within the realm.

## **requested\_subject**

*OPTIONAL.* This specifies a username or user id if your client wants to impersonate a different user.

## **scope**

*NOT IMPLEMENTED.* This parameter represents the target set of OAuth and OpenID Connect scopes the client is requesting. It is not implemented at this time but will be once KeyCloak has better support for scopes in general.

We currently only support OpenID Connect and OAuth exchanges. Support for SAML based clients and identity providers may be added in the future depending on user demand.

A successful response from an exchange invocation will return the HTTP 200 response code with a content type that depends on the `requested-token-type` and `requested_issuer` the client asks for. OAuth requested token types will return a JSON document as described in the [OAuth Token Exchange](#) specification.

```
{  
  "access_token" : ".....",  
  "refresh_token" : ".....",
```

```
    "expires_in" : "...."
}
```

Clients requesting a refresh token will get back both an access and refresh token in the response. Clients requesting only access token type will only get an access token in the response. Expiration information may or may not be included for clients requesting an external issuer through the `requested_issuer` parameter.

Error responses generally fall under the 400 HTTP response code category, but other error status codes may be returned depending on the severity of the error. Error responses may include content depending on the `requested_issuer`. OAuth based exchanges may return a JSON document as follows:

```
{
  "error" : "...."
  "error_description" : "...."
}
```

Additional error claims may be returned depending on the exchange type. For example, OAuth Identity Providers may include an additional `account-link-url` claim if the user does not have a link to an identity provider. This link can be used for a client initiated link request.

Token exchange setup requires knowledge of fine grain admin permissions (See the [{adminguide!}](#) for more information). You will need to grant clients permission to exchange. This is discussed more later in this chapter.

The rest of this chapter discusses the setup requirements and provides examples for different exchange scenarios. For simplicity's sake, let's call a token minted by the current realm as an *internal* token and a token minted by an external realm or identity provider as an *external* token.

## 7.1. Internal Token to Internal Token Exchange

With an internal token to token exchange you have an existing token minted to a specific client and you want to exchange this token for a new one minted for a different target client. Why would you want to do this? This generally happens when a client has a token minted for itself, and needs to make additional requests to other applications that require different claims and permissions within the access token. Other reasons this type of exchange might be required is if you need to perform a "permission downgrade" where your app needs to invoke on a less trusted app and you don't want to propagate your current access token.

### 7.1.1. Granting Permission for the Exchange

Clients that want to exchange tokens for a different client need to be authorized in the admin console to do so. You'll need to define a `token-exchange` fine grain permission in the target client you want permission to exchange to.

#### *Target Client Permission*

The screenshot shows the Keycloak admin interface for a realm named 'Master'. In the left sidebar, under the 'Configure' section, the 'Clients' option is selected. On the main page, the 'target-client' client is displayed. At the top of the page, there is a navigation bar with tabs: 'Settings', 'Roles', 'Mappers', 'Scope', 'Revocation', 'Sessions', 'Offline Access', 'Installation', and 'Permissions'. The 'Permissions' tab is currently active. Below the tabs, there is a section titled 'Permissions Enabled' with a toggle switch labeled 'OFF'. The rest of the page is mostly empty.

Toggle the `Permissions Enabled` switch to ON.

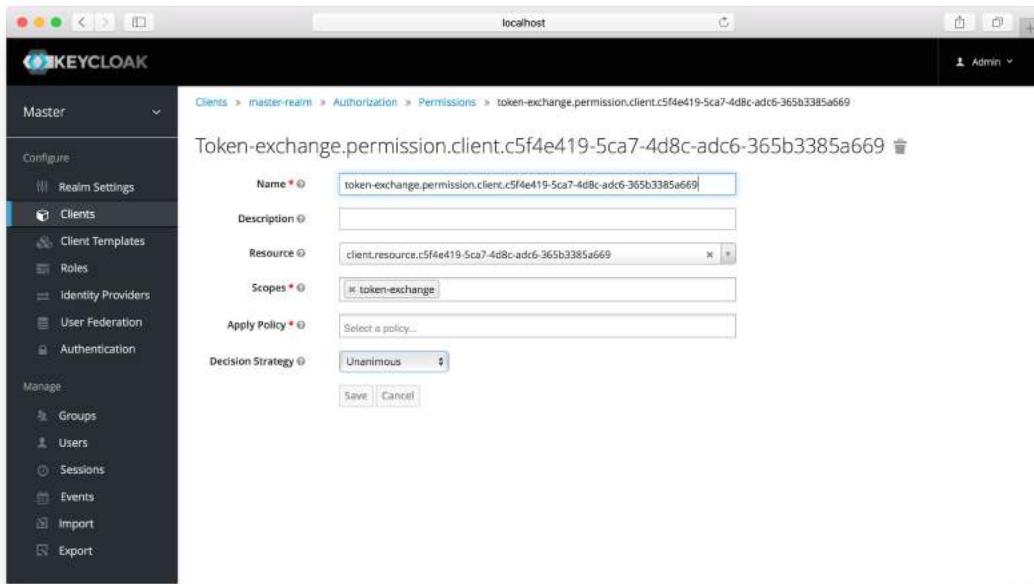
### *Target Client Permission*

The screenshot shows the same Keycloak admin interface as the previous one, but with a key difference: the 'Permissions Enabled' switch is now set to 'ON'. This change is reflected in the table below, which lists various permission scopes and their descriptions. The table has columns for 'scope-name', 'Description', and 'Actions'.

scope-name	Description	Actions
view	Policies that decide if an admin can view this client	Edit
manage	Policies that decide if an admin can manage this client	Edit
configure	Reduced management permissions for admin. Cannot set scope, template, or protocol mappers.	Edit
map-roles	Policies that decide if an admin can map roles defined by this client	Edit
map-roles-client-scope	Policies that decide if an admin can apply roles defined by this client to the client scope of another client	Edit
map-roles-composite	Policies that decide if an admin can apply roles defined by this client as a composite to another role	Edit
token-exchange	Policies that decide which clients are allowed exchange tokens for a token that is targeted to this client.	Edit

You should see a `token-exchange` link on the page. Click that to start defining the permission. It will bring you to this page.

## Target Client Exchange Permission Setup



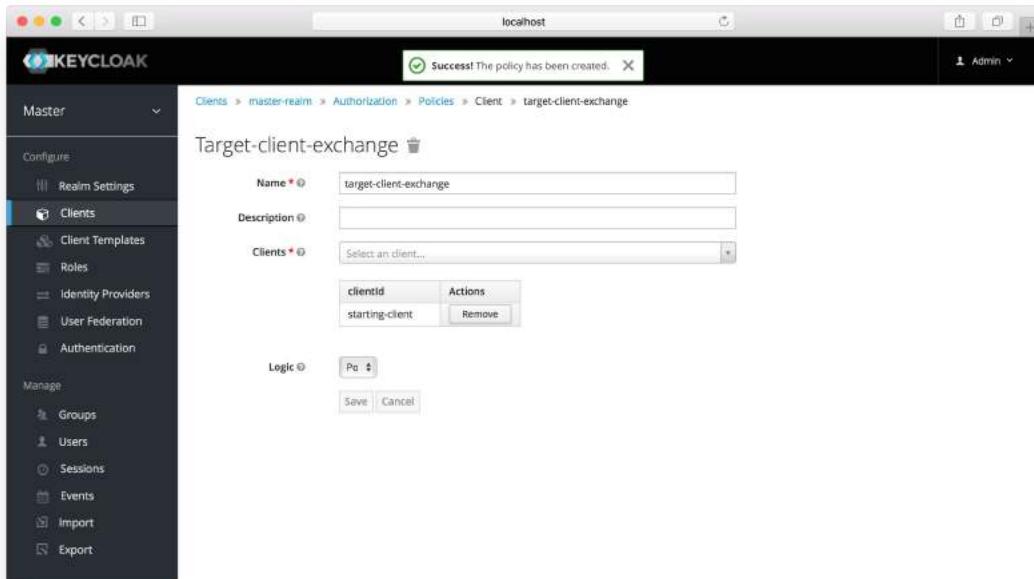
The screenshot shows the Keycloak administration interface on a Mac OS X system. The left sidebar is titled 'Master' and contains sections for 'Configure' (Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication) and 'Manage' (Groups, Users, Sessions, Events, Import, Export). The 'Clients' section is currently selected. The main content area shows a form for creating a permission named 'token-exchange.permission.client.c5f4e419-5ca7-4d8c-adc6-365b3385a669'. The form fields include:

- Name: token-exchange.permission.client.c5f4e419-5ca7-4d8c-adc6-365b3385a669
- Description: (empty)
- Resource: client.resource.c5f4e419-5ca7-4d8c-adc6-365b3385a669
- Scopes: token-exchange
- Apply Policy: Select a policy...
- Decision Strategy: Unanimous

At the bottom right of the form are 'Save' and 'Cancel' buttons.

You'll have to define a policy for this permission. Click the `Authorization` link, go to the `Policies` tab and create a `Client` Policy.

## Client Policy Creation



The screenshot shows the Keycloak administration interface on a Mac OS X system. The left sidebar is titled 'Master' and contains sections for 'Configure' (Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication) and 'Manage' (Groups, Users, Sessions, Events, Import, Export). The 'Clients' section is currently selected. The main content area shows a form for creating a client policy named 'target-client-exchange'. The form fields include:

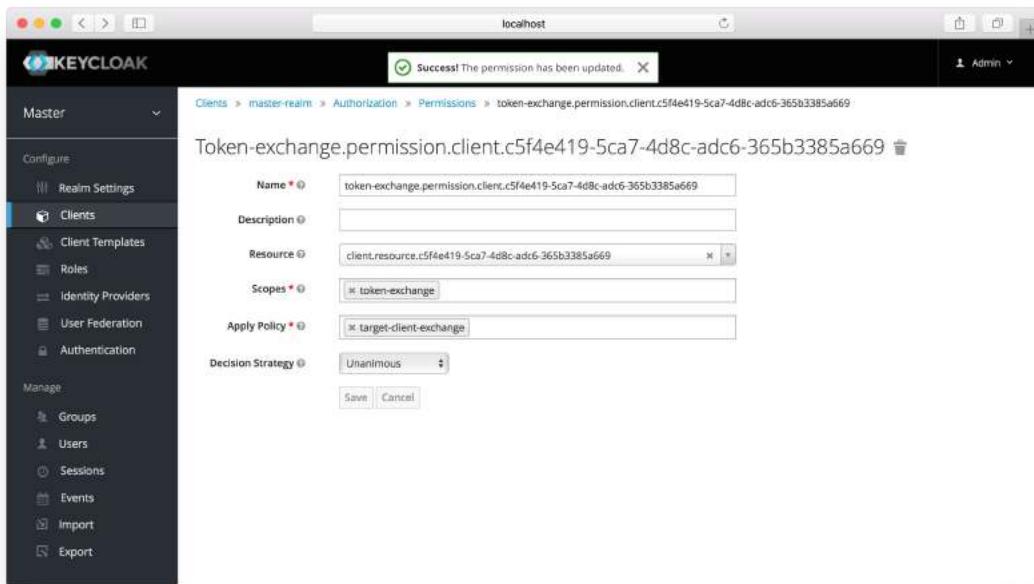
- Name: target-client-exchange
- Description: (empty)
- Clients: Select an client... (dropdown menu showing 'starting-client')
- Logic: (radio button group) (selected: 'And')

At the bottom right of the form are 'Save' and 'Cancel' buttons. A success message at the top right of the screen says 'Success! The policy has been created.'

Here you enter in the starting client, that is the authenticated client that

is requesting a token exchange. After you create this policy, go back to the target client's `token-exchange` permission and add the client policy you just defined.

## Apply Client Policy

A screenshot of the Keycloak admin console. The left sidebar shows 'Clients' selected under 'Configure'. The main content area shows a form for creating a permission named 'token-exchange.permission.client.c5f4e419-5ca7-4d8c-adc6-365b3385a669'. The form fields include: Name (token-exchange.permission.client.c5f4e419-5ca7-4d8c-adc6-365b3385a669), Description (empty), Resource (client.resource.c5f4e419-5ca7-4d8c-adc6-365b3385a669), Scopes (token-exchange), Apply Policy (target-client-exchange), and Decision Strategy (Unanimous). A success message at the top says 'Success! The permission has been updated.' and there is a 'Save' button at the bottom.

Your client now has permission to invoke. If you do not do this correctly, you will get a 403 Forbidden response if you try to make an exchange.

### 7.1.2. Making the Request

When your client is exchanging an existing token for a token targeting another client, you must use the `audience` parameter. This parameter must be the client identifier for the target client that you configured in the admin console.

```
curl -X POST \
-d "client_id=starting-client" \
-d "client_secret=geheim" \
--data-urlencode "grant_type=urn:ietf:pa-
```

```
  rams:oauth:grant-type:token-exchange" \
    -d "subject_token=...." \
    --data-urlencode "requested_token_type=urn:ietf:pa-
  rams:oauth:token-type:refresh_token" \
    -d "audience=target-client" \
    http://localhost:8080/auth/realms/myrealm/proto-
  col/openid-connect/token
```

The `subject_token` parameter must be an access token for the target realm. If your `requested_token_type` parameter is a refresh token type, then the response will contain both an access token, refresh token, and expiration. Here's an example JSON response you get back from this call.

```
{
  "access_token" : "...",
  "refresh_token" : "...",
  "expires_in" : 3600
}
```

## 7.2. Internal Token to External Token Exchange

You can exchange a realm token for an external token minted by an external identity provider. This external identity provider must be configured within the `Identity Provider` section of the admin console.

Currently only OAuth/OpenID Connect based external identity providers are supported, this includes all social providers. Keycloak does not perform a backchannel exchange to the external provider. So if the account is not linked, you will not be able to get the external token. To be able to obtain an external token one of these conditions must be met:

- The user must have logged in with the external identity provider at least once

- The user must have linked with the external identity provider through the User Account Service
- The user account was linked through the external identity provider using [Client Initiated Account Linking API](#).

Finally, the external identity provider must have been configured to store tokens, or, one of the above actions must have been performed with the same user session as the internal token you are exchanging.

If the account is not linked, the exchange response will contain a link you can use to establish it. This is discussed more in the [Making the Request](#) section.

### 7.2.1. Granting Permission for the Exchange

Internal to external token exchange requests will be denied with a 403, Forbidden response until you grant permission for the calling client to exchange tokens with the external identity provider. To grant permission to the client you must go to the identity provider's configuration page to the `Permissions` tab.

#### *Identity Provider Permission*

The screenshot shows the Keycloak admin interface with the URL `localhost:8080/auth/admin/realms/master/identity-providers/google`. The left sidebar is titled 'Master' and includes sections for Configure (Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication) and Manage (Groups, Users, Sessions, Events, Import, Export). The main content area is titled 'google' and shows the 'Permissions' tab selected. Under 'Permissions Enabled', there is a switch labeled 'OFF'. Below the switch is a table with one row:

scope-name	Description	Actions
token-exchange	Policies that decide which clients are allowed exchange tokens for an external token minted by this identity provider.	Edit

Toggle the `Permissions Enabled` switch to true.

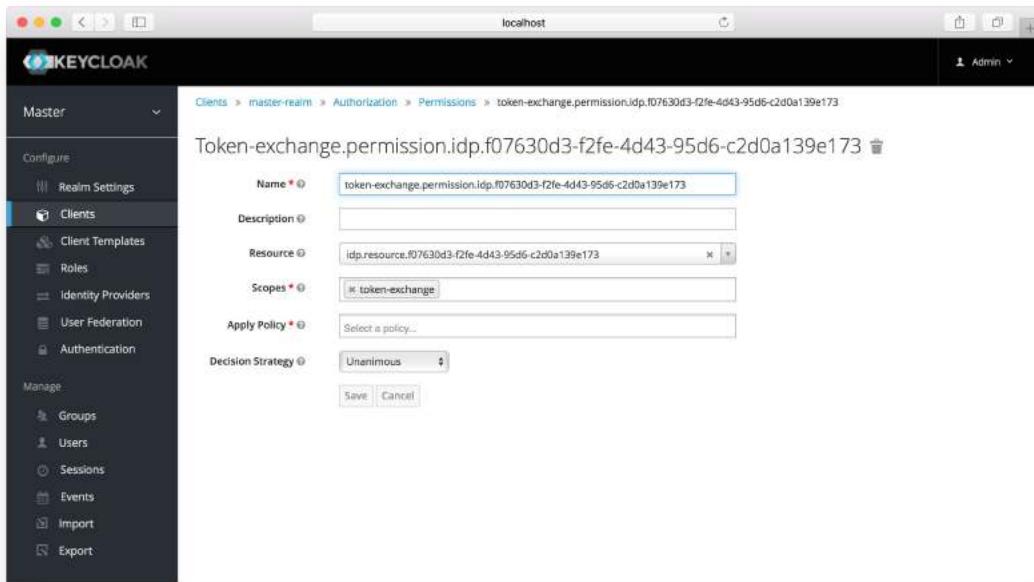
### *Identity Provider Permission*

The screenshot shows the same Keycloak admin interface as the previous one, but with the 'Permissions Enabled' switch now set to 'ON'. The table below the switch now contains two rows:

scope-name	Description	Actions
token-exchange	Policies that decide which clients are allowed exchange tokens for an external token minted by this identity provider.	Edit
profile	Policies that decide which clients are allowed to access user profile information.	Edit

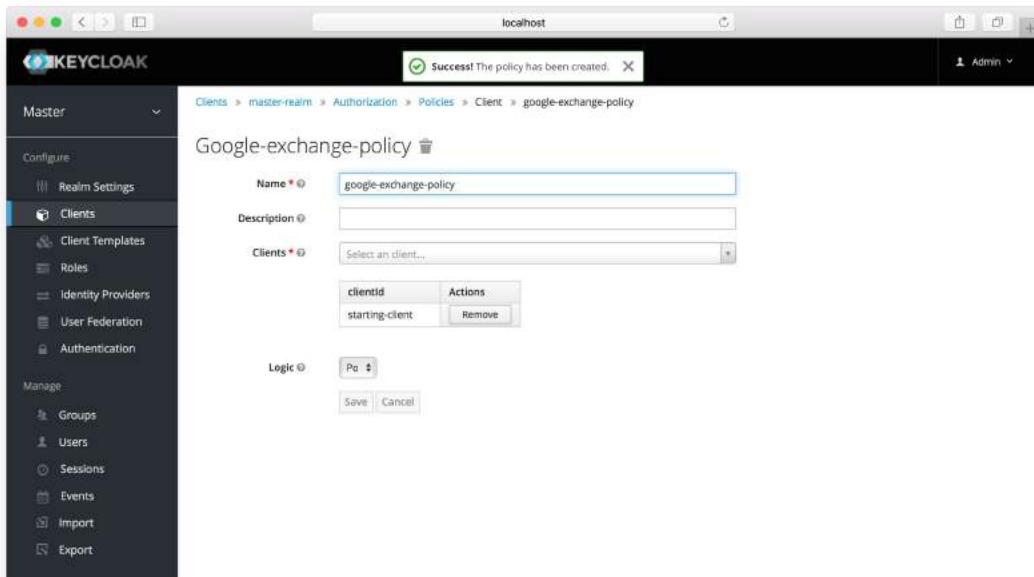
You should see a `token-exchange` link on the page. Click that to start defining the permission. It will bring you to this page.

## Identity Provider Exchange Permission Setup



You'll have to define a policy for this permission. Click the `Authorization` link, go to the `Policies` tab and create a `Client` Policy.

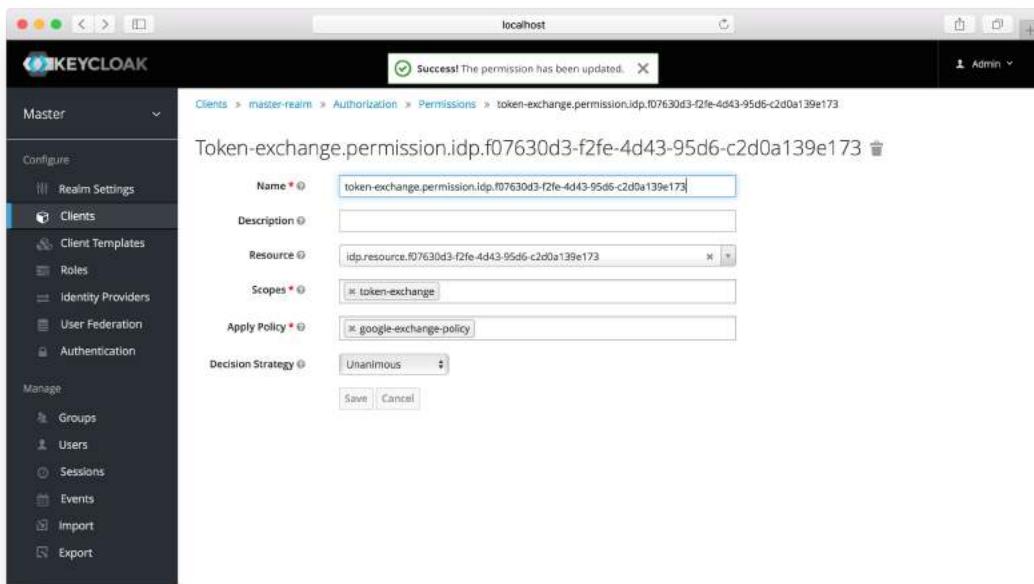
## Client Policy Creation



Here you enter in the starting client, that is the authenticated client that

is requesting a token exchange. After you create this policy, go back to the identity providers's `token-exchange` permission and add the client policy you just defined.

## Apply Client Policy



The screenshot shows the Keycloak administration interface on a Mac OS X system. The window title is "localhost". The top navigation bar has a success message: "Success! The permission has been updated." and a user dropdown "Admin". The left sidebar menu is open, showing "Master" and several sections: "Configure" (selected), "Clients", "Client Templates", "Roles", "Identity Providers", "User Federation", and "Authentication". Under "Configure", there are also "Groups", "Users", "Sessions", "Events", "Import", and "Export". The main content area is titled "Token-exchange.permission.idp.f07630d3-f2fe-4d43-95d6-c2d0a139e173". It contains the following form fields:

- Name: token-exchange.permission.idp.f07630d3-f2fe-4d43-95d6-c2d0a139e173
- Description: (empty)
- Resource: idp.resource.f07630d3-f2fe-4d43-95d6-c2d0a139e173
- Scopes: token-exchange
- Apply Policy: google-exchange-policy
- Decision Strategy: Unanimous

At the bottom right are "Save" and "Cancel" buttons.

Your client now has permission to invoke. If you do not do this correctly, you will get a 403 Forbidden response if you try to make an exchange.

### 7.2.2. Making the Request

When your client is exchanging an existing internal token to an external one, you must provide the `requested_issuer` parameter. The parameter must be the alias of a configured identity provider.

```
curl -X POST \
  -d "client_id=starting-client" \
  -d "client_secret=geheim" \
  --data-urlencode "grant_type=urn:ietf:params:oauth:grant-type:token-exchange" \
```

```
-d "subject_token=...." \
--data-urlencode "requested_token_type=urn:ietf:params:oauth:token-type:access_token" \
-d "requested_issuer=google" \
http://localhost:8080/auth/realms/myrealm/protocol/openid-connect/token
```

The `subject_token` parameter must be an access token for the target realm. The `requested_token_type` parameter must be `urn:ietf:params:oauth:token-type:access_token` or left blank. No other requested token type is supported at this time. Here's an example successful JSON response you get back from this call.

```
{
  "access_token" : "....",
  "expires_in" : 3600
  "account-link-url" : "https://...."
}
```

If the external identity provider is not linked for whatever reason, you will get an HTTP 400 response code with this JSON document:

```
{
  "error" : "....",
  "error_description" : "...",
  "account-link-url" : "https://...."
}
```

The `error` claim will be either `token_expired` or `not_linked`. The `account-link-url` claim is provided so that the client can perform [Client Initiated Account Linking](#). Most (all?) providers are requiring linking through browser OAuth protocol. With the `account-link-url` just add a `redirect_uri` query parameter to it and you

can forward browsers to perform the link.

### 7.3. External Token to Internal Token Exchange

You can trust and exchange external tokens minted by external identity providers for internal tokens. This can be used to bridge between realms or just to trust tokens from your social provider. It works similarly to an identity provider browser login in that a new user is imported into your realm if it doesn't exist.

The current limitation on external token exchanges is that if the external token maps to an existing user an exchange will not be allowed unless the existing user already has an account link to the external identity provider.

When the exchange is complete, a user session will be created within the realm, and you will receive an access and or refresh token depending on the `requested_token_type` parameter value. You should note that this new user session will remain active until it times out or until you call the logout endpoint of the realm passing this new access token.

These types of changes required a configured identity provider in the admin console.

SAML identity providers are not supported at this time.  
Twitter tokens cannot be exchanged either.

### 7.3.1. Granting Permission for the Exchange

Before external token exchanges can be done, you must grant permission for the calling client to make the exchange. This permission is granted in the same manner as [internal to external permission is granted](#).

If you also provide an `audience` parameter whose value points to a different client other than the calling one, you must also grant the calling client permission to exchange to the target client specific in the `audience` parameter. How to do this is [discussed earlier](#) in this section.

### 7.3.2. Making the Request

The `subject_token_type` must either be `urn:ietf:params:oauth:token-type:access_token` or `urn:ietf:params:oauth:token-type:jwt`. If the type is `urn:ietf:params:oauth:token-type:access_token` you must specify the `subject_issuer` parameter and it must be the alias of the configured identity provider. If the type is `urn:ietf:params:oauth:token-type:jwt`, the provider will be matched via the `issuer` claim within the JWT which must be the alias of the provider, or a registered issuer within the providers configuration.

For validation, if the token is an access token, the provider's user info service will be invoked to validate the token. A successful call will mean that the access token is valid. If the subject token is a JWT and if the provider has signature validation enabled, that will be attempted, otherwise, it will default to also invoking on the user info service to validate the token.

By default, the internal token minted will use the calling client to determine what's in the token using the protocol mappers defined for the calling client. Alternatively, you can specify a different target client using the `audience` parameter.

```
curl -X POST \
  -d "client_id=starting-client" \
  -d "client_secret=geheim" \
  --data-urlencode "grant_type=urn:ietf:params:oauth:grant-type:token-exchange" \
  -d "subject_token=...." \
  -d "subject_issuer=myOidcProvider" \
  --data-urlencode "subject_token_type=urn:ietf:params:oauth:token-type:access_token" \
  -d "audience=target-client" \
  http://localhost:8080/auth/realms/myrealm/protocol/openid-connect/token
```

If your `requested_token_type` parameter is a refresh token type, then the response will contain both an access token, refresh token, and expiration. Here's an example JSON response you get back from this call.

```
{
  "access_token" : "...",
  "refresh_token" : "...",
  "expires_in" : 3600
}
```

## 7.4. Impersonation

For internal and external token exchanges, the client can request on behalf of a user to impersonate a different user. For example, you may have an admin application that needs to impersonate a user so that a

support engineer can debug a problem.

### 7.4.1. Granting Permission for the Exchange

The user that the subject token represents must have permission to impersonate other users. See the [{adminguide!}](#) on how to enable this permission. It can be done through a role or through fine grain admin permissions.

### 7.4.2. Making the Request

Make the request as described in other chapters except additionally specify the `request_subject` parameter. The value of this parameter must be a username or user id.

```
curl -X POST \
  -d "client_id=starting-client" \
  -d "client_secret=geheim" \
  --data-urlencode "grant_type=urn:ietf:parameters:oauth:grant-type:token-exchange" \
  -d "subject_token=...." \
  --data-urlencode "requested_token_type=urn:ietf:parameters:oauth:token-type:access_token" \
  -d "audience=target-client" \
  -d "requested_subject=wburke" \
  http://localhost:8080/auth/realm/myrealm/protocol/openid-connect/token
```

## 7.5. Direct Naked Impersonation

You can make an internal token exchange request without providing a `subject_token`. This is called a direct naked impersonation because it places a lot of trust in a client as that client can impersonate any user in the realm. You might need this to bridge for applications where it is impossible to obtain a subject token to exchange. For example, you may

be integrating a legacy application that performs login directly with LDAP. In that case, the legacy app is able to authenticate users itself, but not able to obtain a token.

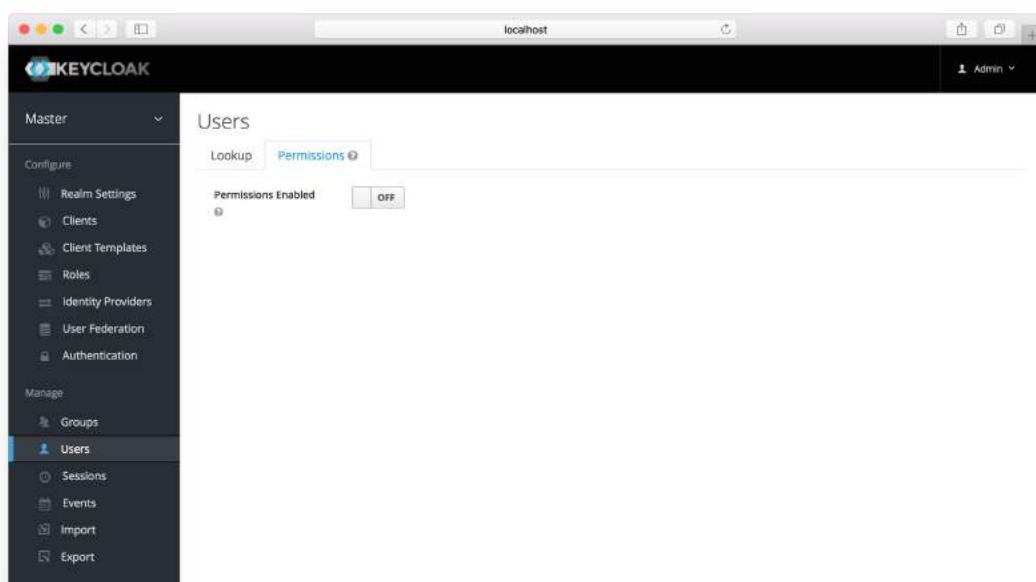
It is very risky to enable direct naked impersonation for a client. If the client's credentials are ever stolen, that client can impersonate any user in the system.

### 7.5.1. Granting Permission for the Exchange

If the `audience` parameter is provided, then the calling client must have permission to exchange to the client. How to set this up is discussed earlier in this chapter.

Additionally, the calling client must be granted permission to impersonate users. In the admin console, go to the `Users` screen and click on the `Permissions` tab.

#### *Users Permission*



Toggle the `Permissions Enabled` switch to true.

## Identity Provider Permission

scope-name	Description	Actions
view	Policies that decide if an admin can view all users in realm	Edit
manage	Policies that decide if an admin can manage all users in the realm	Edit
map-roles	Policies that decide if admin can map roles for all users	Edit
manage-group-membership	Policies that decide if an admin can manage group membership for all users in the realm. This is used in conjunction with specific group policy	Edit
impersonate	Policies that decide if admin can impersonate other users	Edit
user-impersonated	Policies that decide which users can be impersonated. These policies are applied to the user being impersonated.	Edit

You should see a `Impersonation` link on the page. Click that to start defining the permission. It will bring you to this page.

## Users Impersonation Permission Setup

Clients > master-realm > Authorization > Permissions > admin-impersonating.permission.users

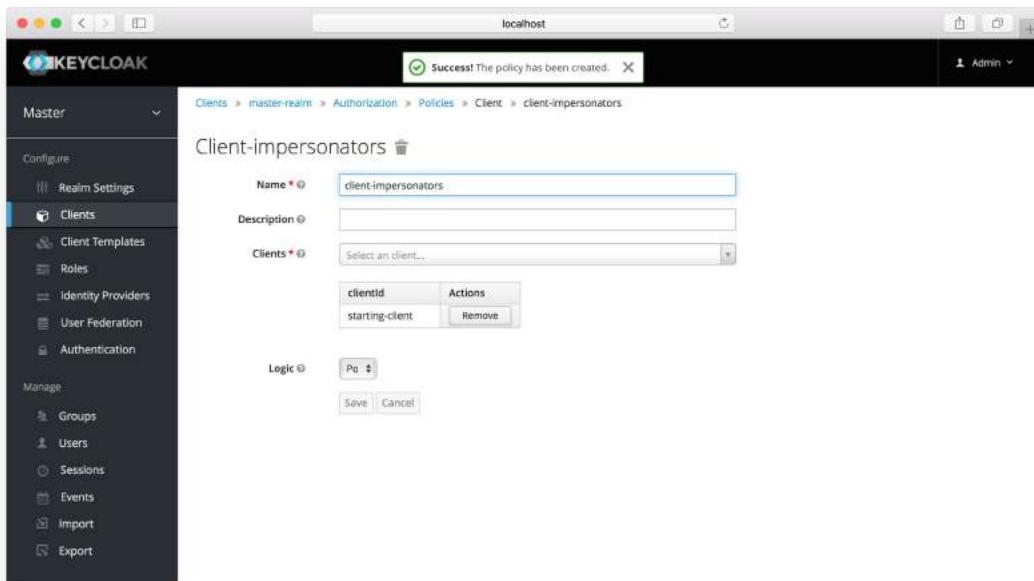
Admin-impersonating.permission.users

Name *	admin-impersonating.permission.users
Description	
Resource	Users
Scopes *	impersonate
Apply Policy *	Select a policy...
Decision Strategy	Unanimous

Save Cancel

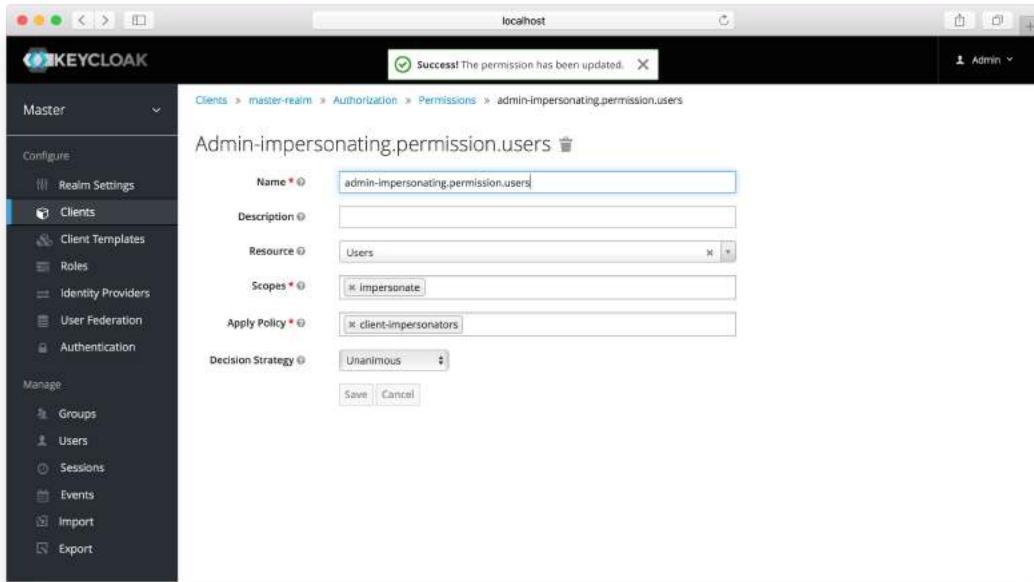
You'll have to define a policy for this permission. Click the `Authorization` link, go to the `Policies` tab and create a `Client` Policy.

### *Client Policy Creation*



Here you enter in the starting client, that is the authenticated client that is requesting a token exchange. After you create this policy, go back to the users' `impersonation` permission and add the client policy you just defined.

### *Apply Client Policy*



Your client now has permission to impersonate users. If you do not do this correctly, you will get a 403 Forbidden response if you try to make this type of exchange.

Public clients are not allowed to do direct naked impersonations.

### 7.5.2. Making the Request

To make the request, simply specify the `requested_subject` parameter. This must be the username or user id of a valid user. You can also specify an `audience` parameter if you wish.

```
curl -X POST \
  -d "client_id=starting-client" \
  -d "client_secret=geheim" \
  --data-urlencode "grant_type=urn:ietf:params:oauth:grant-type:token-exchange" \
  -d "requested_subject=wburke" \
  http://localhost:8080/auth/realms/myrealm/proto-
```

## 7.6. Expand Permission Model With Service Accounts

When granting clients permission to exchange, you don't necessarily have to manually enable those permissions for each and every client. If the client has a service account associated with it, you can use a role to group permissions together and assign exchange permissions by assigning a role to the client's service account. For example, you might define a `naked-exchange` role and any service account that has that role can do a naked exchange.

## 7.7. Exchange Vulnerabilities

When you start allowing token exchanges, there's various things you have to both be aware of and careful of.

The first is public clients. Public clients do not have or require a client credential in order to perform an exchange. Anybody that has a valid token will be able to *impersonate* the public client and perform the exchanges that public client is allowed to perform. If there are any untrustworthy clients that are managed by your realm, public clients may open up vulnerabilities in your permission models. This is why direct naked exchanges do not allow public clients and will abort with an error if the calling client is public.

It is possible to exchange social tokens provided by Facebook, Google, etc. for a realm token. Be careful and vigilante on what the exchange token is allowed to do as its not hard to create fake accounts on these social websites. Use default roles, groups, and identity provider map-

pers to control what attributes and roles are assigned to the external social user.

Direct naked exchanges are quite dangerous. You are putting a lot of trust in the calling client that it will never leak out its client credentials. If those credentials are leaked, then the thief can impersonate anybody in your system. This is in direct contrast to confidential clients that have existing tokens. You have two factors of authentication, the access token and the client credentials, and you're only dealing with one user. So use direct naked exchanges sparingly.

Last updated 2019-06-13 12:48:56 MESZ

# Table of Contents

{adminguide!}

## 1. Overview

### 1.1. Features

### 1.2. How Does Security Work?

### 1.3. Core Concepts and Terms

## 2. Server Initialization

## 3. Admin Console

### 3.1. The Master Realm

### 3.2. Create a New Realm

### 3.3. SSL Mode

### 3.4. Clearing Server Caches

### 3.5. Email Settings

### 3.6. Themes and Internationalization

#### 3.6.1. Internationalization

## 4. User Management

### 4.1. Searching For Users

### 4.2. Creating New Users

### 4.3. Deleting Users

### 4.4. User Attributes

### 4.5. User Credentials

#### 4.5.1. Changing Passwords

#### 4.5.2. Changing OTPs

### 4.6. Required Actions

#### 4.6.1. Default Required Actions

#### 4.6.2. Terms and Conditions

### 4.7. Impersonation

## 4.8. User Registration

### 4.8.1. reCAPTCHA Support

## 5. Login Page Settings

### 5.1. Forgot Password

### 5.2. Remember Me

## 6. Authentication

### 6.1. Password Policies

#### 6.1.1. Password Policy Types

### 6.2. OTP Policies

#### 6.2.1. TOTP vs. HOTP

#### 6.2.2. TOTP Configuration Options

#### 6.2.3. HOTP Configuration Options

### 6.3. Authentication Flows

### 6.4. Kerberos

#### 6.4.1. Setup of Kerberos server

#### 6.4.2. Setup and configuration of KeyCloak server

#### 6.4.3. Setup and configuration of client machines

#### 6.4.4. Credential Delegation

#### 6.4.5. Cross-realm trust

#### 6.4.6. Troubleshooting

### 6.5. X.509 Client Certificate User Authentication

#### 6.5.1. Features

#### 6.5.2. Enable X.509 Client Certificate User Authentication

#### 6.5.3. Adding X.509 Client Certificate Authentication to a Browser Flow

#### 6.5.4. Adding X.509 Client Certificate Authentication to a Direct Grant Flow

#### 6.5.5. Client certificate lookup

#### 6.5.6. Troubleshooting

## 7. SSO Protocols

### 7.1. OpenID Connect

#### 7.1.1. OIDC Auth Flows

#### 7.1.2. KeyCloak Server OIDC URI Endpoints

### 7.2. SAML

#### 7.2.1. SAML Bindings

#### 7.2.2. KeyCloak Server SAML URI Endpoints

### 7.3. OpenID Connect vs. SAML

### 7.4. Docker Registry v2 Authentication

#### 7.4.1. Docker Auth Flow

#### 7.4.2. KeyCloak Docker Registry v2 Authentication Server URI Endpoints

## 8. Managing Clients

### 8.1. OIDC Clients

#### 8.1.1. Advanced Settings

#### 8.1.2. Confidential Client Credentials

#### 8.1.3. Service Accounts

#### 8.1.4. Audience Support

### 8.2. SAML Clients

#### 8.2.1. IDP Initiated Login

#### 8.2.2. SAML Entity Descriptors

### 8.3. Client Links

### 8.4. OIDC Token and SAML Assertion Mappings

#### 8.4.1. Priority order

#### 8.4.2. OIDC User Session Note Mappers

### 8.5. Generating Client Adapter Config

### 8.6. Client Scopes

#### 8.6.1. Protocol

- 8.6.2. Consent related settings
- 8.6.3. Link Client Scope with the Client
- 8.6.4. Evaluating Client Scopes
- 8.6.5. Client Scopes Permissions
- 8.6.6. Realm Default Client Scopes
- 8.6.7. Scopes explained

## 9. Roles

- 9.1. Realm Roles
- 9.2. Client Roles
- 9.3. Composite Roles
- 9.4. User Role Mappings
  - 9.4.1. Default Roles
- 9.5. Role Scope Mappings

## 10. Groups

- 10.1. Groups vs. Roles
- 10.2. Default Groups

## 11. Admin Console Access Control and Permissions

- 11.1. Master Realm Access Control
  - 11.1.1. Global Roles
  - 11.1.2. Realm Specific Roles
- 11.2. Dedicated Realm Admin Consoles
- 11.3. Fine Grain Admin Permissions
  - 11.3.1. Managing One Specific Client
  - 11.3.2. Restrict User Role Mapping
  - 11.3.3. Full List of Permissions
- 11.4. Realm Keys
  - 11.4.1. Rotating keys
  - 11.4.2. Adding a generated keypair

- 11.4.3. Adding an existing keypair and certificate
- 11.4.4. Loading keys from a Java Keystore
- 11.4.5. Making keys passive
- 11.4.6. Disabling keys
- 11.4.7. Compromised keys

## 12. Identity Brokering

- 12.1. Brokering Overview
- 12.2. Default Identity Provider
- 12.3. General Configuration
- 12.4. Social Identity Providers
  - 12.4.1. Bitbucket
  - 12.4.2. Facebook
  - 12.4.3. GitHub
  - 12.4.4. GitLab
  - 12.4.5. Google
  - 12.4.6. LinkedIn
  - 12.4.7. Microsoft
  - 12.4.8. OpenShift
  - 12.4.9. PayPal
  - 12.4.10. Stack Overflow
  - 12.4.11. Twitter
- 12.5. OpenID Connect v1.0 Identity Providers
- 12.6. SAML v2.0 Identity Providers
  - 12.6.1. SP Descriptor
- 12.7. Client-suggested Identity Provider
- 12.8. Mapping Claims and Assertions
- 12.9. Available User Session Data
- 12.10. First Login Flow

- 12.10.1. Default First Login Flow
  - 12.10.2. Automatically Link Existing First Login Flow
- 12.11. Retrieving External IDP Tokens
- 12.12. Identity broker logout
- 13. User Session Management
  - 13.1. Administering Sessions
    - 13.1.1. Logout All Limitations
    - 13.1.2. Application Drilldown
    - 13.1.3. User Drilldown
  - 13.2. Revocation Policies
  - 13.3. Session and Token Timeouts
  - 13.4. Offline Access
- 14. User Storage Federation
  - 14.1. Adding a Provider
  - 14.2. Dealing with Provider Failures
  - 14.3. LDAP and Active Directory
    - 14.3.1. Storage Mode
    - 14.3.2. Edit Mode
    - 14.3.3. Other config options
    - 14.3.4. Connect to LDAP over SSL
    - 14.3.5. Sync of LDAP users to Keycloak
    - 14.3.6. LDAP Mappers
    - 14.3.7. Password Hashing
  - 14.4. SSSD and FreeIPA Identity Management Integration
    - 14.4.1. FreeIPA/IdM Server
    - 14.4.2. SSSD and D-Bus
    - 14.4.3. Enabling the SSSD Federation Provider
  - 14.5. Configuring a Federated SSSD Store

- 14.6. Custom Providers
- 15. Auditing and Events
  - 15.1. Login Events
    - 15.1.1. Event Types
    - 15.1.2. Event Listener
  - 15.2. Admin Events
- 16. Export and Import
  - 16.1. Admin console export/import
- 17. User Account Service
  - 17.1. Themeable
- 18. Threat Model Mitigation
  - 18.1. Host
    - 18.1.1. Request provider
    - 18.1.2. Fixed provider
    - 18.1.3. Custom provider
  - 18.2. Admin Endpoints and Console
    - 18.2.1. IP Restriction
    - 18.2.2. Port Restriction
  - 18.3. Password guess: brute force attacks
    - 18.3.1. Password Policies
  - 18.4. Clickjacking
  - 18.5. SSL/HTTPS Requirement
  - 18.6. CSRF Attacks
  - 18.7. Unspecific Redirect URIs
  - 18.8. Compromised Access and Refresh Tokens
  - 18.9. Compromised Authorization Code
  - 18.10. Open redirectors
  - 18.11. Password database compromised

[18.12. Limiting Scope](#)

[18.13. Limit Token Audience](#)

[18.14. SQL Injection Attacks](#)

## [19. The Admin CLI](#)

[19.1. Installing the Admin CLI](#)

[19.2. Using the Admin CLI](#)

[19.3. Authenticating](#)

[19.4. Working with alternative configurations](#)

[19.5. Basic operations and resource URIs](#)

[19.6. Realm operations](#)

[19.7. Role operations](#)

[19.8. Client operations](#)

[19.9. User operations](#)

[19.10. Group operations](#)

[19.11. Identity provider operations](#)

[19.12. Storage provider operations](#)

[19.13. Adding mappers](#)

[19.14. Authentication operations](#)

---

{adminguide!}

# 1. Overview

KeyCloak is a single sign on solution for web apps and RESTful web services. The goal of KeyCloak is to make security simple so that it is easy for application developers to secure the apps and services they have deployed in their organization. Security features that developers normally have to write for themselves are provided out of the box and are easily tailorable to the individual requirements of your organization. KeyCloak provides customizable user interfaces for login, registration, administration, and account management. You can also use KeyCloak as an integration platform to hook it into existing LDAP and Active Directory servers. You can also delegate authentication to third party identity providers like Facebook and Google+.

## 1.1. Features

- Single-Sign On and Single-Sign Out for browser applications.
- OpenID Connect support.
- OAuth 2.0 support.
- SAML support.
- Identity Brokering - Authenticate with external OpenID Connect or SAML Identity Providers.
- Social Login - Enable login with Google, GitHub, Facebook, Twitter, and other social networks.
- User Federation - Sync users from LDAP and Active Directory servers.

- Kerberos bridge - Automatically authenticate users that are logged-in to a Kerberos server.
- Admin Console for central management of users, roles, role mappings, clients and configuration.
- Account Management console that allows users to centrally manage their account.
- Theme support - Customize all user facing pages to integrate with your applications and branding.
- Two-factor Authentication - Support for TOTP/HOTP via Google Authenticator or FreeOTP.
- Login flows - optional user self-registration, recover password, verify email, require password update, etc.
- Session management - Admins and users themselves can view and manage user sessions.
- Token mappers - Map user attributes, roles, etc. how you want into tokens and statements.
- Not-before revocation policies per realm, application and user.
- CORS support - Client adapters have built-in support for CORS.
- Supports any platform/language that has an OpenID Connect Relying Party library or SAML 2.0 Service Provider library.

## 1.2. How Does Security Work?

KeyCloak is a separate server that you manage on your network. Applications are configured to point to and be secured by this server. KeyCloak uses open protocol standards like [OpenID Connect](#) or [SAML 2.0](#) to

secure your applications. Browser applications redirect a user's browser from the application to the Keycloak authentication server where they enter their credentials. This is important because users are completely isolated from applications and applications never see a user's credentials. Applications instead are given an identity token or assertion that is cryptographically signed. These tokens can have identity information like username, address, email, and other profile data. They can also hold permission data so that applications can make authorization decisions. These tokens can also be used to make secure invocations on REST-based services.

## 1.3. Core Concepts and Terms

There are some key concepts and terms you should be aware of before attempting to use Keycloak to secure your web applications and REST services.

### **users**

Users are entities that are able to log into your system. They can have attributes associated with themselves like email, username, address, phone number, and birth day. They can be assigned group membership and have specific roles assigned to them.

### **authentication**

The process of identifying and validating a user.

### **authorization**

The process of granting access to a user.

### **credentials**

Credentials are pieces of data that Keycloak uses to verify the identity of a user. Some examples are passwords, one-time-passwords, digital certificates, or even fingerprints.

## roles

Roles identify a type or category of user. `Admin`, `user`, `manager`, and `employee` are all typical roles that may exist in an organization. Applications often assign access and permissions to specific roles rather than individual users as dealing with users can be too fine grained and hard to manage.

## user role mapping

A user role mapping defines a mapping between a role and a user. A user can be associated with zero or more roles. This role mapping information can be encapsulated into tokens and assertions so that applications can decide access permissions on various resources they manage.

## composite roles

A composite role is a role that can be associated with other roles. For example a `superuser` composite role could be associated with the `sales-admin` and `order-entry-admin` roles. If a user is mapped to the `superuser` role they also inherit the `sales-admin` and `order-entry-admin` roles.

## groups

Groups manage groups of users. Attributes can be defined for a group. You can map roles to a group as well. Users that become members of a group inherit the attributes and role mappings that

group defines.

## **realms**

A realm manages a set of users, credentials, roles, and groups. A user belongs to and logs into a realm. Realms are isolated from one another and can only manage and authenticate the users that they control.

## **clients**

Clients are entities that can request KeyCloak to authenticate a user. Most often, clients are applications and services that want to use KeyCloak to secure themselves and provide a single sign-on solution. Clients can also be entities that just want to request identity information or an access token so that they can securely invoke other services on the network that are secured by KeyCloak.

## **client adapters**

Client adapters are plugins that you install into your application environment to be able to communicate and be secured by KeyCloak. KeyCloak has a number of adapters for different platforms that you can download. There are also third-party adapters you can get for environments that we don't cover.

## **consent**

Consent is when you as an admin want a user to give permission to a client before that client can participate in the authentication process. After a user provides their credentials, KeyCloak will pop up a screen identifying the client requesting a login and what identity information is requested of the user. User can decide whether or not to

grant the request.

## **client scopes**

When a client is registered, you must define protocol mappers and role scope mappings for that client. It is often useful to store a client scope, to make creating new clients easier by sharing some common settings. This is also useful for requesting some claims or roles to be conditionally based on the value of `scope` parameter. Keycloak provides the concept of a client scope for this.

## **client role**

Clients can define roles that are specific to them. This is basically a role namespace dedicated to the client.

## **identity token**

A token that provides identity information about the user. Part of the OpenID Connect specification.

## **access token**

A token that can be provided as part of an HTTP request that grants access to the service being invoked on. This is part of the OpenID Connect and OAuth 2.0 specification.

## **assertion**

Information about a user. This usually pertains to an XML blob that is included in a SAML authentication response that provided identity metadata about an authenticated user.

## **service account**

Each client has a built-in service account which allows it to obtain an access token.

### **direct grant**

A way for a client to obtain an access token on behalf of a user via a REST invocation.

### **protocol mappers**

For each client you can tailor what claims and assertions are stored in the OIDC token or SAML assertion. You do this per client by creating and configuring protocol mappers.

### **session**

When a user logs in, a session is created to manage the login session. A session contains information like when the user logged in and what applications have participated within single-sign on during that session. Both admins and users can view session information.

### **user federation provider**

KeyCloak can store and manage users. Often, companies already have LDAP or Active Directory services that store user and credential information. You can point KeyCloak to validate credentials from those external stores and pull in identity information.

### **identity provider**

An identity provider (IDP) is a service that can authenticate a user. KeyCloak is an IDP.

### **identity provider federation**

Keycloak can be configured to delegate authentication to one or more IDPs. Social login via Facebook or Google+ is an example of identity provider federation. You can also hook Keycloak to delegate authentication to any other OpenID Connect or SAML 2.0 IDP.

## identity provider mappers

When doing IDP federation you can map incoming tokens and assertions to user and session attributes. This helps you propagate identity information from the external IDP to your client requesting authentication.

## required actions

Required actions are actions a user must perform during the authentication process. A user will not be able to complete the authentication process until these actions are complete. For example, an admin may schedule users to reset their passwords every month. An `update password` required action would be set for all these users.

## authentication flows

Authentication flows are work flows a user must perform when interacting with certain aspects of the system. A login flow can define what credential types are required. A registration flow defines what profile information a user must enter and whether something like re-CAPTCHA must be used to filter out bots. Credential reset flow defines what actions a user must do before they can reset their password.

## events

Events are audit streams that admins can view and hook into.

## **themes**

Every screen provided by KeyCloak is backed by a theme. Themes define HTML templates and stylesheets which you can override as needed.

## 2. Server Initialization

After performing all the installation and configuration tasks defined in the [{installguide!}](#), you will need to create an initial admin account. KeyCloak does not have any configured admin account out of the box. This account will allow you to create an admin that can log into the *master* realm's administration console so that you can start creating realms, users and registering applications to be secured by KeyCloak.

If your server is accessible from `localhost`, you can boot it up and create this admin user by going to the <http://localhost:8080/auth> URL.

*Welcome Page*



### Welcome to Keycloak

Please create an initial admin user to get started.

Username	<input type="text"/>
Password	<input type="password"/>
Password confirmation	<input type="password"/>
<input type="button" value="Create"/>	

[Documentation](#) | [Administration Console](#)

[Keycloak Project](#) | [Mailing List](#) | [Report an issue](#)



Simply specify the username and password you want for this initial admin.

If you cannot access the server via a `localhost` address, or just want to provision Keycloak from the command line you can do this with the `.../bin/add-user-keycloak` script.

### *add-user-keycloak script*



The parameters are a little different depending if you are using the standalone operation mode or domain operation mode. For standalone mode, here is how you use the script.

### *Linux/Unix*

```
$ .../bin/add-user-keycloak.sh -r master -u <username> -p  
<password>
```

### *Windows*

```
> ...\\bin\\add-user-keycloak.bat -r master -u <username> -p  
<password>
```

For domain mode, you have to point the script to one of your server hosts using the `-sc` switch.

### *Linux/Unix*

```
$ .../bin/add-user-keycloak.sh --sc domain/servers/server-one/configuration -r master -u <username> -p <password>
```

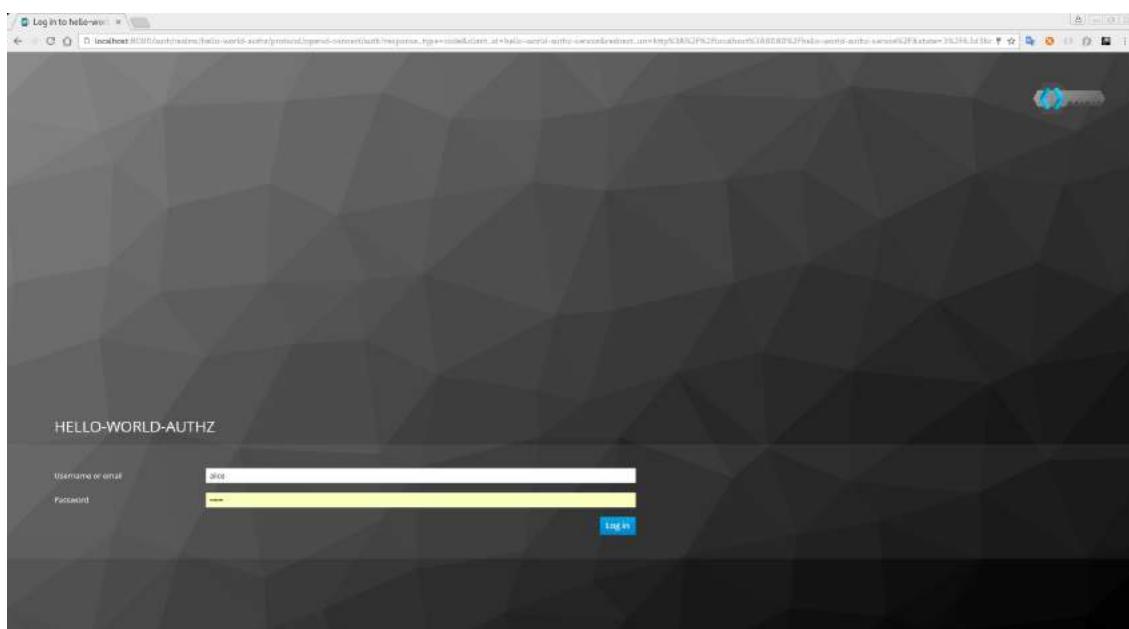
### *Windows*

```
> ...\\bin\\add-user-keycloak.bat --sc domain/servers/server-one/configuration -r master -u <username> -p <password>
```

## 3. Admin Console

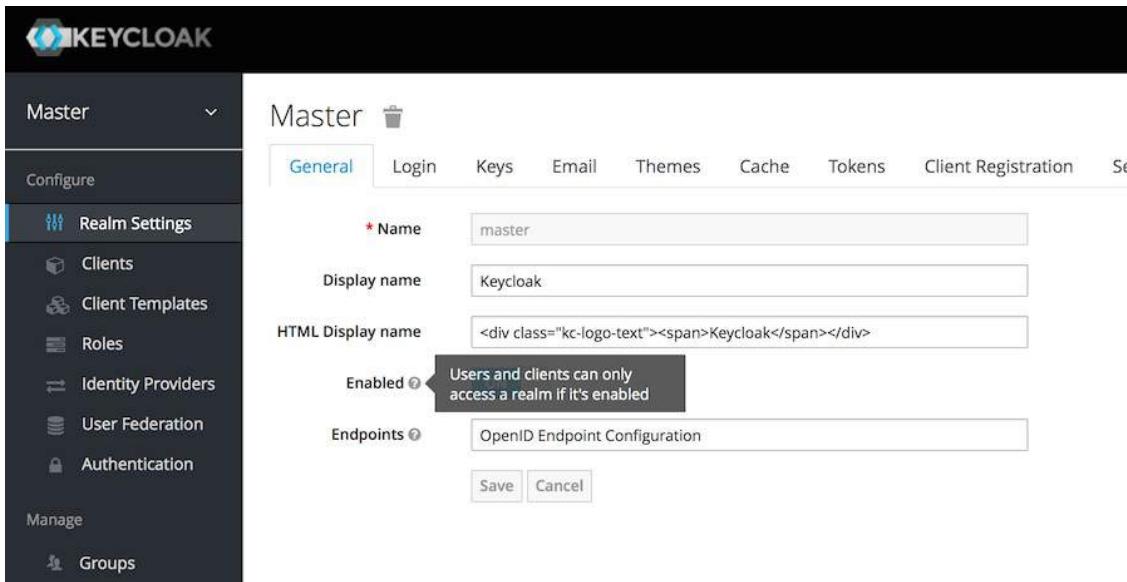
The bulk of your administrative tasks will be done through the Keycloak Admin Console. You can go to the console url directly at <http://localhost:8080/auth/admin/>

### *Login Page*



Enter the username and password you created on the Welcome Page or the `add-user-keycloak` script in the bin directory. This will bring you to the Keycloak Admin Console.

### *Admin Console*



The left drop down menu allows you to pick a realm you want to manage or to create a new one. The right drop down menu allows you to view your user account or logout. If you are curious about a certain feature, button, or field within the Admin Console, simply hover your mouse over any question mark ? icon. This will pop up tooltip text to describe the area of the console you are interested in. The image above shows the tooltip in action.

### 3.1. The Master Realm

When you boot KeyCloak for the first time KeyCloak creates a pre-defined realm for you. This initial realm is the *master* realm. It is the highest level in the hierarchy of realms. Admin accounts in this realm have permissions to view and manage any other realm created on the server instance. When you define your initial admin account, you create an account in the *master* realm. Your initial login to the admin console will also be via the *master* realm.

We recommend that you do not use the *master* realm to manage the users and applications in your organization. Reserve use of the *master*

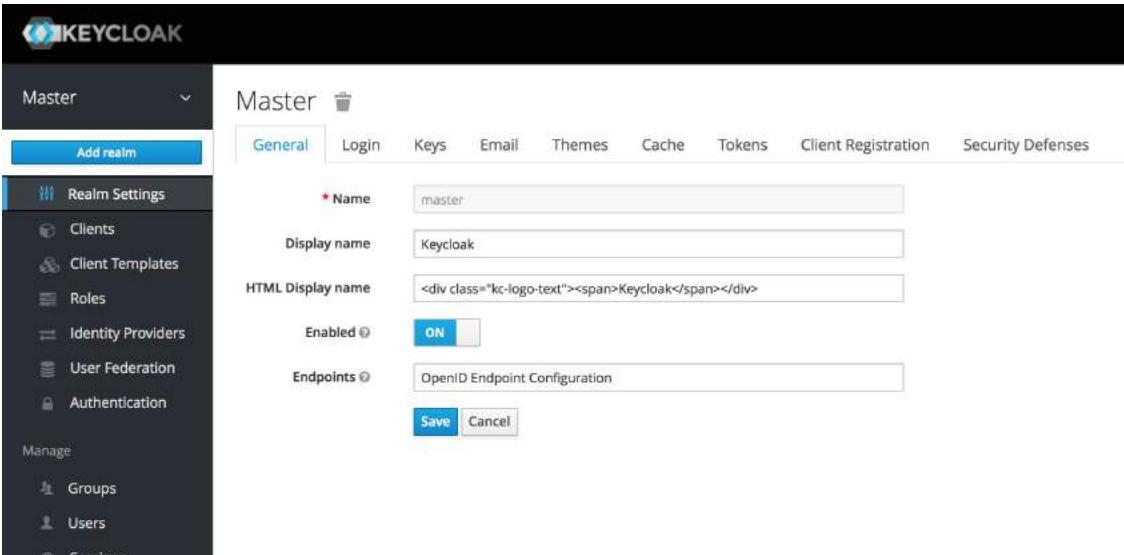
realm for *super* admins to create and manage the realms in your system. Following this security model helps prevent accidental changes and follows the tradition of permitting user accounts access to only those privileges and powers necessary for the successful completion of their current task.

It is possible to disable the *master* realm and define admin accounts within each individual new realm you create. Each realm has its own dedicated Admin Console that you can log into with local accounts. This guide talks more about this in the [Dedicated Realm Admin Consoles](#) chapter.

## 3.2. Create a New Realm

Creating a new realm is very simple. Mouse over the top left corner drop down menu that is titled with `Master`. If you are logged in the master realm this drop down menu lists all the realms created. The last entry of this drop down menu is always `Add Realm`. Click this to add a realm.

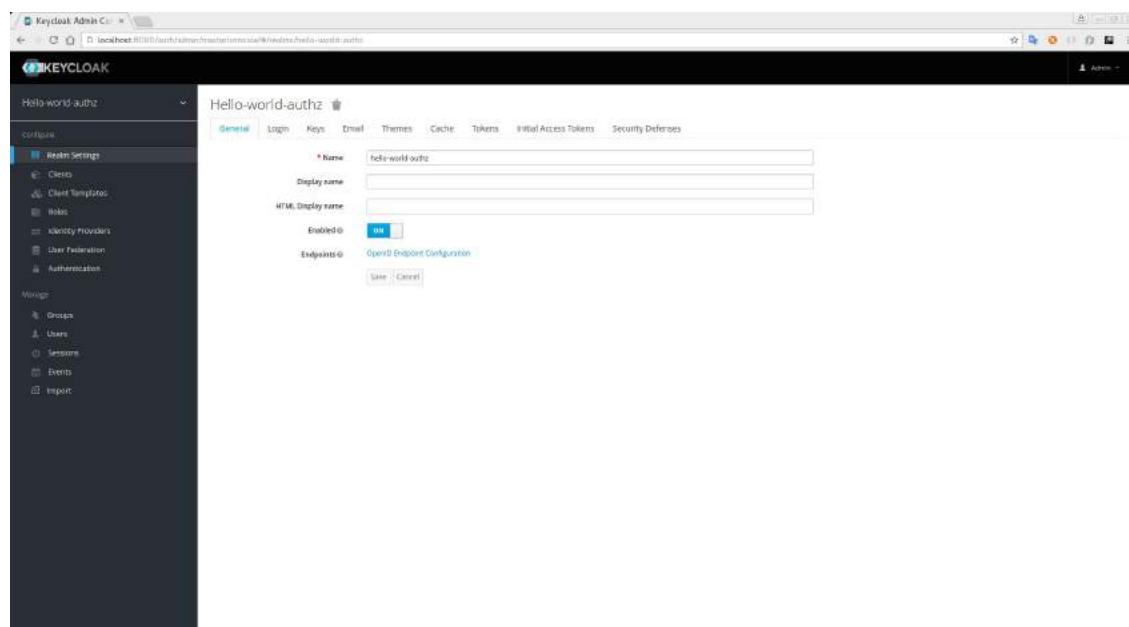
### Add Realm Menu



The screenshot shows the Keycloak Admin Console interface. The top navigation bar has a logo and the word "KEYCLOAK". Below it, a dropdown menu is open under the "Master" label, showing a list of realms: "master", "test-realm", and "Add Realm". The "Add Realm" option is highlighted with a blue background. The main content area is titled "Master" and shows the configuration for the "master" realm. The "General" tab is selected, displaying fields for "Name" (set to "master"), "Display name" (set to "Keycloak"), "HTML Display name" (containing the Keycloak logo), "Enabled" (set to "ON"), and "Endpoints" (set to "OpenID Endpoint Configuration"). At the bottom are "Save" and "Cancel" buttons. On the left side, there's a sidebar with sections like "Realm Settings", "Clients", "Client Templates", "Roles", "Identity Providers", "User Federation", "Authentication", "Manage", "Groups", "Users", and "Sessions".

This menu option will bring you to the `Add Realm` page. Specify the realm name you want to define and click the `Create` button. Alternatively you can import a JSON document that defines your new realm. We'll go over this in more detail in the [Export and Import](#) chapter.

### *Create Realm*



After creating the realm you are brought back to the main Admin Console page. The current realm will now be set to the realm you just created. You can switch between managing different realms by doing a mouse over on the top left corner drop down menu.

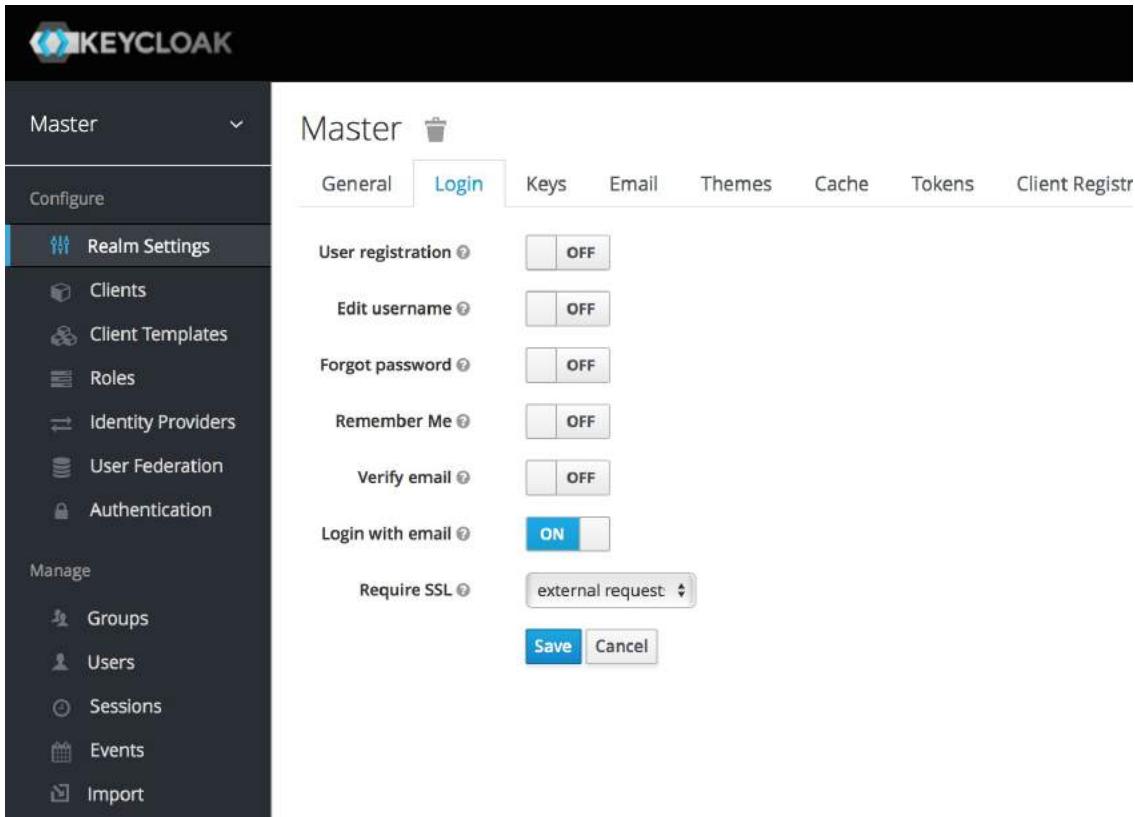
### **3.3. SSL Mode**

Each realm has an SSL Mode associated with it. The SSL Mode defines the SSL/HTTPS requirements for interacting with the realm. Browsers and applications that interact with the realm must honor the SSL/HTTPS requirements defined by the SSL Mode or they will not be allowed to interact with the server.

KeyCloak generates a self-signed certificate the first time it runs. Please note that self-signed certificates are not secure, and should only be used for testing purposes. It is highly recommended that you install a CA-signed certificate on the KeyCloak server itself or on a reverse proxy in front of the KeyCloak server. See the [{installguide!}](#).

To configure the SSL Mode of your realm, you need to click on the `Realm Settings` left menu item and go to the `Login` tab.

### *Login Tab*



The screenshot shows the Keycloak Admin UI interface. The top navigation bar has the Keycloak logo and the word "KEYCLOAK". Below it, the left sidebar is titled "Master" and contains the following menu items:

- Configure
  - Realm Settings** (selected)
  - Clients
  - Client Templates
  - Roles
  - Identity Providers
  - User Federation
  - Authentication
- Manage
  - Groups
  - Users
  - Sessions
  - Events
  - Import

The main content area is titled "Master" and shows the "Login" tab selected. The "Login" tab has the following configuration options:

Setting	Value
User registration	<input type="checkbox"/> OFF
Edit username	<input type="checkbox"/> OFF
Forgot password	<input type="checkbox"/> OFF
Remember Me	<input type="checkbox"/> OFF
Verify email	<input type="checkbox"/> OFF
Login with email	<input checked="" type="checkbox"/> ON
Require SSL	external request

At the bottom right of the form are "Save" and "Cancel" buttons.

The `Require SSL` option allows you to pick the SSL Mode you want. Here is an explanation of each mode:

## **external requests**

Users can interact with Keycloak without SSL so long as they stick to private IP addresses like `localhost`, `127.0.0.1`, `10.0.x.x`, `192.168.x.x`, and `172.16.x.x`. If you try to access Keycloak without SSL from a non-private IP address you will get an error.

## **none**

Keycloak does not require SSL. This should really only be used in development when you are playing around with things and don't want to bother configuring SSL on your server.

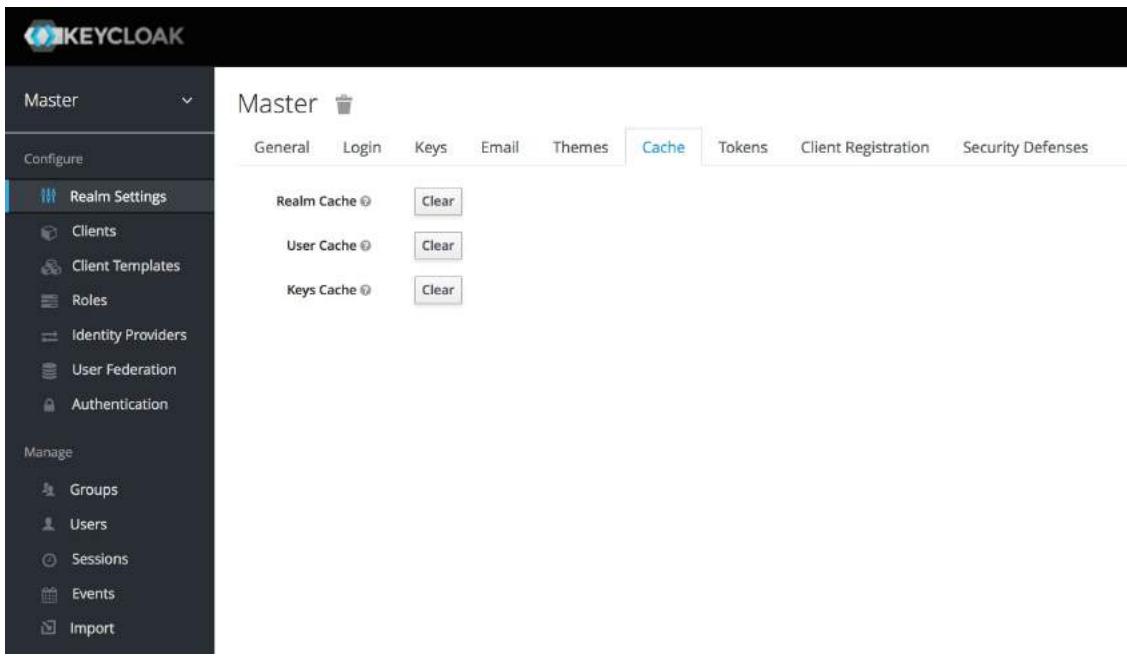
## **all requests**

Keycloak requires SSL for all IP addresses.

## **3.4. Clearing Server Caches**

Keycloak will cache everything it can in memory within the limits of your JVM and/or the limits you've configured it for. If the Keycloak database is modified by a third party (i.e. a DBA) outside the scope of the server's REST APIs or Admin Console there's a chance parts of the in-memory cache may be stale. You can clear the realm cache, user cache or cache of external public keys (Public keys of external clients or Identity providers, which Keycloak usually uses to verify signatures of particular external entity) from the Admin Console by going to the `Realm Settings` left menu item and the `Cache` tab.

*Cache tab*



Just click the `clear` button on the cache you want to evict.

### 3.5. Email Settings

KeyCloak sends emails to users to verify their email address, when they forget their passwords, or when an admin needs to receive notifications about a server event. To enable KeyCloak to send emails you need to provide KeyCloak with your SMTP server settings. This is configured per realm. Go to the `Realm Settings` left menu item and click the `Email` tab.

#### *Email Tab*

The screenshot shows the Keycloak Admin UI with the 'Master' realm selected. The left sidebar has 'Realm Settings' selected. The top navigation bar includes 'General', 'Login', 'Keys', 'Email' (which is active and highlighted in blue), 'Themes', 'Cache', 'Tokens', 'Client Registration', and 'Security Defenses'. The main content area is titled 'Master' and contains the 'Email' configuration section. It includes fields for 'Host' (SMTP Host), 'Port' (SMTP Port defaults to 25), 'From' (Sender Email Address), and toggle switches for 'Enable SSL', 'Enable StartTLS', and 'Enable Authentication', all of which are currently set to 'OFF'. At the bottom are 'Save' and 'Cancel' buttons.

## Host

**Host** denotes the SMTP server hostname used for sending emails.

## Port

**Port** denotes the SMTP server port.

## From

**From** denotes the address used for the **From** SMTP-Header for the emails sent.

## From Display Name

**From Display Name** allows to configure a user friendly email address aliases (optional). If not set the plain **From** email address will be displayed in email clients.

## Reply To

**Reply To** denotes the address used for the **Reply-To** SMTP-Header for the mails sent (optional). If not set the plain **From** email address will be used.

## Reply To Display Name

`Reply To Display Name` allows to configure a user friendly email address aliases (optional). If not set the plain `Reply To` email address will be displayed.

## Envelope From

`Envelope From` denotes the [Bounce Address](#) used for the `Return-Path` SMTP-Header for the mails sent (optional).

As emails are used for recovering usernames and passwords it's recommended to use SSL or TLS, especially if the SMTP server is on an external network. To enable SSL click on `Enable SSL` or to enable TLS click on `Enable TLS`. You will most likely also need to change the `Port` (the default port for SSL/TLS is 465).

If your SMTP server requires authentication click on `Enable Authentication` and insert the `Username` and `Password`.

## 3.6. Themes and Internationalization

Themes allow you to change the look and feel of any UI in Keycloak. Themes are configured per realm. To change a theme go to the `Realm Settings` left menu item and click on the `Themes` tab.

### *Themes Tab*

The screenshot shows the Keycloak Admin UI. On the left is a sidebar with a 'Master' dropdown, 'Configure' button, and several icons for 'Realm Settings' (Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication), 'Manage' (Groups), and 'Groups'. The main area is titled 'Master' with a trash icon. It has tabs for General, Login, Keys, Email, Themes (which is selected and highlighted in blue), and Cache. Under the Themes tab, there are four dropdown menus: 'Login Theme' (Select one...), 'Account Theme' (Select one...), 'Admin Console Theme' (Select one...), and 'Email Theme' (Select one...). Below these is a 'Internationalization Enabled' switch set to 'OFF'. At the bottom are 'Save' and 'Cancel' buttons.

Pick the theme you want for each UI category and click **Save**.

## Login Theme

Username password entry, OTP entry, new user registration, and other similar screens related to login.

## Account Theme

Each user has an User Account Management UI.

## Admin Console Theme

The skin of the KeyCloak Admin Console.

## Email Theme

Whenever KeyCloak has to send out an email, it uses templates defined in this theme to craft the email.

The [{developerguide!}](#) goes into how to create a new themes or modify existing ones.

### 3.6.1. Internationalization

Every UI screen is internationalized in Keycloak. The default language is English, but if you turn on the `Internationalization` switch on the `Theme` tab you can choose which locales you want to support and what the default locale will be. The next time a user logs in, they will be able to choose a language on the login page to use for the login screens, User Account Management UI, and Admin Console. The [{developer-guide!}](#) explains how you can offer additional languages.

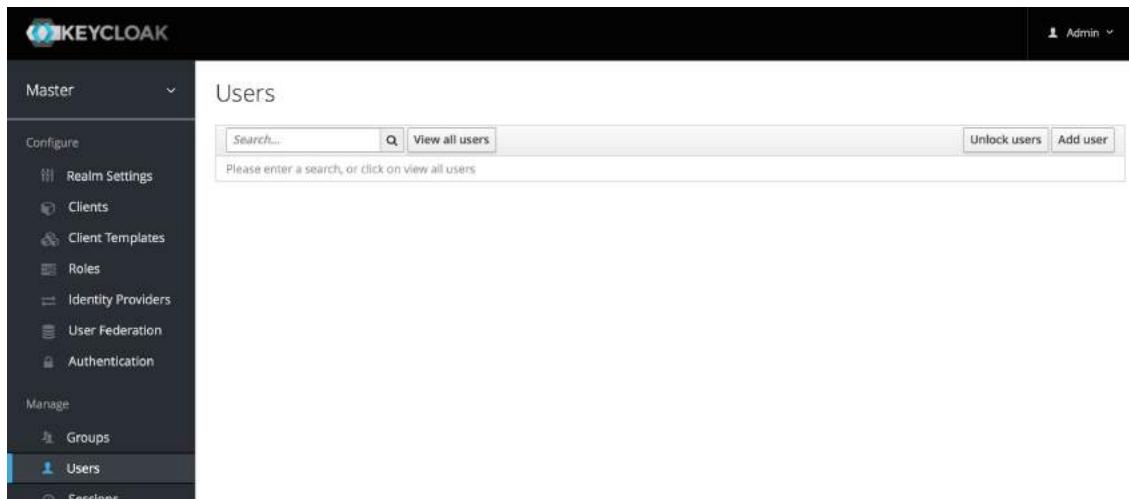
# 4. User Management

This section describes the administration functions for managing users.

## 4.1. Searching For Users

If you need to manage a specific user, click on `Users` in the left menu bar.

### *Users*



This menu option brings you to the user list page. In the search box you can type in a full name, last name, or email address you want to search for in the user database. The query will bring up all users that match your criteria. The `View all users` button will list every user in the system. This will search just local KeyCloak database and not the federated database (ie. LDAP) because some backends like LDAP don't have a way to page through users. So if you want the users from federated backend to be synced into KeyCloak database you need to either:

- Adjust search criteria. That will sync just the backend users

matching the criteria into KeyCloak database.

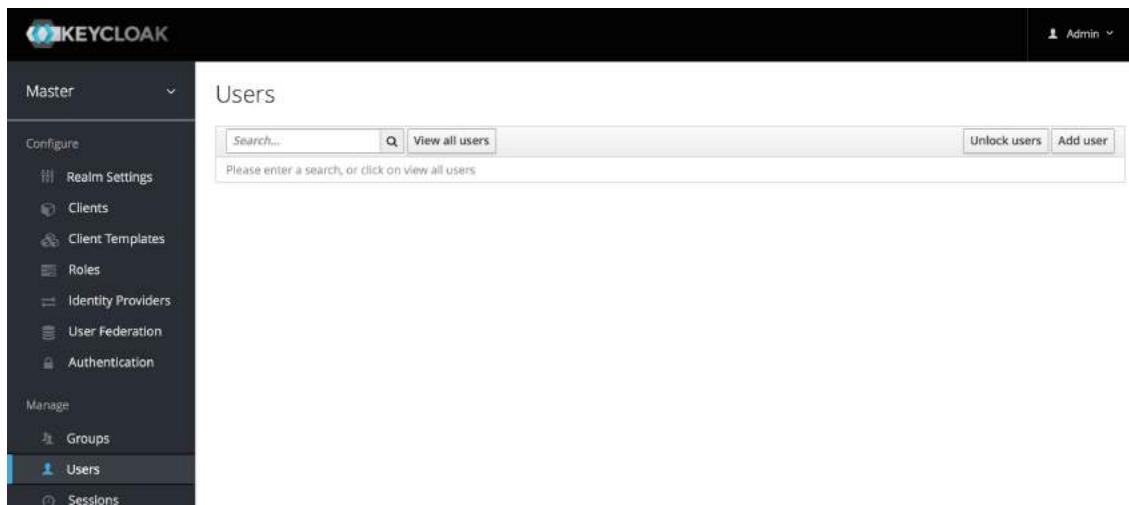
- Go to **User Federation** tab and click **Sync all users** or **Sync changed users** in the page with your federation provider.

See [User Federation](#) for more details.

## 4.2. Creating New Users

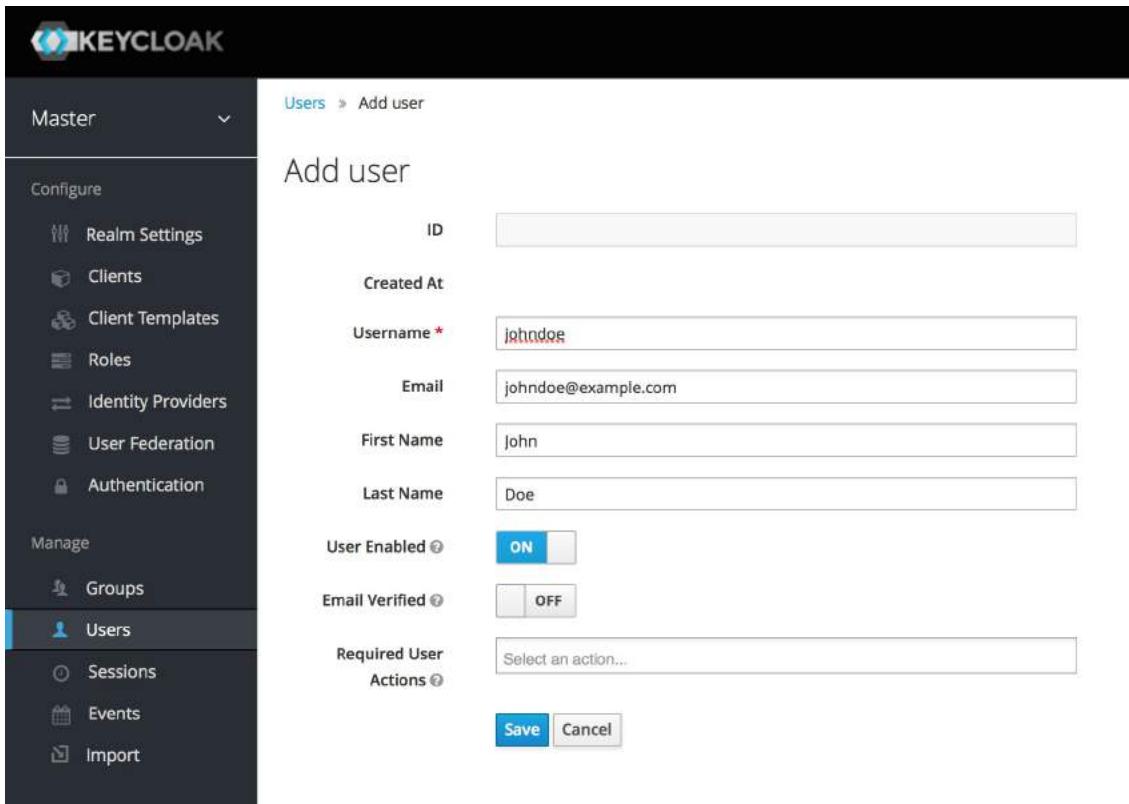
To create a user click on **Users** in the left menu bar.

### *Users*



This menu option brings you to the user list page. On the right side of the empty user list, you should see an **Add User** button. Click that to start creating your new user.

### *Add User*



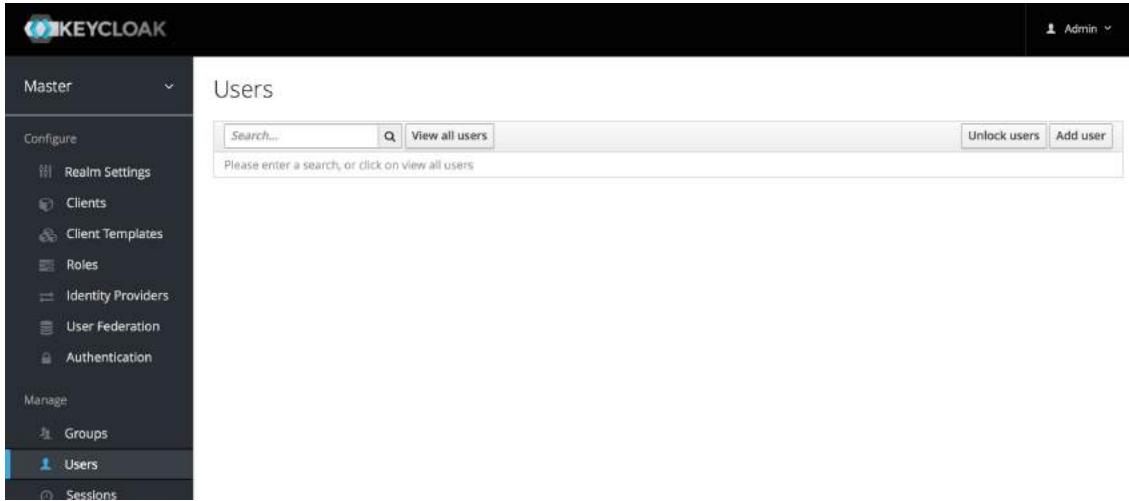
The screenshot shows the Keycloak 'Add user' form. The left sidebar is titled 'Master' and includes sections for 'Configure' (Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication) and 'Manage' (Groups, Users, Sessions, Events, Import). The 'Users' option under 'Manage' is highlighted with a blue bar. The main form has a title 'Add user'. It contains fields for 'ID' (empty), 'Created At' (empty), 'Username \*' (containing 'johndoe'), 'Email' (containing 'johndoe@example.com'), 'First Name' (containing 'John'), 'Last Name' (containing 'Doe'), 'User Enabled' (set to 'ON'), 'Email Verified' (set to 'OFF'), and a 'Required User Actions' dropdown menu ('Select an action...'). At the bottom are 'Save' and 'Cancel' buttons.

The only required field is `Username`. Click save. This will bring you to the management page for your new user.

## 4.3. Deleting Users

To delete a user click on `Users` in the left menu bar.

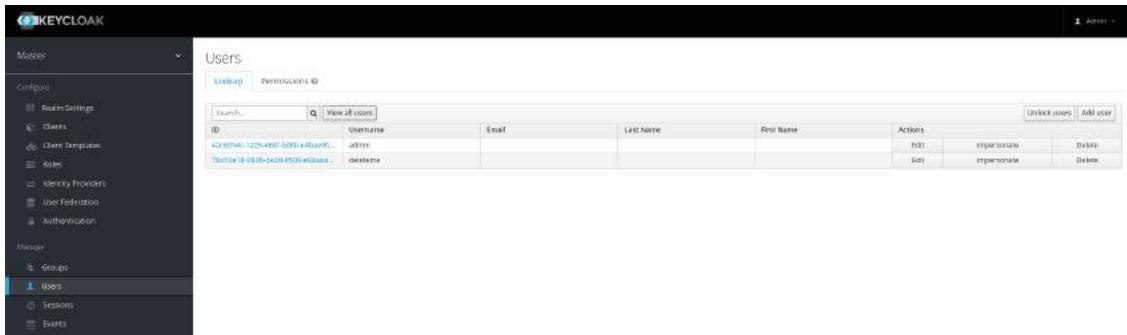
### Users



The screenshot shows the Keycloak 'Users' management page. The left sidebar is identical to the previous 'Add user' screen. The main area is titled 'Users' and features a search bar ('Search...', 'View all users'), an 'Unlock users' button, and an 'Add user' button. Below the search bar is a placeholder text 'Please enter a search, or click on view all users'. The right side of the page is currently empty, showing a list of users.

This menu option brings you to the user list page. Click `View all users` or search to find the user you intend to delete.

## Add User



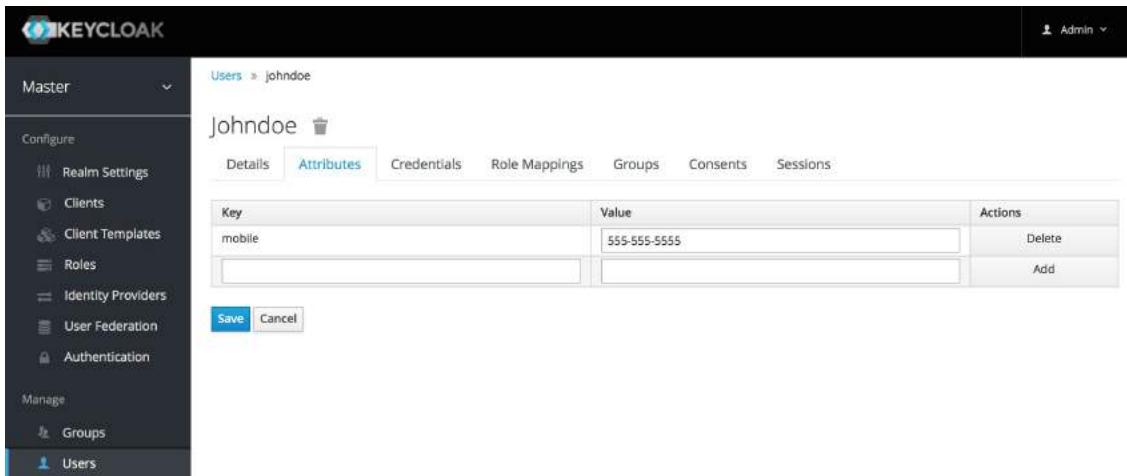
The screenshot shows the Keycloak Admin UI with the 'Users' tab selected in the left sidebar. The main area displays a table of users with columns: ID, Username, Email, First Name, Last Name, Actions, First Name, Last Name, and Actions. There are two users listed: 'admin' and 'johndoe'. The 'Actions' column for 'johndoe' contains three buttons: 'Edit', 'Impersonate', and 'Delete'.

In the list of users, click `Delete` next to the user you want to remove. You will be asked to confirm that you are sure you want to delete this user. Click `Delete` in the confirmation box to confirm.

## 4.4. User Attributes

Beyond basic user metadata like name and email, you can store arbitrary user attributes. Choose a user to manage then click on the `Attributes` tab.

### Users



The screenshot shows the Keycloak Admin UI with the 'Users' tab selected in the left sidebar. A specific user, 'johndoe', is selected. The 'Attributes' tab is active in the top navigation bar. Below it, there is a table with columns: Key, Value, and Actions. One attribute is listed: 'mobile' with value '555-555-5555'. There are 'Delete' and 'Add' buttons in the 'Actions' column. At the bottom of the table are 'Save' and 'Cancel' buttons.

Enter in the attribute name and value in the empty fields and click the

Add button next to it to add a new field. Note that any edits you make on this page will not be stored until you hit the Save button.

## 4.5. User Credentials

When viewing a user if you go to the `Credentials` tab you can manage a user's credentials.

### *Credential Management*

The screenshot shows the Keycloak interface for managing user credentials. On the left, a sidebar menu includes 'Master', 'Configure' (with sub-options like Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication), and 'Manage' (with sub-options like Groups, Users, Sessions, Events, Import). The main area shows a user named 'johndoe'. The 'Credentials' tab is selected. In the 'Manage Password' section, there are fields for 'New Password' (containing '\*\*\*\*\*'), 'Password Confirmation' (also containing '\*\*\*\*\*'), and a 'Temporary' switch which is set to 'ON'. A red 'Reset Password' button is present. Below this, the 'Credential Reset' section has two buttons: 'Update Password' and 'Send email'.

### 4.5.1. Changing Passwords

To change a user's password, type in a new one. A `Reset Password` button will show up that you click after you've typed everything in. If the `Temporary` switch is on, this new password can only be used once and the user will be asked to change their password after they have logged in.

Alternatively, if you have [email](#) set up, you can send an email to the user that asks them to reset their password. Choose [Update Password](#) from the [Reset Actions](#) list box and click [Send Email](#). You can optionally set the validity of the e-mail link which defaults to the one preset in [Tokens](#) tab in the realm settings. The sent email contains a link that will bring the user to the update password screen.

#### 4.5.2. Changing OTPs

You cannot configure One-Time Passwords for a specific user within the Admin Console. This is the responsibility of the user. If the user has lost their OTP generator all you can do is disable OTP for them on the [Credentials](#) tab. If OTP is optional in your realm, the user will have to go to the User Account Management service to re-configure a new OTP generator. If OTP is required, then the user will be asked to re-configure a new OTP generator when they log in.

Like passwords, you can alternatively send an email to the user that will ask them to reset their OTP generator. Choose [Configure OTP](#) in the [Reset Actions](#) list box and click the [Send Email](#) button. The sent email contains a link that will bring the user to the OTP setup screen.

## 4.6. Required Actions

Required Actions are tasks that a user must finish before they are allowed to log in. A user must provide their credentials before required actions are executed. Once a required action is completed, the user will not have to perform the action again. Here are explanations of some of the built-in required action types:

### Update Password

When set, a user must change their password.

## Configure OTP

When set, a user must configure a one-time password generator on their mobile device using either the Free OTP or Google Authenticator application.

## Verify Email

When set, a user must verify that they have a valid email account. An email will be sent to the user with a link they have to click. Once this workflow is successfully completed, they will be allowed to log in.

## Update Profile

This required action asks the user to update their profile information, i.e. their name, address, email, and/or phone number.

Admins can add required actions for each individual user within the user's `Details` tab in the Admin Console.

### *Setting Required Action*

The screenshot shows the Keycloak administration interface. On the left, there's a sidebar with 'Master' selected. Under 'Configure', it lists 'Realm Settings', 'Clients', 'Client Templates', 'Roles', 'Identity Providers', 'User Federation', and 'Authentication'. Under 'Manage', it lists 'Groups', 'Users' (which is highlighted in blue), 'Sessions', 'Events', and 'Import'. The main content area shows a user named 'johndoe'. The 'Details' tab is active. The user's ID is '15d229e6-a532-4d09-92b6-26567e1276e6'. It was created on '1/17/17 5:17:35 PM'. The username is 'johndoe' and the email is 'johndoe@example.com'. The first name is 'John' and the last name is 'Doe'. The 'User Enabled' switch is set to 'ON'. The 'Email Verified' switch is set to 'OFF'. There's a 'Required User Actions' section with a button labeled 'Update Password'. Below that is an 'Impersonate user' section with a 'Impersonate' button. At the bottom are 'Save' and 'Cancel' buttons.

In the `Required User Actions` list box, select all the actions you want to add to the account. If you want to remove one, click the `X` next to the action name. Also remember to click the `Save` button after you've decided what actions to add.

#### 4.6.1. Default Required Actions

You can also specify required actions that will be added to an account whenever a new user is created, i.e. through the `Add User` button the user list screen, or via the [user registration](#) link on the login page. To specify the default required actions go to the `Authentication` left menu item and click on the `Required Actions` tab.

#### *Default Required Actions*

Required Action	Enabled	Default Action
Configure OTP	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Terms and Conditions	<input type="checkbox"/>	<input type="checkbox"/>
Update Password	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Update Profile	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Verify Email	<input checked="" type="checkbox"/>	<input type="checkbox"/>

Simply click the checkbox in the `Default Action` column of the required actions that you want to be executed when a brand new user logs in.

#### 4.6.2. Terms and Conditions

Many organizations have a requirement that when a new user logs in for the first time, they need to agree to the terms and conditions of the website. KeyCloak has this functionality implemented as a required action, but it requires some configuration. For one, you have to go to the `Required Actions` tab described earlier and enable the `Terms and Conditions` action. You must also edit the `terms.ftl` file in the `base` login theme. See the [{developerguide!}](#) for more information on extending and creating themes.

## 4.7. Impersonation

It is often useful for an admin to impersonate a user. For example, a user may be experiencing a bug in one of your applications and an admin may want to impersonate the user to see if they can duplicate the problem. Admins with the appropriate permission can impersonate a user. There are two locations an admin can initiate impersonation. The first is

on the `Users` list tab.

## *Users*

ID	Username	Email	Last Name	First Name	Actions
15d229e6-a532-4d...	johndoe	johndoe@example....	Doe	John	Edit Impersonate Delete

You can see here that the admin has searched for `john`. Next to John's account you can see an impersonate button. Click that to impersonate the user.

Also, you can impersonate the user from the user `Details` tab.

## *User Details*

The screenshot shows the Keycloak Admin Console interface. On the left, a sidebar menu is visible with sections like 'Configure' (Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication) and 'Manage' (Groups, Users, Sessions, Events, Import). The 'Users' section is currently selected. The main content area shows a user named 'johndoe'. The 'Details' tab is active, showing the following user information:

Field	Value
ID	15d229e6-a532-4d09-92b6-26567e1276e6
Created At	1/17/17 5:17:35 PM
Username	johndoe
Email	johndoe@example.com
First Name	John
Last Name	Doe
User Enabled	ON
Email Verified	OFF
Required User	Select an action...
Actions	(button)
Impersonate user	Impersonate

At the bottom of the form, there are 'Save' and 'Cancel' buttons.

Near the bottom of the page you can see the `Impersonate` button. Click that to impersonate the user.

When impersonating, if the admin and the user are in the same realm, then the admin will be logged out and automatically logged in as the user being impersonated. If the admin and user are not in the same realm, the admin will remain logged in, but additionally be logged in as the user in that user's realm. In both cases, the browser will be redirected to the impersonated user's User Account Management page.

Any user with the realm's `impersonation` role can impersonate a user. Please see the [Admin Console Access Control](#) chapter for more details on assigning administration permissions.

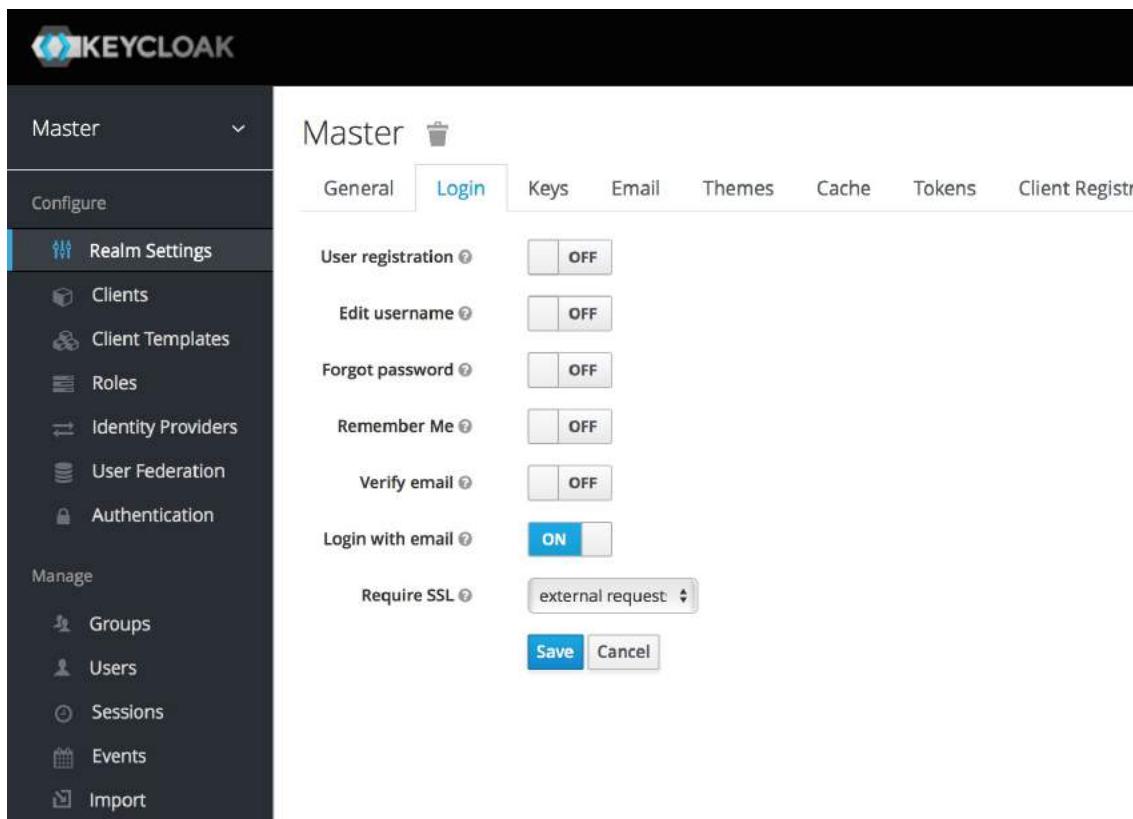
## 4.8. User Registration

You can enable Keycloak to allow user self registration. When enabled, the login page has a registration link the user can click on to create their new account.

When user self registration is enabled it is possible to use the registration form to detect valid usernames and emails. It is also possible to enable [reCAPTCHA Support](#).

Enabling registration is pretty simple. Go to the `Realm Settings` left menu and click it. Then go to the `Login` tab. There is a `User Registration` switch on this tab. Turn it on, then click the `Save` button.

### *Login Tab*



The screenshot shows the Keycloak Admin UI interface. The top navigation bar is black with the Keycloak logo and the word "KEYCLOAK". Below it, the main header "Master" is displayed. On the left, there is a dark sidebar with the following menu items:

- Configure
- Realm Settings** (highlighted with a blue border)
- Clients
- Client Templates
- Roles
- Identity Providers
- User Federation
- Authentication

Under "Manage", the following items are listed:

- Groups
- Users
- Sessions
- Events
- Import

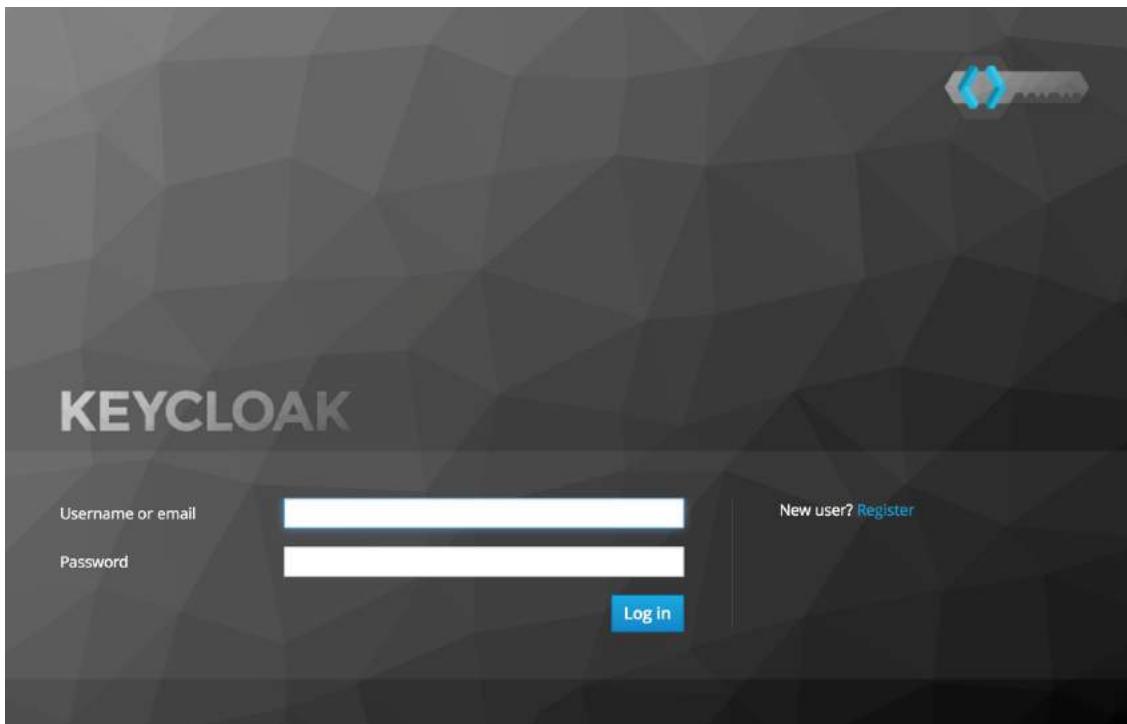
The main content area is titled "Master" and contains the "Login" tab, which is currently selected (indicated by a blue border). The "Login" tab has the following configuration options:

Setting	Status
User registration	OFF
Edit username	OFF
Forgot password	OFF
Remember Me	OFF
Verify email	OFF
Login with email	ON
Require SSL	external request

At the bottom right of the configuration area are two buttons: "Save" (in blue) and "Cancel".

After you enable this setting, a `Register` link should show up on the login page.

## *Registration Link*



Clicking on this link will bring the user to the registration page where they have to enter in some user profile information and a new password.

## *Registration Form*



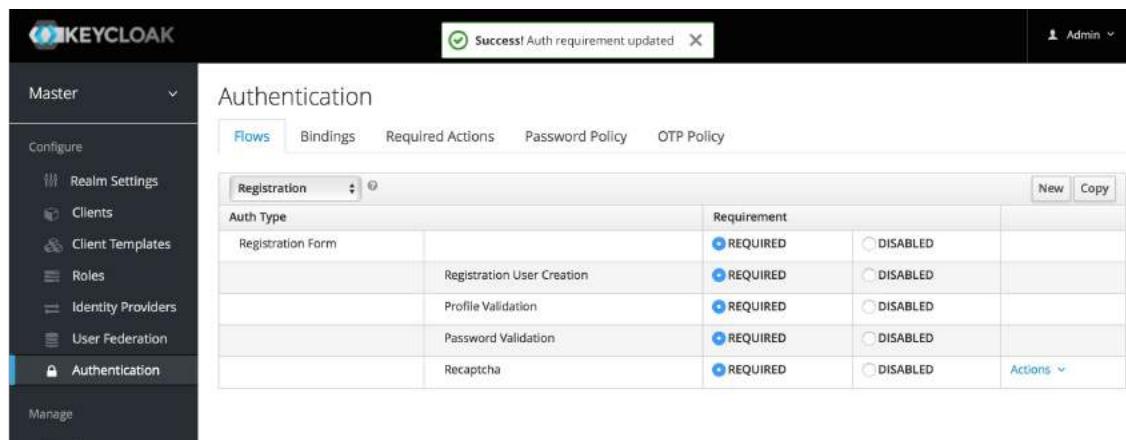
You can change the look and feel of the registration form as well as removing or adding additional fields that must be entered. See the [{developerguide!}](#) for more information.

#### 4.8.1. reCAPTCHA Support

To safeguard registration against bots, Keycloak has integration with Google reCAPTCHA. To enable this you need to first go to [Google Re-captcha Website](#) and create an API key so that you can get your reCAPTCHA site key and secret. (FYI, localhost works by default so you don't have to specify a domain).

Next, there are a few steps you need to perform in the Keycloak Admin Console. Click the `Authentication` left menu item and go to the `Flows` tab. Select the `Registration` flow from the drop down list on this page.

#### *Registration Flow*



The screenshot shows the Keycloak Admin Console interface. The left sidebar is titled "Master" and contains the following navigation items: Configure (with sub-items: Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication), and Manage. The "Authentication" item under "Configure" is currently selected. The main content area is titled "Authentication" and shows the "Flows" tab selected. A sub-tab "Registration" is active. Below this, there is a table with columns "Auth Type", "Requirement", and "Actions". The table rows are:

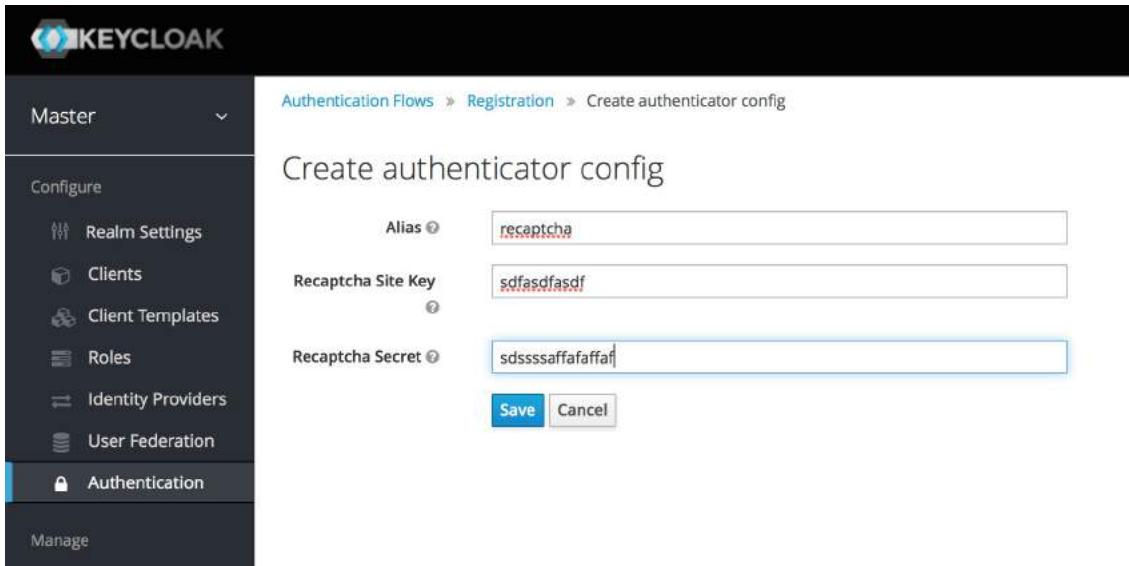
Auth Type	Requirement	Actions
Registration Form	<input checked="" type="radio"/> REQUIRED <input type="radio"/> DISABLED	
Registration User Creation	<input checked="" type="radio"/> REQUIRED <input type="radio"/> DISABLED	
Profile Validation	<input checked="" type="radio"/> REQUIRED <input type="radio"/> DISABLED	
Password Validation	<input checked="" type="radio"/> REQUIRED <input type="radio"/> DISABLED	
Recaptcha	<input checked="" type="radio"/> REQUIRED <input type="radio"/> DISABLED	Actions

A success message "Success! Auth requirement updated" is displayed at the top right of the main content area. The top right corner also shows the user "Admin".

Set the 'reCAPTCHA' requirement to `Required` by clicking the appropriate radio button. This will enable reCAPTCHA on the screen. Next, you have to enter in the reCAPTCHA site key and secret that you generated at the Google reCAPTCHA Website. Click on the 'Actions' button that is to the right of the reCAPTCHA flow entry, then "Config" link,

and enter in the reCAPTCHA site key and secret on this config page.

### *Recaptcha Config Page*



The screenshot shows the Keycloak administration interface. The left sidebar is titled 'Master' and contains the following menu items under 'Configure': Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication (which is highlighted in blue), and Manage. The main content area is titled 'Create authenticator config'. It has three input fields: 'Alias' with the value 'recaptcha', 'ReCaptcha Site Key' with the value 'sdfasdfsadf', and 'ReCaptcha Secret' with the value 'sdssssaffafaffaf'. Below these fields are two buttons: 'Save' (in blue) and 'Cancel'.

The final step you have to do is to change some default HTTP response headers that KeyCloak sets. KeyCloak will prevent a website from including any login page within an iframe. This is to prevent clickjacking attacks. You need to authorize Google to use the registration page within an iframe. Go to the `Realm Settings` left menu item and then go to the `Security Defenses` tab. You will need to add `https://www.google.com` to the values of both the `X-Frame-Options` and `Content-Security-Policy` headers.

### *Authorizing Iframes*

The screenshot shows the Keycloak administration interface. On the left, there's a sidebar with a 'Master' dropdown, a 'Configure' section, and a 'Realm Settings' section which is currently active, indicated by a blue background. Under 'Realm Settings', there are links for Clients, Client Templates, Roles, Identity Providers, User Federation, and Authentication. Below these are 'Manage' sections for Groups and Users. At the top right, there are tabs for General, Login, Keys, Email, Themes, Cache, Tokens, Client Registration, and Security Defenses, with 'Security Defenses' being the active tab. The main content area is titled 'Master' and shows the 'Brute Force Detection' configuration. It has three input fields: 'X-Frame-Options' containing 'ALLOW-FROM https://www.google.com', 'Content-Security-Policy' containing 'frame-src \'self\' https://www.google.com', and 'X-Content-Type-Options' containing 'nosniff'. At the bottom are 'Save' and 'Cancel' buttons.

Once you do this, reCAPTCHA should show up on your registration page. You may want to edit `register.ftl` in your login theme to muck around with the placement and styling of the reCAPTCHA button. See the [{developerguide!}](#) for more information on extending and creating themes.

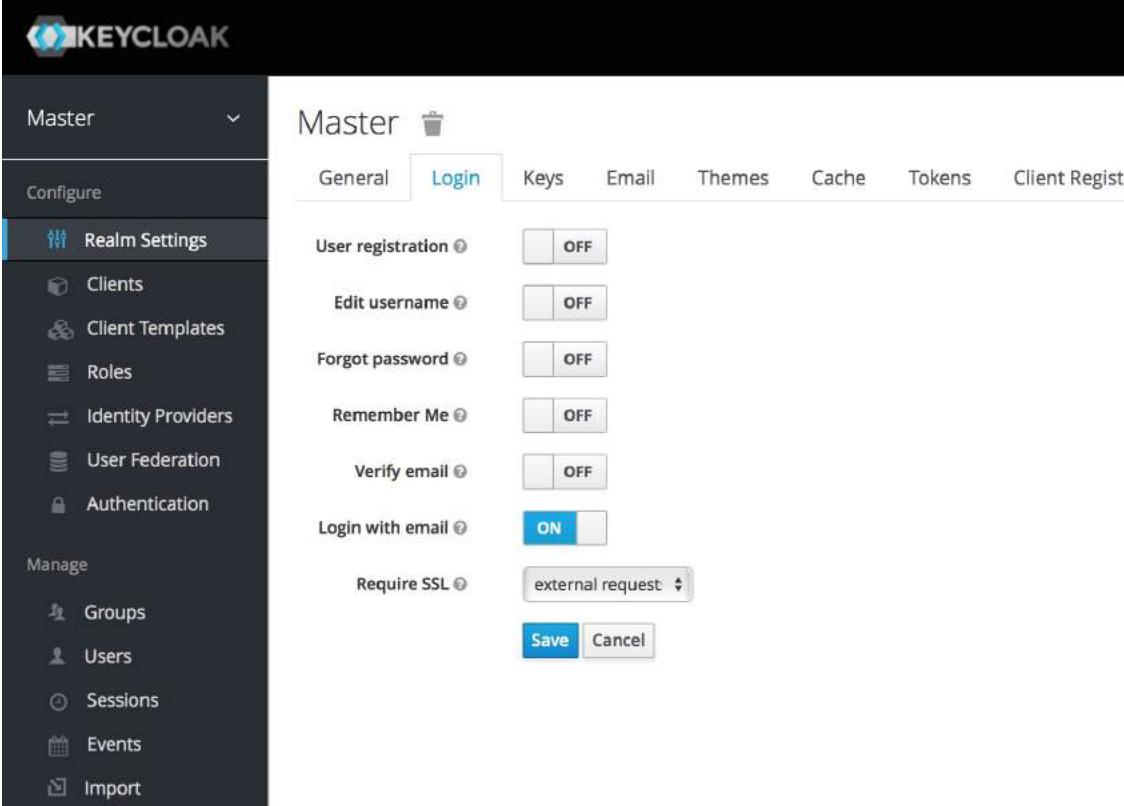
# 5. Login Page Settings

There are several nice built-in login page features you can enable if you need the functionality.

## 5.1. Forgot Password

If you enable it, users are able to reset their credentials if they forget their password or lose their OTP generator. Go to the `Realm Settings` left menu item, and click on the `Login` tab. Switch on the `Forgot Password` switch.

### *Login Tab*

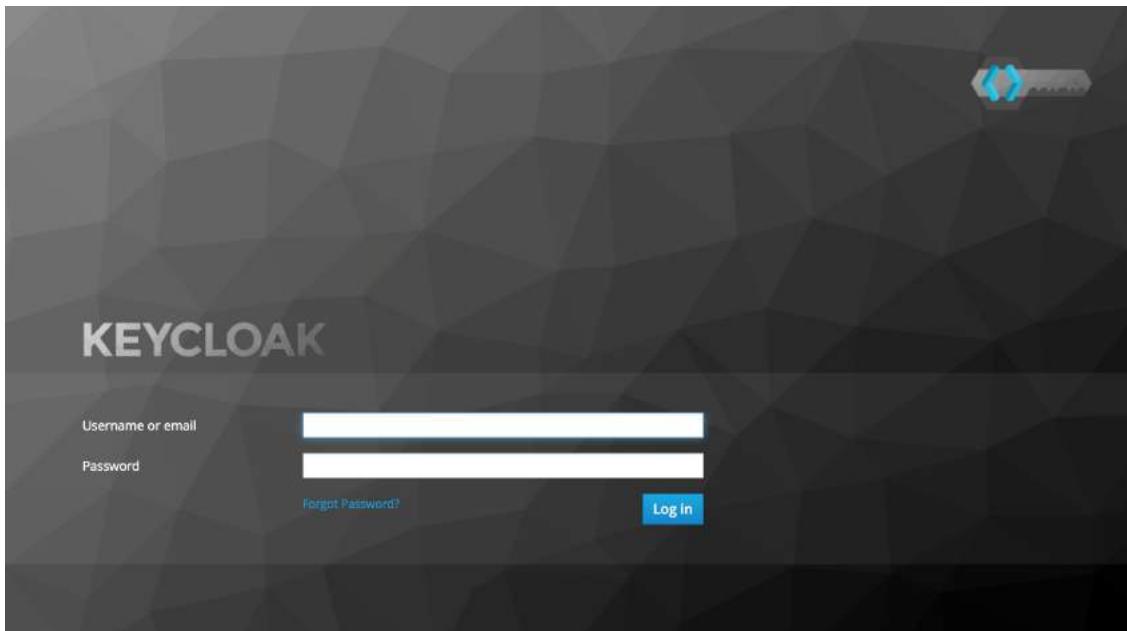


The screenshot shows the Keycloak Admin UI with the following details:

- Left Sidebar:** Shows the navigation tree under "Master". The "Realm Settings" item is currently selected.
- Top Bar:** Shows the Keycloak logo and the realm name "Master".
- Tab Bar:** Shows tabs for General, Login, Keys, Email, Themes, Cache, Tokens, and Client Registrations. The "Login" tab is active.
- Configuration Options:**
  - User registration: OFF
  - Edit username: OFF
  - Forgot password: OFF (This is the switch being enabled)
  - Remember Me: OFF
  - Verify email: OFF
  - Login with email: ON (This is the switch being enabled)
  - Require SSL: external request
- Buttons:** Save and Cancel buttons at the bottom right.

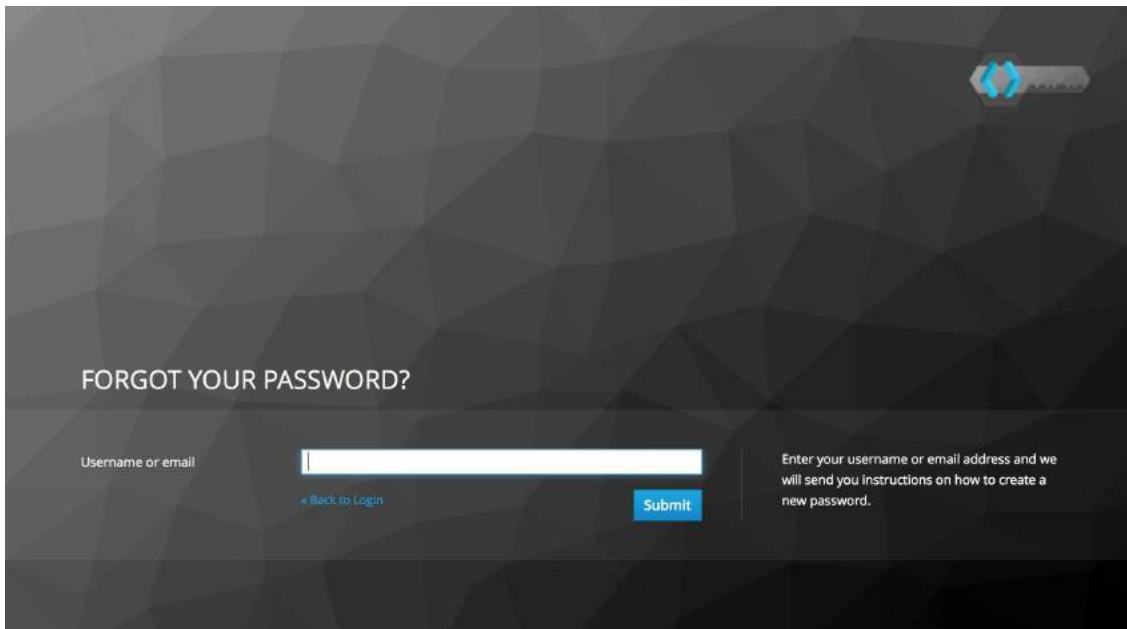
A `forgot password` link will now show up on your login pages.

## *Forgot Password Link*



Clicking on this link will bring the user to a page where they can enter in their username or email and receive an email with a link to reset their credentials.

## *Forgot Password Page*

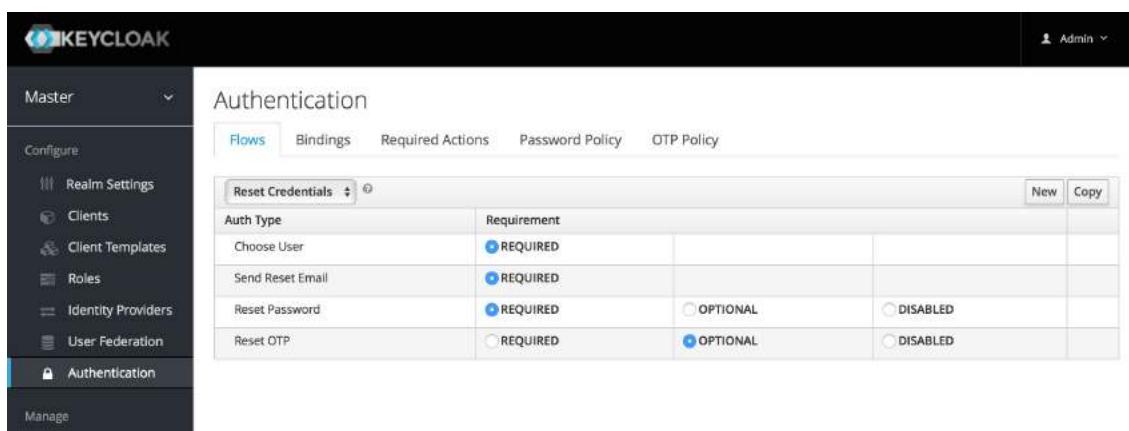


The text sent in the email is completely configurable. You just need to extend or edit the theme associated with it. See the [{developerguide!}](#)

for more information.

When the user clicks on the email link, they will be asked to update their password, and, if they have an OTP generator set up, they will also be asked to reconfigure this as well. Depending on the security requirements of your organization you may not want users to be able to reset their OTP generator through email. You can change this behavior by going to the `Authentication` left menu item, clicking on the `Flows` tab, and selecting the `Reset Credentials` flow:

### *Reset Credentials Flow*



The screenshot shows the Keycloak Admin UI with the following details:

- Header:** KEYCLOAK with a user icon and "Admin" dropdown.
- Left Sidebar:** Master dropdown, Configure section (Clients, Client Templates, Roles, Identity Providers, User Federation), Authentication (selected), and Manage.
- Top Bar:** Authentication tab, Flows (selected), Bindings, Required Actions, Password Policy, OTP Policy.
- Flow Configuration:** A table titled "Reset Credentials".

Auth Type	Requirement		
Choose User	<input checked="" type="radio"/> REQUIRED		
Send Reset Email	<input checked="" type="radio"/> REQUIRED		
Reset Password	<input checked="" type="radio"/> REQUIRED	<input type="radio"/> OPTIONAL	<input type="radio"/> DISABLED
Reset OTP	<input type="radio"/> REQUIRED	<input checked="" type="radio"/> OPTIONAL	<input type="radio"/> DISABLED

If you do not want OTP reset, then just chose the `disabled` radio button to the right of `Reset OTP`.

## 5.2. Remember Me

If a logged in user closes their browser, their session is destroyed and they will have to log in again. You can set things up so that if a user checks a *remember me* checkbox, they will remain logged in even if the browser is closed. This basically turns the login cookie from a session-only cookie to a persistence cookie.

To enable this feature go to `Realm Settings` left menu item and click

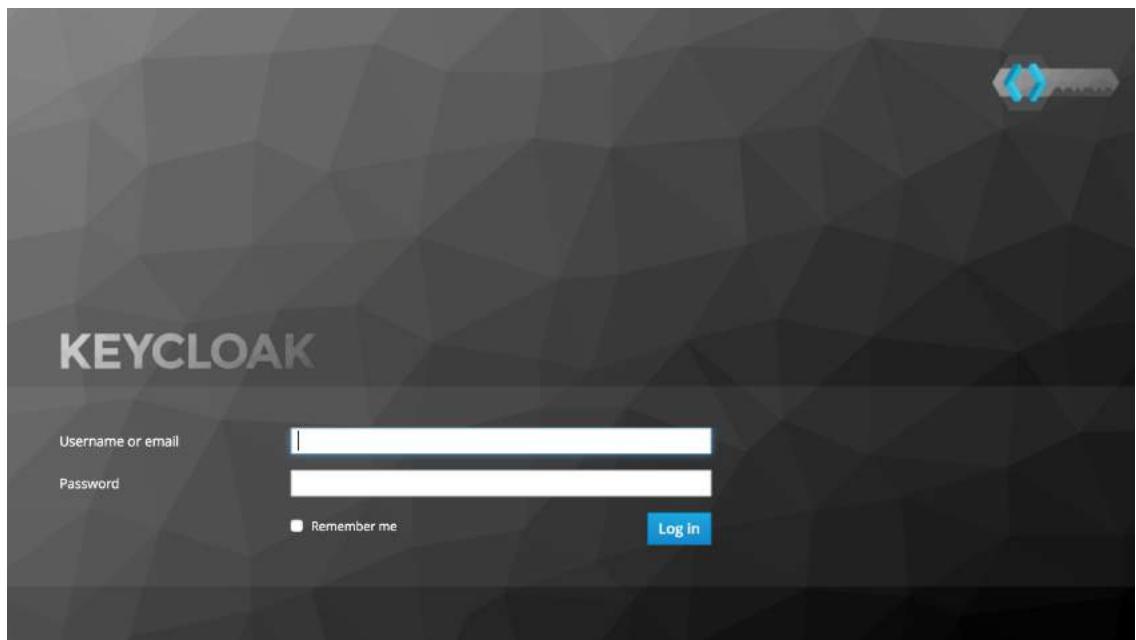
on the `Login` tab and turn on the `Remember Me` switch:

### *Login Tab*

The screenshot shows the Keycloak Admin UI with the 'Master' realm selected. The left sidebar has 'Realm Settings' highlighted. The top navigation bar includes tabs for General, Login (which is active), Keys, Email, Themes, Cache, Tokens, and Client Registr. The 'Login' tab page contains several configuration options with their current status: User registration (OFF), Edit username (OFF), Forgot password (OFF), Remember Me (OFF), Verify email (OFF), and Login with email (ON). Below these is a dropdown for 'Require SSL' set to 'external request'. At the bottom are 'Save' and 'Cancel' buttons.

Once you save this setting, a `remember me` checkbox will be displayed on the realm's login page.

### *Remember Me*



---

## 6. Authentication

There are a few features you should be aware of when configuring authentication for your realm. Many organizations have strict password and OTP policies that you can enforce via settings in the Admin Console. You may or may not want to require different credential types for authentication. You may want to give users the option to login via Kerberos or disable or enable various built-in credential types. This chapter covers all of these topics.

### 6.1. Password Policies

Each new realm created has no password policies associated with it. Users can have as short, as long, as complex, as insecure a password, as they want. Simple settings are fine for development or learning KeyCloak, but unacceptable in production environments. KeyCloak has a rich set of password policies you can enable through the Admin Console.

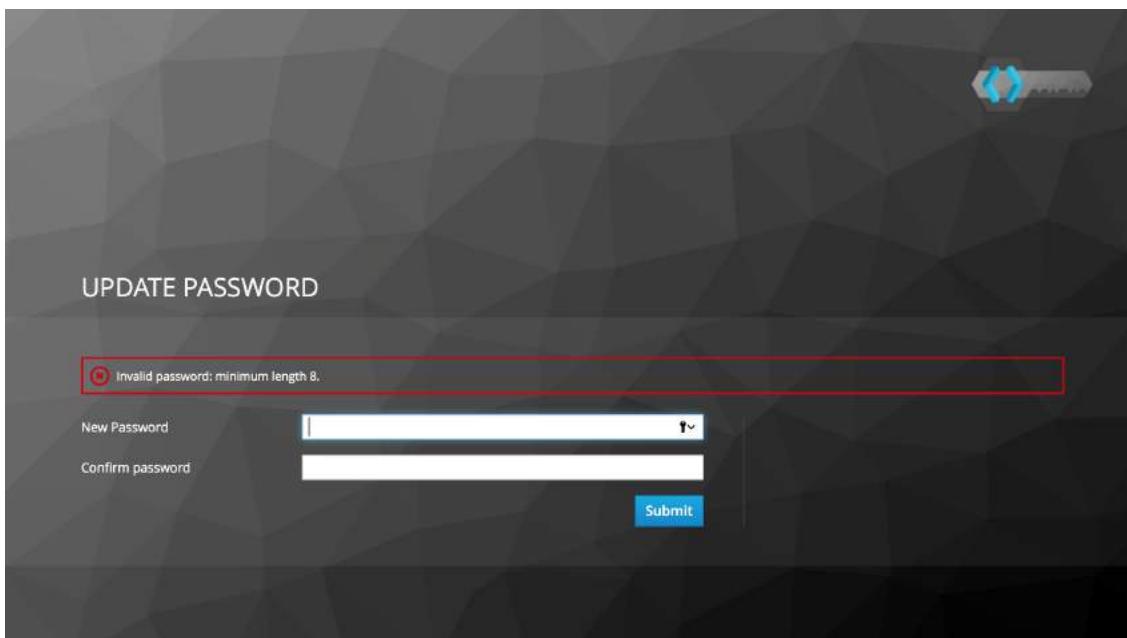
Click on the `Authentication` left menu item and go to the `Password Policy` tab. Choose the policy you want to add in the right side drop down list box. This will add the policy in the table on the screen. Choose the parameters for the policy. Hit the `Save` button to store your changes.

#### *Password Policy*

The screenshot shows the Keycloak Admin UI with the navigation bar at the top. The left sidebar has a 'Master' dropdown, followed by sections for 'Configure' (Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation) and 'Authentication' (which is selected). Below the sidebar is the main content area titled 'Authentication'. Under 'Authentication', tabs include 'Flows', 'Bindings', 'Required Actions', 'Password Policy' (which is selected), and 'OTP Policy'. The 'Password Policy' tab displays a table with two rows: 'Hashing Iterations' set to '20000' and 'Minimum Length' set to '8'. A 'Save' button is at the bottom left, and a 'Delete' button is at the bottom right. A 'Add policy...' dropdown menu is visible at the top right of the table.

After saving your policy, user registration and the Update Password required action will enforce your new policy. An example of a user failing the policy check:

### *Failed Password Policy*



If the password policy is updated, an Update Password action must be set for every user. An automatic trigger is scheduled as a future enhancement.

#### 6.1.1. Password Policy Types

Here's an explanation of each policy type:

## **Hashing Iterations**

This value specifies the number of times a password will be hashed before it is stored or verified. The default value is 20,000. This hashing is done in the rare case that a hacker gets access to your password database. Once they have access to the database, they can reverse engineer user passwords. The industry recommended value for this parameter changes every year as CPU power improves. A higher hashing iteration value takes more CPU power for hashing, and can impact performance. You'll have to weigh what is more important to you: performance or protecting your passwords stores. There may be more cost effective ways of protecting your password stores.

## **Digits**

The number of digits required to be in the password string.

## **Lowercase Characters**

The number of lower case letters required to be in the password string.

## **Uppercase Characters**

The number of upper case letters required to be in the password string.

## **Special Characters**

The number of special characters like '?!#%\$' required to be in the password string.

## **Not Username**

When set, the password is not allowed to be the same as the username.

me.

## Regular Expression

Define one or more Perl regular expression patterns that passwords must match.

## Expire Password

The number of days for which the password is valid. After the number of days has expired, the user is required to change their password.

## Not Recently Used

This policy saves a history of previous passwords. The number of old passwords stored is configurable. When a user changes their password they cannot use any stored passwords.

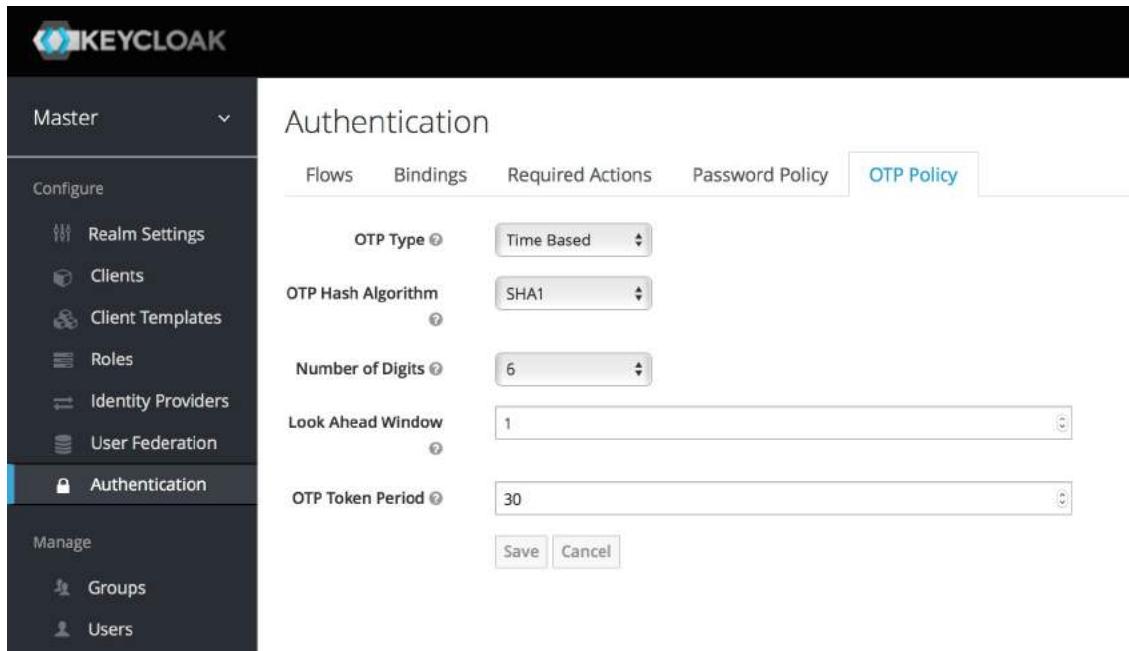
## Password Blacklist

This policy checks if a given password is contained in a blacklist file, which is potentially a very large file. Password blacklists are UTF-8 plain-text files with Unix line endings where every line represents a blacklisted password. The file name of the blacklist file must be provided as the password policy value, e.g. `10_million_password-list_top_1000000.txt`. Blacklist files are resolved against `#{jboss.server.data.dir}/password-blacklists/` by default. This path can be customized via the `keycloak.password.blacklists.path` system property, or the `blacklistsPath` property of the `passwordBlacklist` policy SPI configuration.

## 6.2. OTP Policies

KeyCloak has a number of policies you can set up for your FreeOTP or Google Authenticator One-Time Password generator. Click on the `Authentication` left menu item and go to the `OTP Policy` tab.

### *OTP Policy*



The screenshot shows the Keycloak administrative interface. The top navigation bar has the Keycloak logo and the word "KEYCLOAK". Below it, the left sidebar has a "Master" dropdown and several menu items under "Configure": "Realm Settings", "Clients", "Client Templates", "Roles", "Identity Providers", "User Federation", "Authentication" (which is highlighted in blue), "Manage", "Groups", and "Users". The main content area is titled "Authentication" and contains tabs: "Flows", "Bindings", "Required Actions", "Password Policy", and "OTP Policy" (which is highlighted in blue). Under these tabs, there are configuration fields: "OTP Type" (set to "Time Based"), "OTP Hash Algorithm" (set to "SHA1"), "Number of Digits" (set to "6"), "Look Ahead Window" (set to "1"), and "OTP Token Period" (set to "30"). At the bottom right are "Save" and "Cancel" buttons.

Any policies you set here will be used to validate one-time passwords. When configuring OTP, FreeOTP and Google Authenticator can scan a QR code that is generated on the OTP set up page that KeyCloak has. The bar code is also generated from information configured on the `OTP Policy` tab.

#### **6.2.1. TOTP vs. HOTP**

There are two different algorithms to choose from for your OTP generators. Time Based (TOTP) and Counter Based (HOTP). For TOTP, your token generator will hash the current time and a shared secret. The server validates the OTP by comparing all the hashes within a certain window of time to the submitted value. So, TOTPs are valid only for a short window of time (usually 30 seconds). For HOTP a shared counter

is used instead of the current time. The server increments the counter with each successful OTP login. So, valid OTPs only change after a successful login.

TOTP is considered a little more secure because the matchable OTP is only valid for a short window of time while the OTP for HOTP can be valid for an indeterminate amount of time. HOTP is much more user friendly as the user won't have to hurry to enter in their OTP before the time interval is up. With the way KeyCloak has implemented TOTP this distinction becomes a little more blurry. HOTP requires a database update every time the server wants to increment the counter. This can be a performance drain on the authentication server when there is heavy load. So, to provide a more efficient alternative, TOTP does not remember passwords used. This bypasses the need to do any DB updates, but the downside is that TOTPs can be re-used in the valid time interval. For future versions of KeyCloak it is planned that you will be able to configure whether TOTP checks older OTPs in the time interval.

### 6.2.2. TOTP Configuration Options

#### OTP Hash Algorithm

Default is SHA1, more secure options are SHA256 and SHA512.

#### Number of Digits

How many characters is the OTP? Short means more user friendly as it is less the user has to type. More means more security.

#### Look Ahead Window

How many intervals ahead should the server try and match the hash? This exists so just in case the clock of the TOTP generator or authen-

tication server get out of sync. The default value of 1 is usually good enough. For example, if the time interval for a new token is every 30 seconds, the default value of 1 means that it will only accept valid tokens in that 30 second window. Each increment of this config value will increase the valid window by 30 seconds.

## **OTP Token Period**

Time interval in seconds during which the server will match a hash. Each time the interval passes, a new TOTP will be generated by the token generator.

### **6.2.3. HOTP Configuration Options**

#### **OTP Hash Algorithm**

Default is SHA1, more secure options are SHA256 and SHA512.

#### **Number of Digits**

How many characters is the OTP? Short means more user friendly as it is less the user has to type. More means more security.

#### **Look Ahead Window**

How many counters ahead should the server try and match the hash? The default value is 1. This exists to cover the case where the user's counter gets ahead of the server's. This can often happen as users often increment the counter manually too many times by accident. This value really should be increased to a value of 10 or so.

#### **Initial Counter**

What is the value of the initial counter?

## 6.3. Authentication Flows

An *authentication flow* is a container for all authentications, screens, and actions that must happen during login, registration, and other Keycloak workflows. If you go to the admin console `Authentication` left menu item and go to the `Flows` tab, you can view all the defined flows in the system and what actions and checks each flow requires. This section does a walk-through of the browser login flow. In the left drop-down list select `browser` to come to the screen shown below:

### Browser Flow

Auth Type	Requirement		
Cookie	<input checked="" type="radio"/> ALTERNATIVE <input type="radio"/> REQUIRED <input type="radio"/> DISABLED		
Kerberos	<input type="radio"/> ALTERNATIVE <input checked="" type="radio"/> REQUIRED <input checked="" type="radio"/> DISABLED		
Identity Provider Redirector	<input checked="" type="radio"/> ALTERNATIVE <input type="radio"/> REQUIRED <input type="radio"/> DISABLED		Actions
Forms	<input checked="" type="radio"/> ALTERNATIVE <input checked="" type="radio"/> REQUIRED <input type="radio"/> OPTIONAL	<input checked="" type="radio"/> REQUIRED <input type="radio"/> OPTIONAL	<input type="radio"/> DISABLED
Username Password Form			
OTP Form	<input type="radio"/> REQUIRED <input checked="" type="radio"/> OPTIONAL <input type="radio"/> DISABLED		

If you hover over the tooltip (the tiny question mark) to the right of the flow selection list, this will describe what the flow is and does.

The `Auth Type` column is the name of authentication or action that will be executed. If an authentication is indented this means it is in a sub-flow and may or may not be executed depending on the behavior of its parent. The `Requirement` column is a set of radio buttons which define whether or not the action will execute. Let's describe what each radio button means:

### Required

This authentication execution must execute successfully. If the user doesn't have that type of authentication mechanism configured and there is a required action associated with that authentication type, then a required action will be attached to that account. For example, if you switch `OTP Form` to `Required`, users that don't have an OTP generator configured will be asked to do so.

## **Optional**

If the user has the authentication type configured, it will be executed. Otherwise, it will be ignored.

## **Disabled**

If disabled, the authentication type is not executed.

## **Alternative**

This means that at least one alternative authentication type must execute successfully at that level of the flow.

This is better described in an example. Let's walk through the `browser` authentication flow.

1. The first authentication type is `Cookie`. When a user successfully logs in for the first time, a session cookie is set. If this cookie has already been set, then this authentication type is successful. Since the cookie provider returned success and each execution at this level of the flow is *alternative*, no other execution is executed and this results in a successful login.
2. Next the flow looks at the Kerberos execution. This authenticator is disabled by default and will be skipped.

3. The next execution is a subflow called Forms. Since this subflow is marked as *alternative* it will not be executed if the `Cookie` authentication type passed. This subflow contains additional authentication type that needs to be executed. The executions for this subflow are loaded and the same processing logic occurs
4. The first execution in the Forms subflow is the Username Password Form. This authentication type renders the username and password page. It is marked as *required* so the user must enter in a valid username and password.
5. The next execution is the OTP Form. This is marked as *optional*. If the user has OTP set up, then this authentication type must run and be successful. If the user doesn't have OTP set up, this authentication type is ignored.

## 6.4. Kerberos

KeyCloak supports login with a Kerberos ticket through the SPNEGO protocol. SPNEGO (Simple and Protected GSSAPI Negotiation Mechanism) is used to authenticate transparently through the web browser after the user has been authenticated when logging-in his session. For non-web cases or when ticket is not available during login, KeyCloak also supports login with Kerberos username/password.

A typical use case for web authentication is the following:

1. User logs into his desktop (Such as a Windows machine in Active Directory domain or Linux machine with Kerberos integration enabled).
2. User then uses his browser (IE/Firefox/Chrome) to access a web

application secured by KeyCloak.

3. Application redirects to KeyCloak login.
4. KeyCloak renders HTML login screen together with status 401 and HTTP header `WWW-Authenticate: Negotiate`
5. In case that the browser has Kerberos ticket from desktop login, it transfers the desktop sign on information to the KeyCloak in header `Authorization: Negotiate 'spnego-token'`. Otherwise it just displays the login screen.
6. KeyCloak validates token from the browser and authenticates the user. It provisions user data from LDAP (in case of LDAPFederationProvider with Kerberos authentication support) or let user to update his profile and prefill data (in case of KerberosFederationProvider).
7. KeyCloak returns back to the application. Communication between KeyCloak and application happens through OpenID Connect or SAML messages. The fact that KeyCloak was authenticated through Kerberos is hidden from the application. So KeyCloak acts as broker to Kerberos/SPNEGO login.

For setup there are 3 main parts:

1. Setup and configuration of Kerberos server (KDC)
2. Setup and configuration of KeyCloak server
3. Setup and configuration of client machines

#### 6.4.1. Setup of Kerberos server

This is platform dependent. Exact steps depend on your OS and the Kerberos vendor you're going to use. Consult Windows Active Directory, MIT Kerberos and your OS documentation for how exactly to setup and configure Kerberos server.

At least you will need to:

- Add some user principals to your Kerberos database. You can also integrate your Kerberos with LDAP, which means that user accounts will be provisioned from LDAP server.
- Add service principal for "HTTP" service. For example if your KeyCloak server will be running on `www.mydomain.org` you may need to add principal `HTTP/www.mydomain.org@MYDOMAIN.ORG` assuming that MYDOMAIN.ORG will be your Kerberos realm.

For example on MIT Kerberos you can run a "kadmin" session. If you are on the same machine where is MIT Kerberos, you can simply use the command:

```
sudo kadmin.local
```

Then add HTTP principal and export his key to a keytab file with the commands like:

```
addprinc -randkey HTTP/www.mydomain.org@MYDOMAIN.ORG  
ktadd -k /tmp/http.keytab HTTP/www.mydomain.org@MYDOMAIN.  
ORG
```

The Keytab file `/tmp/http.keytab` will need to be accessible on the host where KeyCloak server will be running.

#### 6.4.2. Setup and configuration of KeyCloak server

You need to install a kerberos client on your machine. This is also platform dependent. If you are on Fedora, Ubuntu or RHEL, you can install the package `freeipa-client`, which contains a Kerberos client and several other utilities. Configure the kerberos client (on Linux it's in file `/etc/krb5.conf`). You need to put your Kerberos realm and at least configure the HTTP domains your server will be running on. For the example realm `MYDOMAIN.ORG` you may configure the `domain_realm` section like this:

```
[domain_realm]
.mydomain.org = MYDOMAIN.ORG
mydomain.org = MYDOMAIN.ORG
```

Next you need to export the keytab file with the HTTP principal and make sure the file is accessible to the process under which KeyCloak server is running. For production, it's ideal if it's readable just by this process and not by someone else. For the MIT Kerberos example above, we already exported keytab to `/tmp/http.keytab`. If your KDC and KeyCloak are running on same host, you have that file already available.

#### Enable SPNEGO Processing

KeyCloak does not have the SPNEGO protocol support turned on by default. So, you have to go to the [browser flow](#) and enable `Kerberos`.

#### *Browser Flow*

Switch the `Kerberos` requirement from *disabled* to either *alternative* or *required*. *Alternative* basically means that Kerberos is optional. If the user's browser hasn't been configured to work with SPNEGO/Kerberos, then KeyCloak will fall back to the regular login screens. If you set the requirement to *required* then all users must have Kerberos enabled for their browser.

## Configure Kerberos User Storage Federation Provider

Now that the SPNEGO protocol is turned on at the authentication server, you'll need to configure how KeyCloak interprets the Kerberos ticket. This is done through [User Storage Federation](#). We have 2 different federation providers with Kerberos authentication support.

If you want to authenticate with Kerberos backed by an LDAP server, you have to first configure the [LDAP Federation Provider](#). If you look at the configuration page for your LDAP provider you'll see a `Kerberos Integration` section.

### *LDAP Kerberos Integration*

**Connection Pooling**  ON

**Connection Timeout**  Connection Timeout

**Read Timeout**  Read Timeout

**Pagination**  ON

## Kerberos Integration

**Allow Kerberos authentication**  OFF

**Use Kerberos For Password Authentication**  OFF

## Sync Settings

**Batch Size**  1000

**Periodic Full Sync**  OFF

**Periodic Changed Users Sync**  OFF

## Cache Settings

**Cache Policy**  DEFAULT

**Save** **Cancel**

Turning on the switch `Allow Kerberos authentication` will make Keycloak use the Kerberos principal to lookup information about the user so that it can be imported into the Keycloak environment.

If your Kerberos solution is not backed by an LDAP server, you have to use the `Kerberos User Storage Federation Provider`. Go to the `User Federation` left menu item and select `Kerberos` from the `Add provider` select box.

## *Kerberos User Storage Provider*

This provider parses the Kerberos ticket for simple principal information and does a small import into the local Keycloak database. User profile information like first name, last name, and email are not provisioned.

#### 6.4.3. Setup and configuration of client machines

Clients need to install kerberos client and setup krb5.conf as described above. Additionally they need to enable SPNEGO login support in their browser. See [configuring Firefox for Kerberos](#) if you are using that browser. URI `.mydomain.org` must be allowed in the `network.negotiate-auth.trusted-uris` config option.

In a Windows domain, clients usually don't need to configure anything

special as IE is already able to participate in SPNEGO authentication for the Windows domain.

#### 6.4.4. Credential Delegation

Kerberos 5 supports the concept of credential delegation. In this scenario, your applications may want access to the Kerberos ticket so that they can re-use it to interact with other services secured by Kerberos. Since the SPNEGO protocol is processed in the KeyCloak server, you have to propagate the GSS credential to your application within the OpenID Connect token claim or a SAML assertion attribute that is transmitted to your application from the KeyCloak server. To have this claim inserted into the token or assertion, each application will need to enable the built-in protocol mapper called `gss delegation credential`. This is enabled in the `Mappers` tab of the application's client page. See [Protocol Mappers](#) chapter for more details.

Applications will need to deserialize the claim it receives from KeyCloak before it can use it to make GSS calls against other services. Once you deserialize the credential from the access token to the `GSSCredential` object, the `GSSContext` will need to be created with this credential passed to the method `GSSManager.createContext` for example like this:

```
// Obtain accessToken in your application.  
KeycloakPrincipal keycloakPrincipal = (KeycloakPrincipal)  
    servletReq.getUserPrincipal();  
AccessToken accessToken = keycloakPrincipal.getKeycloakSe-  
    curityContext().getToken();  
  
// Retrieve kerberos credential from accessToken and dese-  
// rialize it  
String serializedGssCredential = (String) accessToken.getO-
```

```

therClaims().
    get(org.keycloak.common.constants.KerberosConstants.
GSS_DELEGATION_CREDENTIAL);

GSSCredential serializedGssCredential = org.keycloak.
common.util.KerberosSerializationUtils.
    deserializeCredential(serializedGssCredential);

// Create GSSContext to call other kerberos-secured ser-
vices
GSSContext context = gssManager.createContext(serviceName,
krb5oid,
    serializedGssCredential, GSSContext.DEFAULT_LIFE-
TIME);

```

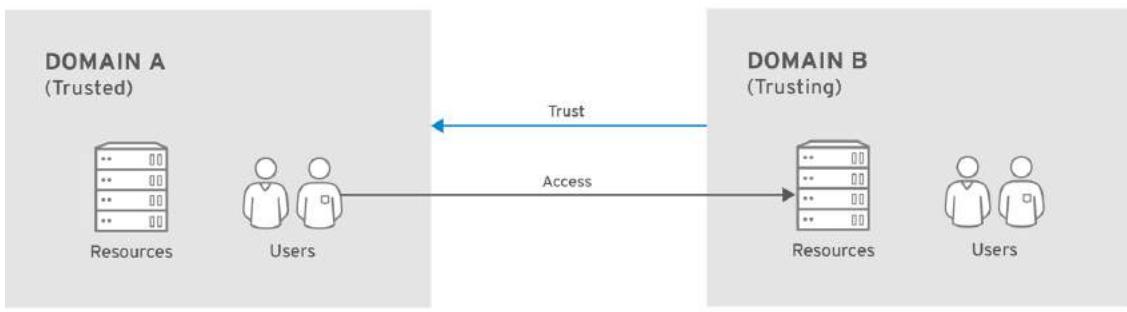
Note that you also need to configure `forwardable` kerberos tickets in `krb5.conf` file and add support for delegated credentials to your browser.

Credential delegation has some security implications so only use it if you really need it. It's highly recommended to use it together with HTTPS. See for example [this article](#) for more details.

#### 6.4.5. Cross-realm trust

In the Kerberos V5 protocol, the `realm` is a set of Kerberos principals defined in the Kerberos database (typically LDAP server). The Kerberos protocol has a concept of cross-realm trust. For example, if there are 2 kerberos realms A and B, the cross-realm trust will allow the users from realm A to access resources (services) of realm B. This means that realm B trusts the realm A.

*Kerberos cross-realm trust*



RHEL\_404973\_0516

The Keycloak server has support for cross-realm trust. There are few things which need to be done to achieve this:

- Configure the Kerberos servers for the cross-realm trust. This step is dependent on the concrete Kerberos server implementations used. In general, it is needed to add the Kerberos principal `krbtgt/B@A` to both Kerberos databases of realm A and B. It is needed that this principal has same keys on both Kerberos realms. This is usually achieved when the principals have same password, key version number and there are same ciphers used in both realms. It is recommended to consult the Kerberos server documentation for more details.

The cross-realm trust is unidirectional by default. If you want bidirectional trust to have realm A also trust realm B, you must also add the principal `krbtgt/A@B` to both Kerberos databases. However, trust is transitive by default. If realm B trusts realm A and realm C trusts realm B, then realm C automatically trusts realm A without a need to have principal `krbtgt/C@A` available. Some additional configuration (for example `capaths`) may be needed to configure on Kerberos client side, so

that the clients are able to find the trust path. Consult the Kerberos documentation for more details.

- Configure KeyCloak server
  - If you use an LDAP storage provider with Kerberos support, you need to configure the server principal for realm B as in this example: `HTTP/mydomain.com@B`. The LDAP server must be able to find the users from realm A if you want users from realm A to successfully authenticate to KeyCloak, as KeyCloak server must be able to do SPNEGO flow and then find the users. For example, kerberos principal user `john@A` must be available as a user in the LDAP under an LDAP DN such as  
`uid=john,ou=People,dc=example,dc=com`. If you want both users from realm A and B to authenticate, you need to ensure that LDAP is able to find users from both realms A and B. We want to improve this limitation in future versions, so you can potentially create more separate LDAP providers for separate realms and ensure that SPNEGO works for both of them.
  - If you use a Kerberos user storage provider (typically the Kerberos without LDAP integration), you need to configure the server principal as `HTTP/mydomain.com@B` and users from both Kerberos realms A and B should be able to authenticate.

For the Kerberos user storage provider, it is recommended that there are no conflicting users among kerberos realms. If conflicting users exist, they will be mapped to the same KeyCloak user. This is also something, which we want to improve in future versions and provi-

de some more flexible mappings from Kerberos principals to Keycloak usernames.

#### 6.4.6. Troubleshooting

If you have issues, we recommend that you enable additional logging to debug the problem:

- Enable `Debug` flag in admin console for Kerberos or LDAP federation providers
- Enable TRACE logging for category `org.keycloak` in logging section of `standalone/configuration/standalone.xml` to receive more info `standalone/log/server.log`
- Add system properties `-Dsun.security.krb5.debug=true` and `-Dsun.security.spnego.debug=true`

### 6.5. X.509 Client Certificate User Authentication

Keycloak supports login with a X.509 client certificate if the server is configured for mutual SSL authentication.

A typical workflow is as follows:

- A client sends an authentication request over SSL/TLS channel
- During SSL/TLS handshake, the server and the client exchange their x.509/v3 certificates
- The container (WildFly) validates the certificate PKIX path and the certificate expiration
- The x.509 client certificate authenticator validates the client certifi-

cate as follows:

- Optionally checks the certificate revocation status using CRL and/or CRL Distribution Points
- Optionally checks the Certificate revocation status using OCSP (Online Certificate Status Protocol)
- Optionally validates whether the key usage in the certificate matches the expected key usage
- Optionally validates whether the extended key usage in the certificate matches the expected extended key usage
- If any of the above checks fails, the x.509 authentication fails
- Otherwise, the authenticator extracts the certificate identity and maps it to an existing user
- Once the certificate is mapped to an existing user, the behavior diverges depending on the authentication flow:
  - In the Browser Flow, the server prompts the user to confirm identity or to ignore it and instead sign in with username/password
  - In the case of the Direct Grant Flow, the server signs in the user

### 6.5.1. Features

#### Supported Certificate Identity Sources

- Match SubjectDN using regular expression
- X500 Subject's e-mail attribute
- X500 Subject's e-mail from Subject Alternative Name Extension

(RFC822Name General Name)

- X500 Subject's other name from Subject Alternative Name Extension. This is typically UPN (User Principal Name)
- X500 Subject's Common Name attribute
- Match IssuerDN using regular expression
- X500 Issuer's e-mail attribute
- X500 Issuer's Common Name attribute
- Certificate Serial Number

## Regular Expressions

The certificate identity can be extracted from either Subject DN or Issuer DN using a regular expression as a filter. For example, the regular expression below will match the e-mail attribute:

```
emailAddress=( . *?)(?:, |$)
```

The regular expression filtering is applicable only if the `Identity Source` is set to either `Match SubjectDN using regular expression` or `Match IssuerDN using regular expression`.

## Mapping certificate identity to an existing user

The certificate identity mapping can be configured to map the extracted user identity to an existing user's username or e-mail or to a custom attribute which value matches the certificate identity. For example, setting the `Identity source` to *Subject's e-mail* and `User mapping method` to *Username or email* will have the X.509 client

certificate authenticator use the e-mail attribute in the certificate's Subject DN as a search criteria to look up an existing user by username or by e-mail.

Please notice that if we disable `Login with email` at realm settings, the same rules will be applied to certificate authentication. In other words, users won't be able to log in using e-mail attribute.

## Other Features: Extended Certificate Validation

- Revocation status checking using CRL
- Revocation status checking using CRL/Distribution Point
- Revocation status checking using OCSP/Responder URI
- Certificate KeyUsage validation
- Certificate ExtendedKeyUsage validation

### 6.5.2. Enable X.509 Client Certificate User Authentication

The following sections describe how to configure WildFly/Undertow and the KeyCloak Server to enable X.509 client certificate authentication.

#### Enable mutual SSL in WildFly

See [Enable SSL](#) and [SSL](#) for the instructions how to enable SSL in WildFly.

- Open {project\_dirref}/standalone/configuration/standalone.xml and add a new realm:

```
<security-realms>
    <security-realm name="ssl-realm">
        <server-identities>
            <ssl>
                <keystore path="servercert.jks"
                    relative-to="jboss.server.config.
dir"
                    keystore-password="servercert
password"/>
                </ssl>
            </server-identities>
            <authentication>
                <truststore path="truststore.jks"
                    relative-to="jboss.server.config.
dir"
                    keystore-password="truststore pass-
word"/>
                </authentication>
            </security-realm>
    </security-realms>
```

### ssl/keystore

The `ssl` element contains the `keystore` element that defines how to load the server public key pair from a JKS keystore

### ssl/keystore/path

A path to a JKS keystore

### ssl/keystore/relative-to

Defines a path the keystore path is relative to

### ssl/keystore/keystore-password

The password to open the keystore

### ssl/keystore/alias (**optional**)

The alias of the entry in the keystore. Set it if the keystore contains

multiple entries

#### ssl/keystore/key-password (optional)

The private key password, if different from the keystore password.

#### authentication/truststore

Defines how to load a trust store to verify the certificate presented by the remote side of the inbound/outgoing connection. Typically, the truststore contains a collection of trusted CA certificates.

#### authentication/truststore/path

A path to a JKS keystore that contains the certificates of the trusted CAs (certificate authorities)

#### authentication/truststore/relative-to

Defines a path the truststore path is relative to

#### authentication/truststore/keystore-password

The password to open the truststore

## Enable https listener

See [HTTPS Listener](#) for the instructions how to enable HTTPS in WildFly.

- Add the <https-listener> element as shown below:

```
<subsystem xmlns="{subsystem_undertow_xml_urn}">
    ...
    <server name="default-server">
        <https-listener name="default"
                        socket-binding="https"
                        security-realm="ssl-realm">
```

```
        verify-client="REQUESTED"/>
    </server>
</subsystem>
```

### https-listener/security-realm

The value must match the name of the realm from the previous section

### https-listener/verify-client

If set to `REQUESTED`, the server will optionally ask for a client certificate. Setting the attribute to `REQUIRED` will have the server to refuse inbound connections if no client certificate has been provided.

## 6.5.3. Adding X.509 Client Certificate Authentication to a Browser Flow

- Select a realm, click on Authentication link, select the "Browser" flow
- Make a copy of the built-in "Browser" flow. You may want to give the new flow a distinctive name, i.e. "X.509 Browser"
- Using the drop down, select the copied flow, and click on "Add execution"
- Select "X509/Validate User Form" using the drop down and click on "Save"

### Create Authenticator Execution

Flows	Bindings	Required Actions	Password Policy	OTP Policy
Provider	X509/Validate Username Form			
<input type="button" value="Save"/> <input type="button" value="Cancel"/>				

- Using the up/down arrows, change the order of the "X509/Validate Username Form" by moving it above the "Browser Forms" execution, and set the requirement to "ALTERNATIVE"

#### Authentication

Auth Type	Requirement	Actions
X509 Browser	<input checked="" type="radio"/> ALTERNATIVE	<input type="radio"/> DISABLED
Kerberos	<input type="radio"/> ALTERNATIVE	<input type="radio"/> REQUIRED
Identity Provider Redirector	<input checked="" type="radio"/> ALTERNATIVE	<input type="radio"/> DISABLED
X509/Validate Username Form (x509-form-config)	<input checked="" type="radio"/> ALTERNATIVE	<input type="radio"/> DISABLED
X509 Browser Forms	<input checked="" type="radio"/> ALTERNATIVE	<input type="radio"/> REQUIRED
	<input checked="" type="radio"/> REQUIRED	<input type="radio"/> DISABLED
	<input type="radio"/> OPTIONAL	<input type="radio"/> DISABLED

- Select the "Bindings" tab, find the drop down for "Browser Flow". Select the newly created X509 browser flow from the drop down and click on "Save".

The screenshot shows the Keycloak Admin UI under the 'Authentication' section. On the left, there's a sidebar with 'Master' selected. Under 'Configure', 'Authentication' is highlighted. The main area has tabs for 'Flows', 'Bindings' (which is active), 'Required Actions', 'Password Policy', and 'OTP Policy'. In the 'Bindings' tab, several dropdown menus are displayed: 'Browser Flow' set to 'X.509 Browser', 'Registration Flow' set to 'registration', 'Direct Grant Flow' set to 'direct grant', 'Reset Credentials' set to 'reset credentials', and 'Client Authentication' set to 'clients'. At the bottom right of this section are 'Save' and 'Cancel' buttons.

## Configuring X.509 Client Certificate Authentication

## X509-form-config

ID	15557226-aaba-45c8-a4d3-97047a6c1cc3
Alias	x509-form-config
User Identity Source	Subject's e-mail
A regular expression to extract user identity	(^.?)(?:\$)
User mapping method	Username or Email
A name of user attribute	usercertificate
CRL Checking Enabled	<input checked="" type="checkbox"/> ON <input type="checkbox"/> OFF
Enable CRL Distribution Point to check certificate revocation status	
CRL File path	crl.pem
OCSP Checking Enabled	<input type="checkbox"/> OFF
OCSP Responder Uri	
Validate Key Usage	
Validate Extended Key Usage	
Bypass identity confirmation	<input type="checkbox"/> OFF

### User Identity Source

Defines how to extract the user identity from a client certificate.

### A regular expression (optional)

Defines a regular expression to use as a filter to extract the certificate identity. The regular expression must contain a single group.

### User Mapping Method

Defines how to match the certificate identity to an existing user.

*Username or e-mail* will search for an existing user by username or e-mail. *Custom Attribute Mapper* will search for an existing user with a custom attribute which value matches the certificate identity. The name of the custom attribute is configurable.

### A name of user attribute (optional)

A custom attribute which value will be matched against the certificate identity.

#### CRL Checking Enabled (optional)

Defines whether to check the revocation status of the certificate using Certificate Revocation List.

#### Enable CRL Distribution Point to check certificate revocation status (optional)

Defines whether to use CDP to check the certificate revocation status. Most PKI authorities include CDP in their certificates.

#### CRL file path (optional)

Defines a path to a file that contains a CRL list. The value must be a path to a valid file if `CRL Checking Enabled` option is turned on.

#### OCSP Checking Enabled (optional)

Defines whether to check the certificate revocation status using Online Certificate Status Protocol.

#### OCSP Responder URI (optional)

Allows to override a value of the OCSP responder URI in the certificate.

#### Validate Key Usage (optional)

Verifies whether the certificate's KeyUsage extension bits are set. For example, "digitalSignature,KeyEncipherment" will verify if bits 0 and 2 in the KeyUsage extension are asserted. Leave the parameter empty to disable the Key Usage validation. See [RFC5280, Section-](#)

[4.2.1.3](#). The server will raise an error only when flagged as critical by the issuing CA and there is a key usage extension mismatch.

#### Validate Extended Key Usage (optional)

Verifies one or more purposes as defined in the Extended Key Usage extension. See [RFC5280, Section-4.2.1.12](#). Leave the parameter empty to disable the Extended Key Usage validation. The server will raise an error only when flagged as critical by the issuing CA and there is a key usage extension mismatch.

#### Bypass identity confirmation

If set, X.509 client certificate authentication will not prompt the user to confirm the certificate identity and will automatically sign in the user upon successful authentication.

### 6.5.4. Adding X.509 Client Certificate Authentication to a Direct Grant Flow

- Using KeyCloak admin console, click on "Authentication" and select the "Direct Grant" flow,
- Make a copy of the build-in "Direct Grant" flow. You may want to give the new flow a distinctive name, i.e. "X509 Direct Grant",
- Delete "Username Validation" and "Password" authenticators,
- Click on "Add execution" and add "X509/Validate Username" and click on "Save" to add the execution step to the parent flow.

#### Create Authenticator Execution

The screenshot shows a portion of the Keycloak admin interface for creating a new authenticator execution. At the top, there are tabs for 'Flows', 'Bindings', 'Required Actions', 'Password Policy', and 'OTP Policy'. The 'Flows' tab is currently active. Below the tabs, there is a dropdown labeled 'Provider' containing the option 'X509/Validate Username'. At the bottom of the dialog are two buttons: 'Save' and 'Cancel'.

- Change the `Requirement` to *REQUIRED*.

Auth Type	Action	Requirement	Actions
X509 Direct Grant	X509/Validate Username (x509-direct-config)	<input checked="" type="radio"/> REQUIRED	<a href="#">Actions</a>

- Set up the x509 authentication configuration by following the steps described earlier in the x.509 Browser Flow section.
- Select the "Bindings" tab, find the drop down for "Direct Grant Flow". Select the newly created X509 direct grant flow from the drop down and click on "Save".

### 6.5.5. Client certificate lookup

When an HTTP request is sent directly to KeyCloak server, the WildFly undertow subsystem will establish an SSL handshake and extract the client certificate. The client certificate will be then saved to the attribute `javax.servlet.request.X509Certificate` of the HTTP request, as specified in the servlet specification. The KeyCloak X509 authenticator will be then able to lookup the certificate from this attribute.

However, when the KeyCloak server listens to HTTP requests behind a

load balancer or reverse proxy, it may be the proxy server which extracts the client certificate and establishes the mutual SSL connection. A reverse proxy usually puts the authenticated client certificate in the HTTP header of the underlying request and forwards it to the back end Keycloak server. In this case, Keycloak must be able to look up the X.509 certificate chain from the HTTP headers instead of from the attribute of HTTP request, as is done for Undertow.

If Keycloak is behind a reverse proxy, you usually need to configure alternative provider of the `x509cert-lookup` SPI in `{project_dir-ref}/standalone/configuration/standalone.xml`. Along with the `default` provider, which looks up the certificate from the HTTP header, we also have two additional built-in providers: `haproxy` and `apache`, which are described next.

## HAProxy certificate lookup provider

You can use this provider when your Keycloak server is behind an HAProxy reverse proxy. Configure the server like this:

```
<spi name="x509cert-lookup">
    <default-provider>haproxy</default-provider>
    <provider name="haproxy" enabled="true">
        <properties>
            <property name="sslClientCert" value="SSL_CLIENT_CERT"/>
            <property name="sslCertChainPrefix" value="CERT_CHAIN"/>
            <property name="certificateChainLength" value="10"/>
        </properties>
    </provider>
</spi>
```

In this example configuration, the client certificate will be looked up from the HTTP header, `SSL_CLIENT_CERT`, and the other certificates from its chain will be looked up from HTTP headers like `CERT_CHAIN_0`, `CERT_CHAIN_1`, ..., `CERT_CHAIN_9`. The attribute `certificateChainLength` is the maximum length of the chain, so the last one tried attribute would be `CERT_CHAIN_9`.

Consult the [HAProxy documentation](#) for the details of how the HTTP Headers for the client certificate and client certificate chain can be configured and their proper names.

## Apache certificate lookup provider

You can use this provider when your Keycloak server is behind an Apache reverse proxy. Configure the server like this:

```
<spi name="x509cert-lookup">
    <default-provider>apache</default-provider>
    <provider name="apache" enabled="true">
        <properties>
            <property name="sslClientCert" value="SSL_CLIENT_CERT"/>
            <property name="sslCertChainPrefix" value="CERT_CHAIN"/>
            <property name="certificateChainLength" value="10"/>
        </properties>
    </provider>
</spi>
```

The configuration is same as for the `haproxy` provider. Consult the Apache documentation on [mod\\_ssl](#) and [mod\\_headers](#) for the details of how the HTTP Headers for the client certificate and client certificate chain can be configured and their proper names.

## Nginx certificate lookup provider

You can use this provider when your Keycloak server is behind an Nginx reverse proxy. Configure the server like this:

```
<spi name="x509cert-lookup">
    <default-provider>nginx</default-provider>
    <provider name="nginx" enabled="true">
        <properties>
            <property name="sslClientCert" value="ssl-client-cert"/>
            <property name="sslCertChainPrefix" value="USE-LESS"/>
            <property name="certificateChainLength" value="2"/>
        </properties>
    </provider>
</spi>
```

NGINX [SSL/TLS module](#) does not expose the client certificate chain, so Keycloak NGINX certificate lookup provider is rebuilding it using the [{installguide-truststore\\_name}](#). Please populate Keycloak truststore using keytool CLI with all root and intermediate CA's needed for rebuilding client certificate chain.

Consult the NGINX documentation for the details of how the HTTP Headers for the client certificate can be configured. Example of NGINX configuration file :

```
...
server {
    ...
    ssl_client_certificate      trusted-ca-
list-for-client-auth.pem;
```

```
ssl_verify_client optional_no_ca;
ssl_verify_depth 2;
...
location / {
    ...
    proxy_set_header ssl-client-cert $ssl_client_escaped_cert;
    ...
}
...
}
```

all certificates in trusted-ca-list-for-client-auth.pem must be added to [{installguide\\_truststore\\_name}](#).

## Other reverse proxy implementations

We do not have built-in support for other reverse proxy implementations. However, it is possible that other reverse proxies can be made to behave in a similar way to `apache` or `haproxy` and that some of those providers can be used. If none of those works, you may need to create your own implementation of the `org.keycloak.services.x509.X509ClientCertificateLookupFactory` and `org.keycloak.services.x509.X509ClientCertificateLookup` provider. See the [{developerguide!}](#) for the details on how to add your own provider.

### 6.5.6. Troubleshooting

#### Dumping HTTP headers

If you want to view what the reverse proxy is sending to Keycloak, simply activate [RequestDumpingHandler](#) and consult `server.log` file.

## Enable TRACE logging under the logging subsystem

```
...
<profile>
    <subsystem xmlns="urn:jboss:domain:logging:3.0">
...
    <logger category="org.keycloak.authentication.authenticators.x509">
        <level name="TRACE"/>
    </logger>
    <logger category="org.keycloak.services.x509">
        <level name="TRACE"/>
    </logger>
```

**WARNING:** Don't use RequestDumpingHandler or TRACE logging in production.

## Direct Grant authentication with X.509

The following template can be used to request a token using the Resource Owner Password Credentials Grant:

```
$ curl https://[host][:port]/auth/realms/master/protocol/openid-connect/token \
    --insecure \
    --data "grant_type=password&scope=openid profile&username=&password=&client_id=CLIENT_ID&client_secret=CLIENT_SECRET" \
    -E /path/to/client_cert.crt \
    --key /path/to/client_cert.key
```

[host][:port]

The host and the port number of a remote KeyCloak server that has been configured to allow users authenticate with x.509 client certificates using the Direct Grant Flow.

`CLIENT_ID`

A client id.

`CLIENT_SECRET`

For confidential clients, a client secret; otherwise, leave it empty.

`client_cert.crt`

A public key certificate that will be used to verify the identity of the client in mutual SSL authentication. The certificate should be in PEM format.

`client_cert.key`

A private key in the public key pair. Also expected in PEM format.

---

## 7. SSO Protocols

The chapter gives a brief overview of the authentication protocols and how the KeyCloak authentication server and the applications it secures interact with these protocols.

### 7.1. OpenID Connect

[OpenID Connect](#) (OIDC) is an authentication protocol that is an extension of [OAuth 2.0](#). While OAuth 2.0 is only a framework for building authorization protocols and is mainly incomplete, OIDC is a full-fledged authentication and authorization protocol. OIDC also makes heavy use of the [Json Web Token](#) (JWT) set of standards. These standards define an identity token JSON format and ways to digitally sign and encrypt that data in a compact and web-friendly way.

There are really two types of use cases when using OIDC. The first is an application that asks the KeyCloak server to authenticate a user for them. After a successful login, the application will receive an *identity token* and an *access token*. The *identity token* contains information about the user such as username, email, and other profile information. The *access token* is digitally signed by the realm and contains access information (like user role mappings) that the application can use to determine what resources the user is allowed to access on the application.

The second type of use cases is that of a client that wants to gain access to remote services. In this case, the client asks KeyCloak to obtain an *access token* it can use to invoke on other remote services on behalf of

the user. KeyCloak authenticates the user then asks the user for consent to grant access to the client requesting it. The client then receives the *access token*. This *access token* is digitally signed by the realm. The client can make REST invocations on remote services using this *access token*. The REST service extracts the *access token*, verifies the signature of the token, then decides based on access information within the token whether or not to process the request.

### 7.1.1. OIDC Auth Flows

OIDC has different ways for a client or application to authenticate a user and receive an *identity* and *access token*. Which path you use depends greatly on the type of application or client requesting access. All of these flows are described in the OIDC and OAuth 2.0 specifications so only a brief overview will be provided here.

#### Authorization Code Flow

This is a browser-based protocol and it is what we recommend you use to authenticate and authorize browser-based applications. It makes heavy use of browser redirects to obtain an *identity* and *access token*. Here's a brief summary:

1. Browser visits application. The application notices the user is not logged in, so it redirects the browser to KeyCloak to be authenticated. The application passes along a callback URL (a redirect URL) as a query parameter in this browser redirect that KeyCloak will use when it finishes authentication.
2. KeyCloak authenticates the user and creates a one-time, very short lived, temporary code. KeyCloak redirects back to the application using the callback URL provided earlier and additionally adds the

temporary code as a query parameter in the callback URL.

3. The application extracts the temporary code and makes a background out of band REST invocation to Keycloak to exchange the code for an *identity*, *access* and *refresh* token. Once this temporary code has been used once to obtain the tokens, it can never be used again. This prevents potential replay attacks.

It is important to note that *access* tokens are usually short lived and often expired after only minutes. The additional *refresh* token that was transmitted by the login protocol allows the application to obtain a new access token after it expires. This refresh protocol is important in the situation of a compromised system. If access tokens are short lived, the whole system is only vulnerable to a stolen token for the lifetime of the access token. Future refresh token requests will fail if an admin has revoked access. This makes things more secure and more scalable.

Another important aspect of this flow is the concept of a *public* vs. a *confidential* client. *Confidential* clients are required to provide a client secret when they exchange the temporary codes for tokens. *Public* clients are not required to provide this client secret. *Public* clients are perfectly fine so long as HTTPS is strictly enforced and you are very strict about what redirect URIs are registered for the client. HTML5/JavaScript clients always have to be *public* clients because there is no way to transmit the client secret to them in a secure manner. Again, this is ok so long as you use HTTPS and strictly enforce redirect URI registration. This guide goes more detail into this in the [Managing Clients](#) chapter.

Keycloak also supports the optional [Proof Key for Code Exchange](#) specification.

## Implicit Flow

This is a browser-based protocol that is similar to Authorization Code Flow except there are fewer requests and no refresh tokens involved. We do not recommend this flow as there remains the possibility of *access* tokens being leaked in the browser history as tokens are transmitted via redirect URIs (see below). Also, since this flow doesn't provide the client with a refresh token, access tokens would either have to be long-lived or users would have to re-authenticate when they expired. This flow is supported because it is in the OIDC and OAuth 2.0 specification. Here's a brief summary of the protocol:

1. Browser visits application. The application notices the user is not logged in, so it redirects the browser to Keycloak to be authenticated. The application passes along a callback URL (a redirect URL) as a query parameter in this browser redirect that Keycloak will use when it finishes authentication.
2. Keycloak authenticates the user and creates an *identity* and *access* token. Keycloak redirects back to the application using the callback URL provided earlier and additionally adding the *identity* and *access* tokens as query parameters in the callback URL.
3. The application extracts the *identity* and *access* tokens from the callback URL.

## Resource Owner Password Credentials Grant (Direct Access Grants)

This is referred to in the Admin Console as *Direct Access Grants*. This is used by REST clients that want to obtain a token on behalf of a user. It is one HTTP POST request that contains the credentials of the user as

well as the id of the client and the client's secret (if it is a confidential client). The user's credentials are sent within form parameters. The HTTP response contains *identity*, *access*, and *refresh* tokens.

## Client Credentials Grant

This is also used by REST clients, but instead of obtaining a token that works on behalf of an external user, a token is created based on the metadata and permissions of a service account that is associated with the client. More info together with example is in [Service Accounts](#) chapter.

### 7.1.2. KeyCloak Server OIDC URI Endpoints

Here's a list of OIDC endpoints that the KeyCloak publishes. These URLs are useful if you are using a non-KeyCloak client adapter to talk OIDC with the auth server. These are all relative URLs and the root of the URL being the HTTP(S) protocol, hostname, and usually path prefixed with /auth: i.e. https://localhost:8080/auth

#### **/realms/{realm-name}/protocol/openid-connect/token**

This is the URL endpoint for obtaining a temporary code in the Authorization Code Flow or for obtaining tokens via the Implicit Flow, Direct Grants, or Client Grants.

#### **/realms/{realm-name}/protocol/openid-connect/auth**

This is the URL endpoint for the Authorization Code Flow to turn a temporary code into a token.

#### **/realms/{realm-name}/protocol/openid-connect/logout**

This is the URL endpoint for performing logouts.

## **/realms/{realm-name}/protocol/openid-connect/userinfo**

This is the URL endpoint for the User Info service described in the OIDC specification.

In all of these replace *{realm-name}* with the name of the realm.

## **7.2. SAML**

[SAML 2.0](#) is a similar specification to OIDC but a lot older and more mature. It has its roots in SOAP and the plethora of WS-\* specifications so it tends to be a bit more verbose than OIDC. SAML 2.0 is primarily an authentication protocol that works by exchanging XML documents between the authentication server and the application. XML signatures and encryption is used to verify requests and responses.

There are really two types of use cases when using SAML. The first is an application that asks the KeyCloak server to authenticate a user for them. After a successful login, the application will receive an XML document that contains something called a SAML assertion that specify various attributes about the user. This XML document is digitally signed by the realm and contains access information (like user role mappings) that the application can use to determine what resources the user is allowed to access on the application.

The second type of use cases is that of a client that wants to gain access to remote services. In this case, the client asks KeyCloak to obtain an SAML assertion it can use to invoke on other remote services on behalf of the user.

### **7.2.1. SAML Bindings**

SAML defines a few different ways to exchange XML documents when executing the authentication protocol. The *Redirect* and *Post* bindings cover browser based applications. The *ECP* binding covers REST invocations. There are other binding types but KeyCloak only supports those three.

## Redirect Binding

The *Redirect* Binding uses a series of browser redirect URIs to exchange information. This is a rough overview of how it works.

1. The user visits the application and the application finds the user is not authenticated. It generates an XML authentication request document and encodes it as a query param in a URI that is used to redirect to the KeyCloak server. Depending on your settings, the application may also digitally sign this XML document and also stuff this signature as a query param in the redirect URI to KeyCloak. This signature is used to validate the client that sent this request.
2. The browser is redirected to KeyCloak. The server extracts the XML auth request document and verifies the digital signature if required. The user then has to enter in their credentials to be authenticated.
3. After authentication, the server generates an XML authentication response document. This document contains a SAML assertion that holds metadata about the user like name, address, email, and any role mappings the user might have. This document is almost always digitally signed using XML signatures, and may also be encrypted.
4. The XML auth response document is then encoded as a query param in a redirect URI that brings the browser back to the applica-

tion. The digital signature is also included as a query param.

5. The application receives the redirect URI and extracts the XML document and verifies the realm's signature to make sure it is receiving a valid auth response. The information inside the SAML assertion is then used to make access decisions or display user data.

## POST Binding

The SAML *POST* binding works almost the exact same way as the *Redirect* binding, but instead of GET requests, XML documents are exchanged by POST requests. The *POST* Binding uses JavaScript to trick the browser into making a POST request to the KeyCloak server or application when exchanging documents. Basically HTTP responses contain an HTML document that contains an HTML form with embedded JavaScript. When the page is loaded, the JavaScript automatically invokes the form. You really don't need to know about this stuff, but it is a pretty clever trick.

*POST* binding is usually recommended because of security and size restrictions. When using *REDIRECT* the SAML response is part of the URL (it is a query parameter as it was explained before), so it can be captured in logs and it is considered less secure. Regarding size, if the assertion contains a lot or large attributes sending the document inside the HTTP payload is always better than in the more limited URL.

## ECP

ECP stands for "Enhanced Client or Proxy", a SAML v.2.0 profile which allows for the exchange of SAML attributes outside the context of a web browser. This is used most often for REST or SOAP-based clients.

### 7.2.2. KeyCloak Server SAML URI Endpoints

KeyCloak really only has one endpoint for all SAML requests.

```
http(s)://authserver.host/auth/realms/{realm-na-  
me}/protocol/saml
```

All bindings use this endpoint.

## 7.3. OpenID Connect vs. SAML

Choosing between OpenID Connect and SAML is not just a matter of using a newer protocol (OIDC) instead of the older more mature protocol (SAML).

In most cases KeyCloak recommends using OIDC.

SAML tends to be a bit more verbose than OIDC.

Beyond verbosity of exchanged data, if you compare the specifications you'll find that OIDC was designed to work with the web while SAML was retrofitted to work on top of the web. For example, OIDC is also more suited for HTML5/JavaScript applications because it is easier to implement on the client side than SAML. As tokens are in the JSON format, they are easier to consume by JavaScript. You will also find several nice features that make implementing security in your web applications easier. For example, check out the [iframe trick](#) that the specification uses to easily determine if a user is still logged in or not.

SAML has its uses though. As you see the OIDC specifications evolve you see they implement more and more features that SAML has had for years. What we often see is that people pick SAML over OIDC because

of the perception that it is more mature and also because they already have existing applications that are secured with it.

## 7.4. Docker Registry v2 Authentication

Docker authentication is disabled by default. To enable see [`{installguide\_profile\_name}`](#).

[Docker Registry V2 Authentication](#) is an OIDC-Like protocol used to authenticate users against a Docker registry. KeyCloak's implementation of this protocol allows for a KeyCloak authentication server to be used by a Docker client to authenticate against a registry. While this protocol uses fairly standard token and signature mechanisms, it has a few wrinkles that prevent it from being treated as a true OIDC implementation. The largest deviations include a very specific JSON format for requests and responses as well as the ability to understand how to map repository names and permissions to the OAuth scope mechanism.

### 7.4.1. Docker Auth Flow

The [Docker API documentation](#) best describes and illustrates this process, however a brief summary will be given below from the perspective of the KeyCloak authentication server.

This flow assumes that a `docker login` command has already been performed

- The flow begins when the Docker client requests a resource from the Docker registry. If the resource is protected and no auth token is present in the request, the Docker registry server will respond to the

client with a 401 + some information on required permissions and where to find the authorization server.

- The Docker client will construct an authentication request based on the 401 response from the Docker registry. The client will then use the locally cached credentials (from a previously run `docker login` command) as part of a [HTTP Basic Authentication](#) request to the KeyCloak authentication server.
- The KeyCloak authentication server will attempt to authenticate the user and return a JSON body containing an OAuth-style Bearer token.
- The Docker client will get the bearer token from the JSON response and use it in the Authorization header to request the protected resource.
- When the Docker registry receives the new request for the protected resource with the token from the KeyCloak server, the registry validates the token and grants access to the requested resource (if appropriate).

#### 7.4.2. KeyCloak Docker Registry v2 Authentication Server URI Endpoints

KeyCloak really only has one endpoint for all Docker auth v2 requests.

```
http(s)://authserver.host/auth/realms/{realm-name}/protocol/docker-v2
```

# 8. Managing Clients

Clients are entities that can request authentication of a user. Clients come in two forms. The first type of client is an application that wants to participate in single-sign-on. These clients just want Keycloak to provide security for them. The other type of client is one that is requesting an access token so that it can invoke other services on behalf of the authenticated user. This section discusses various aspects around configuring clients and various ways to do it.

## 8.1. OIDC Clients

[OpenID Connect](#) is the preferred protocol to secure applications. It was designed from the ground up to be web friendly and work best with HTML5/JavaScript applications.

To create an OIDC client go to the `Clients` left menu item. On this page you'll see a `Create` button on the right.

### *Clients*

Client ID	Enabled	Base URL	Actions		
account	True	/auth/realms/demo/account	Edit	Export	Delete
admin-cli	True	Not defined	Edit	Export	Delete
broker	True	Not defined	Edit	Export	Delete
realm-management	True	Not defined	Edit	Export	Delete
security-admin-console	True	/auth/admin/demo/console/index.html	Edit	Export	Delete

This will bring you to the `Add Client` page.

## Add Client

The screenshot shows the Keycloak administration interface. The left sidebar is titled 'Master' and contains sections for 'Configure' (Realm Settings, Clients, Client Scopes, Roles, Identity Providers, User Federation, Authentication) and 'Manage' (Groups). The 'Clients' section is currently selected. The main content area is titled 'Add Client'. It includes fields for 'Import' (with a 'Select file' button), 'Client ID' (set to 'myapp'), 'Client Protocol' (set to 'openid-connect'), and 'Root URL' (set to 'http://localhost:8080/myapp'). At the bottom are 'Save' and 'Cancel' buttons.

Enter in the `Client ID` of the client. This should be a simple alpha-numeric string that will be used in requests and in the KeyCloak database to identify the client. Next select `openid-connect` in the `Client Protocol` drop down box. Finally enter in the base URL of your application in the `Root URL` field and click `Save`. This will create the client and bring you to the client `Settings` tab.

## Client Settings

The screenshot shows the Keycloak 'Clients' configuration page for a client named 'myapp'. The left sidebar has a dark theme with white text. It shows 'Master' selected under 'Configure' and 'Clients' is highlighted. Other options include 'Realm Settings', 'Client Templates', 'Roles', 'Identity Providers', 'User Federation', and 'Authentication'. The main area is titled 'Myapp' and shows the following configuration:

- Client ID:** myapp
- Name:** (empty)
- Description:** (empty)
- Enabled:** ON
- Consent Required:** OFF
- Client Protocol:** openid-connect
- Client Template:** (empty)
- Access Type:** public
- Standard Flow Enabled:** ON
- Implicit Flow Enabled:** OFF
- Direct Access Grants Enabled:** ON
- Authorization Enabled:** OFF
- Root URL:** http://localhost:8080
- \*Valid Redirect URIs:** http://localhost:8080/\* (with a plus sign icon to add more)
- Base URL:** (empty)
- Admin URL:** http://localhost:8080
- Web Origins:** http://localhost:8080 (with a plus sign icon to add more)

Let's walk through each configuration item on this page.

## Client ID

This specifies an alpha-numeric string that will be used as the client identifier for OIDC requests.

## Name

This is the display name for the client whenever it is displayed in a Keycloak UI screen. You can localize the value of this field by setting up a replacement string value i.e. \${myapp}. See the [{developerguide!}](#) for

more information.

## Description

This specifies the description of the client. This can also be localized.

## Enabled

If this is turned off, the client will not be allowed to request authentication.

## Consent Required

If this is on, then users will get a consent page which asks the user if they grant access to that application. It will also display the metadata that the client is interested in so that the user knows exactly what information the client is getting access to. If you've ever done a social login to Google, you'll often see a similar page. Keycloak provides the same functionality.

## Access Type

This defines the type of the OIDC client.

### *confidential*

Confidential access type is for server-side clients that need to perform a browser login and require a client secret when they turn an access code into an access token, (see [Access Token Request](#) in the OAuth 2.0 spec for more details). This type should be used for server-side applications.

### ***public***

Public access type is for client-side clients that need to perform a browser login. With a client-side application there is no way to keep a secret safe. Instead it is very important to restrict access by configuring correct redirect URIs for the client.

### ***bearer-only***

Bearer-only access type means that the application only allows bearer token requests. If this is turned on, this application cannot participate in browser logins.

## **Root URL**

If Keycloak uses any configured relative URLs, this value is prepended to them.

## **Valid Redirect URIs**

This is a required field. Enter in a URL pattern and click the + sign to add. Click the - sign next to URLs you want to remove. Remember that you still have to click the `Save` button! Wildcards (\*) are only allowed at the end of a URI, i.e. `http://host.com/*`

You should take extra precautions when registering valid redirect URI patterns. If you make them too general you are vulnerable to attacks.

See [Threat Model Mitigation](#) chapter for more information.

## **Base URL**

If Keycloak needs to link to the client, this URL is used.

## **Standard Flow Enabled**

If this is on, clients are allowed to use the OIDC [Authorization Code Flow](#).

## **Implicit Flow Enabled**

If this is on, clients are allowed to use the OIDC [Implicit Flow](#).

## **Direct Access Grants Enabled**

If this is on, clients are allowed to use the OIDC [Direct Access Grants](#).

## **Admin URL**

For KeyCloak specific client adapters, this is the callback endpoint for the client. The KeyCloak server will use this URI to make callbacks like pushing revocation policies, performing backchannel logout, and other administrative operations. For KeyCloak servlet adapters, this can be the root URL of the servlet application. For more information see [{adapterguide!}](#).

## **Web Origins**

This option centers around [CORS](#) which stands for Cross-Origin Resource Sharing. If browser JavaScript tries to make an AJAX HTTP request to a server whose domain is different from the one the JavaScript code came from, then the request must use CORS. The server must handle CORS requests in a special way, otherwise the browser will not display or allow the request to be processed. This protocol exists to protect against XSS, CSRF and other JavaScript-based attacks.

Keycloak has support for validated CORS requests. The way it works is that the domains listed in the `Web Origins` setting for the client are embedded within the access token sent to the client application. The client application can then use this information to decide whether or not to allow a CORS request to be invoked on it. This is an extension to the OIDC protocol so only Keycloak client adapters support this feature. See [{adapterguide!}](#) for more information.

To fill in the `Web Origins` data, enter in a base URL and click the + sign to add. Click the - sign next to URLs you want to remove. Remember that you still have to click the `Save` button!

### 8.1.1. Advanced Settings

#### **OAuth 2.0 Mutual TLS Certificate Bound Access Tokens Enabled**

Mutual TLS binds an access token and a refresh token with a client certificate exchanged during TLS handshake. This prevents an attacker who finds a way to steal these tokens from exercising the tokens. This type of token is called a holder-of-key token. Unlike bearer tokens, the recipient of a holder-of-key token can verify whether the sender of the token is legitimate.

If the following conditions are satisfied on a token request, Keycloak will bind an access token and a refresh token with a client certificate and issue them as holder-of-key tokens. If all conditions are not met, Keycloak rejects the token request.

- The feature is turned on
- A token request is sent to the token endpoint in an authorization

code flow or a hybrid flow

- On TLS handshake, Keycloak requests a client certificate and a client send its client certificate
- On TLS handshake, Keycloak successfully verifies the client certificate

To enable mutual TLS in Keycloak, see [Enable mutual SSL in WildFly](#).

In the following cases, Keycloak will verify the client sending the access token or the refresh token; if verification fails, Keycloak rejects the token.

- A token refresh request is sent to the token endpoint with a holder-of-key refresh token
- A UserInfo request is sent to UserInfo endpoint with a holder-of-key access token
- A logout request is sent to Logout endpoint with a holder-of-key refresh token

Please see [Mutual TLS Client Certificate Bound Access Tokens](#) in the OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens for more details.

**WARNING:** None of the keycloak client adapters currently support holder-of-key token verification. Instead, keycloak adapters currently treat access and refresh tokens as bearer tokens.

### 8.1.2. Confidential Client Credentials

If you've set the client's [access type](#) to `confidential` in the client's `Settings` tab, a new `Credentials` tab will show up. As part of dealing with this type of client you have to configure the client's credentials.

### *Credentials Tab*

The screenshot shows the Keycloak administration interface. On the left, a sidebar menu is open under the 'Master' realm, with 'Clients' selected. The main content area shows a client named 'Myapp'. The 'Credentials' tab is active. Under the 'Client Authenticator' section, the dropdown is set to 'Client Id and Secret'. A text input field contains the secret value 'f07d7fb6-cd29-443f-a4da-df22c76c1', and a 'Regenerate Secret' button is visible. Below this, there is a section for 'Registration access token' with its own input field and a 'Regenerate registration access token' button.

The `Client Authenticator` list box specifies the type of credential you are going to use for your confidential client. It defaults to client ID and secret. The secret is automatically generated for you and the `Regenerate Secret` button allows you to recreate this secret if you want or need to.

Alternatively, you can opt to use a signed Json Web Token (JWT) or x509 certificate validation (also called Mutual TLS) instead of a secret.

### *Signed JWT*

The screenshot shows the Keycloak administration interface. On the left, a sidebar menu is open under the 'Master' realm, showing options like 'Clients', 'Realm Settings', 'Client Templates', 'Roles', 'Identity Providers', 'User Federation', and 'Authentication'. The 'Clients' option is selected. In the main content area, the path 'Clients > myapp' is shown. The 'Credentials' tab is active. Under the 'Credentials' tab, the 'Client Authenticator' dropdown is set to 'Signed Jwt'. Below it, the 'Use JWKs URL' switch is turned off. A note says 'No client certificate configured'. At the bottom of this section, there are buttons for 'Generate new keys and certificate', 'Import Certificate', 'Save', and 'Cancel'. Below this, there's a section for 'Registration access token' with a text input field and a 'Regenerate registration access token' button.

When choosing this credential type you will have to also generate a private key and certificate for the client. The private key will be used to sign the JWT, while the certificate is used by the server to verify the signature. Click on the `Generate new keys and certificate` button to start this process.

### Generate Keys

The screenshot shows a modal dialog titled 'Generate Private Key'. The dialog is part of the Keycloak administration interface, specifically for generating client private keys. The path 'Clients > myapp > Credentials > Generate Client Private Key' is visible in the top navigation. The dialog has fields for 'Archive Format' (set to 'JKS'), 'Key Alias' (set to 'myapp'), 'Key Password' (a masked password), and 'Store Password' (a masked password). At the bottom of the dialog are two buttons: 'Generate and Download' (in blue) and 'Cancel'.

When you generate these keys, KeyCloak will store the certificate, and you'll need to download the private key and certificate for your client to use. Pick the archive format you want and specify the password for the private key and store.

You can also opt to generate these via an external tool and just import the client's certificate.

### *Import Certificate*

The screenshot shows the Keycloak administrative interface. The left sidebar is titled 'Master' and contains the following navigation items: 'Configure', 'Realm Settings', 'Clients' (which is highlighted with a blue bar), 'Client Templates', 'Roles', 'Identity Providers', 'User Federation', and 'Authentication'. The main content area has a breadcrumb path: 'Clients > myapp > Credentials > Client Certificate Import'. The title of the form is 'Import Client Certificate'. It includes fields for 'Archive Format' (set to 'JKS'), 'Key Alias' (empty), 'Store Password' (empty), and 'Import File' (with a 'Select file' button). At the bottom are 'Import' and 'Cancel' buttons.

There are multiple formats you can import from, just choose the archive format you have the certificate stored in, select the file, and click the **Import** button.

Finally note that you don't even need to import certificate if you choose to **Use JWKS URL**. In that case, you can provide the URL where client publishes its public key in **JWK** format. This is flexible because when client changes its keys, KeyCloak will automatically download them without need to re-import anything on KeyCloak side.

If you use client secured by KeyCloak adapter, you can configure the JWKS URL like [https://myhost.com/myapp/k\\_jwks](https://myhost.com/myapp/k_jwks) assuming that

<https://myhost.com/myapp> is the root URL of your client application. See [{developerguide!}](#) for additional details.

For the performance purposes, KeyCloak caches the public keys of the OIDC clients. If you think that private key of your client was compromised, it is obviously good to update your keys, but it's also good to clear the keys cache. See [Clearing the cache](#) section for more details.

### *Signed JWT with Client Secret*

If you select this option in the `Client Authenticator` list box, you can use a JWT signed by client secret instead of the private key.

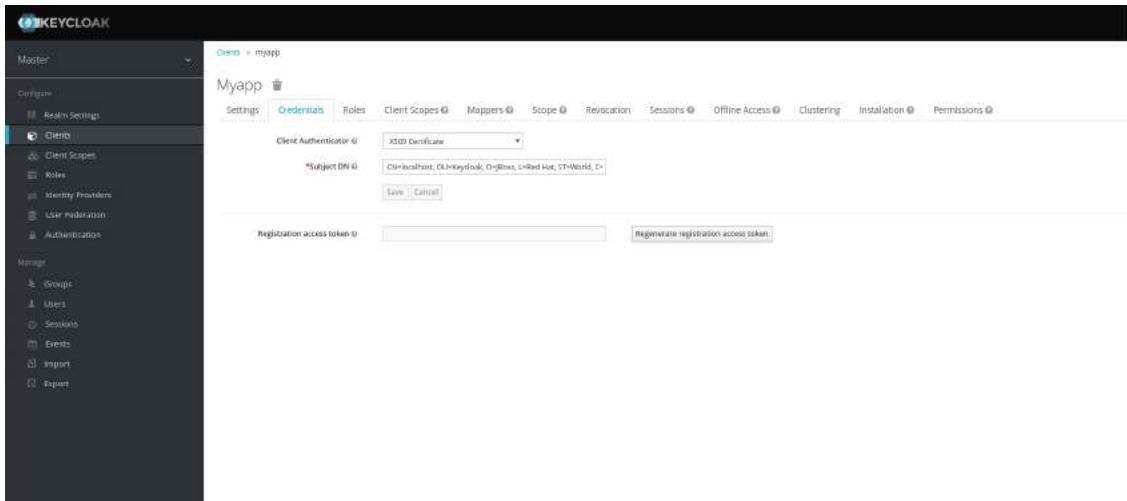
This client secret will be used to sign the JWT by the client.

### *X509 Certificate*

By enabling this option KeyCloak will validate if the client uses proper X509 certificate during the TLS Handshake.

This option requires mutual TLS in KeyCloak, see [Enable mutual SSL in WildFly](#).

### *X509 Certificate*



The validator checks also the certificate's Subject DN field with configured regexp validation expression. For some use cases, it is sufficient to accept all certificates. In that case, you can use `( . *?)(?:$)` expression.

There are two ways for KeyCloak to obtain the Client ID from the request. The first option is the `client_id` parameter in the query (described in Section 2.2 of the [OAuth 2.0 Specification](#)). The second option is to supply `client_id` as a query parameter.

### 8.1.3. Service Accounts

Each OIDC client has a built-in *service account* which allows it to obtain an access token. This is covered in the OAuth 2.0 specification under [Client Credentials Grant](#). To use this feature you must set the [Access Type](#) of your client to `confidential`. When you do this, the `Service Accounts Enabled` switch will appear. You need to turn on this switch. Also make sure that you have configured your [client credentials](#).

To use it you must have registered a valid `confidential` Client and you need to check the switch `Service Accounts Enabled` in KeyCloak admin console for this client. In tab `Service Account Roles`

you can configure the roles available to the service account retrieved on behalf of this client. Remember that you must have the roles available in Role Scope Mappings (tab `Scope`) of this client as well, unless you have `Full Scope Allowed` on. As in a normal login, roles from access token are the intersection of:

- Role scope mappings of particular client combined with the role scope mappings inherited from linked client scopes
- Service account roles

The REST URL to invoke on is `/auth/realms/{realm-name}/protocol/openid-connect/token`. Invoking on this URL is a POST request and requires you to post the client credentials. By default, client credentials are represented by `clientId` and `clientSecret` of the client in `Authorization: Basic` header, but you can also authenticate the client with a signed JWT assertion or any other custom mechanism for client authentication. You also need to use the parameter `grant_type=client_credentials` as per the OAuth2 specification.

For example the POST invocation to retrieve a service account can look like this:

```
POST /auth/realms/demo/protocol/openid-connect/token
Authorization: Basic cHJvZHVVjdc1zYS1jbGllbnQ6cGFzc3dvc-
mQ=
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials
```

The response would be this [standard JSON document](#) from the OAuth

2.0 specification.

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
    "access_token": "2YotnFZFEjr1zCsicMWpAA",
    "token_type": "bearer",
    "expires_in": 60,
    "refresh_token": "tGzv3J0kF0XG5Qx2T1KWIA",
    "refresh_expires_in": 600,
    "id_token": "tGzv3J0kF0XG5Qx2T1KWIA",
    "not-before-policy": 0,
    "session_state": "234234-234234-234234"
}
```

The retrieved access token can be refreshed or logged out by an out-of-bound request.

#### 8.1.4. Audience Support

The typical environment where the KeyCloak is deployed generally consists of a set of *confidential* or *public* client applications (frontend client applications) which use KeyCloak for authentication.

There are also *services* (called *Resource Servers* in the OAuth 2 specification), which serve requests from frontend client applications and provide resources. These services typically require an *Access token* (Bearer token) to be sent to them to authenticate for a particular request. This token was previously obtained by the frontend application when it tries to log in against KeyCloak.

In the environment where the trust among services is low, you may en-

counter this scenario:

1. A frontend client called `my-app` is required to be authenticated against KeyCloak.
2. A user is authenticated in KeyCloak. KeyCloak then issued tokens to the `my-app` application.
3. The application `my-app` used the token to invoke the service `evil-service`. The application needs to invoke `evil-service` as the service is able to serve some very useful data.
4. The `evil-service` application returned the response to `my-app`. However, at the same time, it kept the token previously sent to it.
5. The `evil-service` application then invoked another service called `good-service` with the previously kept token. The invocation was successful and `good-service` returned the data. This results in broken security as the `evil-service` misused the token to access other services on behalf of the client `my-app`.

This flow may not be an issue in many environments with the high level of trust among services. However in other environments, where the trust among services is lower, this can be problematic.

In some environments, this example work flow may be even requested behavior as the `evil-service` may need to retrieve additional data from `good-service` to be able to properly return the requested data to the original caller (`my-app` client). You may notice similarities with the Kerberos Credential Delegation. As with

the Kerberos Credential Delegation, an unlimited audience is a mixed blessing as it is only useful when a high level of trust exists among services. Otherwise, it is recommended to limit audience as described next.

You can limit audience and at the same time allow the `evil-service` to retrieve required data from the `good-service`. In this case, you need to ensure that both the `evil-service` and `good-service` are added as audiences to the token.

To prevent any misuse of the access token as in the example above, it is recommended to limit *Audience* on the token and configure your services to verify the audience on the token. If this is done, the flow above will change, like this:

1. A frontend client called `my-app` is required to be authenticated against KeyCloak.
2. A user is authenticated in KeyCloak. KeyCloak then issued tokens to the `my-app` application. The client application already knows that it will need to invoke service `evil-service`, so it used `scope=evil-service` in the authentication request sent to the KeyCloak server. See [Client Scopes section](#) for more details about the *scope* parameter. The token issued to the `my-app` client contains the audience, as in `"audience": [ "evil-service" ]`, which declares that the client wants to use this access token to invoke just the service `evil-service`.
3. The `evil-service` application served the request to the `my-app`. At the same time, it kept the token previously sent to it.

- The `evil-service` application then invoked the `good-service` with the previously kept token. Invocation was not successful because `good-service` checks the audience on the token and it sees that audience is only `evil-service`. This is expected behavior and security is not broken.

If the client wants to invoke the `good-service` later, it will need to obtain another token by issuing the SSO login with the `scope=good-service`. The returned token will then contain `good-service` as an audience:

```
"audience": [ "good-service" ]
```

and can be used to invoke `good-service`.

## Setup

To properly set up audience checking:

- Ensure that services are configured to check audience on the access token sent to them by adding the flag `verify-token-audience` in the adapter configuration. See [Adapter configuration](#) for details.
- Ensure that when an access token is issued by Keycloak, it contains all requested audiences and does not contain any audiences that are not needed. The audience can be either automatically added due the client roles as described in the [next section](#) or it can be hardcoded as described [below](#).

## Automatically add audience

In the default client scope *roles*, there is an *Audience Resolve* protocol

mapper defined. This protocol mapper will check all the clients for which current token has at least one client role available. Then the client ID of each of those clients will be added as an audience automatically. This is especially useful if your service (usually bearer-only) clients rely on client roles.

As an example, let us assume that you have a bearer-only client `good-service` and the confidential client `my-app`, which you want to authenticate and then use the access token issued for the `my-app` to invoke the `good-service` REST service. If the following are true:

- The `good-service` client has any client roles defined on itself
- Target user has at least one of those client roles assigned
- Client `my-app` has the role scope mappings for the assigned role

then the `good-service` will be automatically added as an audience to the access token issued for the `my-app`.

If you want to ensure that audience is not added automatically, do not configure role scope mappings directly on the `my-app` client, but instead create a dedicated client scope, for example called `good-service`, which will contain the role scope mappings for the client roles of the `good-service` client. Assuming that this client scope will be added as an optional client scope to the `my-app` client, the client roles and audience will be added to the token just if explicitly requested by the `scope=good-service` parameter.

The frontend client itself is not automatically added to the access token audience. This allows for easy differentiation between the access token and the ID token, because the access token will not contain the client for which the token was issued as an audience. So in the example above, the `my-app` won't be added as an audience. If you need the client itself as an audience, see the [hardcoded audience](#) option. However, using the same client as both frontend and REST service is not recommended.

## Hardcoded audience

For the case when your service relies on realm roles or does not rely on the roles in the token at all, it can be useful to use hardcoded audience. This is a protocol mapper, which will add client ID of the specified service client as an audience to the token. You can even use any custom value, for example some URL, if you want different audience than client ID.

You can add protocol mapper directly to the frontend client, however than the audience will be always added. If you want more fine-grain control, you can create protocol mapper on the dedicated client scope, which will be called for example `good-service`.

## *Audience Protocol Mapper*

The screenshot shows the Keycloak administration interface. On the left, there's a sidebar with navigation links like 'Master', 'Configure' (with 'Realm Settings', 'Clients', 'Client Scopes', 'Roles', 'Identity Providers', 'User Federation', 'Authentication'), and 'Manage' (with 'Groups', 'Users', 'Sessions', 'Events', 'Import', 'Export'). The 'Client Scopes' link is highlighted. The main content area has a breadcrumb path: 'Client Scopes > good-service > Mappers > Audience for good-service'. The title is 'Audience For Good-service'. The form fields are as follows:

- Protocol**: openid-connect
- ID**: 69beaf55-b0dc-49aa-986c-c10617808487
- Name**: Audience for good-service
- Mapper Type**: Audience
- Included Client Audience**: good-service
- Included Custom Audience**: (empty)
- Add to ID token**: OFF (disabled)
- Add to access token**: ON (selected)

At the bottom right are 'Save' and 'Cancel' buttons.

- From the [Installation tab](#) of the `good-service` client, you can generate the adapter configuration and you can confirm that `verify-to-ken-audience` option will be set to true. This indicates that the adapter will require verifying the audience if you use this generated configuration.
- Finally, you need to ensure that the `my-app` frontend client is able to request `good-service` as an audience in its tokens. On the `my-app` client, click the *Client Scopes* tab. Then assign `good-service` as an optional (or default) client scope. See [Client Scopes Linking section](#) for more details.
- You can optionally [Evaluate Client Scopes](#) and generate an example access token. If you do, notice that `good-service` will be added to the audience of the generated access token only if `good-service` is included in the `scope` parameter in the case you assigned it as an optional client scope.
- In your `my-app` application, you must ensure that `scope` parameter is used with the value `good-service` always included when you

want to issue the token for accessing the `good-service`. See the [parameters forwarding section](#), if your application uses the servlet adapter, or the [javascript adapter section](#), if your application uses the javascript adapter.

If you are unsure what the correct audience and roles in the token will be, it is always a good idea to [Evaluate Client Scopes](#) in the admin console and do some testing around it.

Both the *Audience* and *Audience Resolve* protocol mappers add the audiences just to the access token by default. The ID Token typically contains only single audience, which is the client ID of the client for which the token was issued. This is a requirement of the OpenID Connect specification. On the other hand, the access token does not necessarily have the client ID of the client, which was the token issued for, unless any of the audience mappers added it.

## 8.2. SAML Clients

KeyCloak supports [SAML 2.0](#) for registered applications. Both POST and Redirect bindings are supported. You can choose to require client signature validation and can have the server sign and/or encrypt responses as well.

To create a SAML client go to the `Clients` left menu item. On this page you'll see a `Create` button on the right.

## Clients

The screenshot shows the Keycloak administration interface under the 'Demo' realm. The left sidebar has 'Configure' and 'Manage' sections. Under 'Configure', 'Clients' is selected. The main area is titled 'Clients' with a search bar and a 'Create' button. A table lists six clients:

Client ID	Enabled	Base URL	Actions		
account	True	/auth/realms/demo/account	Edit	Export	Delete
admin-cli	True	Not defined	Edit	Export	Delete
broker	True	Not defined	Edit	Export	Delete
realm-management	True	Not defined	Edit	Export	Delete
security-admin-console	True	/auth/admin/demo/console/index.html	Edit	Export	Delete

This will bring you to the `Add Client` page.

## Add Client

The screenshot shows the 'Add Client' page in the Keycloak administration interface under the 'Master' realm. The left sidebar has 'Configure' and 'Manage' sections. Under 'Configure', 'Clients' is selected. The main area is titled 'Add Client'. It includes fields for 'Import' (with a 'Select file' button), 'Client ID' (set to 'myapp'), 'Client Protocol' (set to 'saml'), and 'Client SAML Endpoint' (set to 'http://localhost:8080/myapp/saml'). At the bottom are 'Save' and 'Cancel' buttons.

Enter in the `Client ID` of the client. This is often a URL and will be the expected `issuer` value in SAML requests sent by the application. Next select `saml` in the `Client Protocol` drop down box. Finally enter in the `Client SAML Endpoint` URL. Enter the URL you want the KeyCloak server to send SAML requests and responses to. Usually applications have only one URL for processing SAML requests. If your application has different URLs for its bindings, don't worry, you can fix this in the `Settings` tab of the client. Click `Save`. This will create

the client and bring you to the client **Settings** tab.

## Client Settings

The screenshot shows the Keycloak administration interface. On the left, a sidebar menu is open under the 'Master' realm, showing options like 'Configure' (selected), 'Clients' (highlighted with a blue border), 'Realm Settings', 'Client Templates', 'Roles', 'Identity Providers', 'User Federation', and 'Authentication'. The main content area is titled 'Clients > myapp' and shows the 'Myapp' client configuration. The 'Settings' tab is selected. The configuration fields include:

- Client ID:** myapp
- Name:** (empty input field)
- Description:** (empty input field)
- Enabled:** ON (switch is blue)
- Consent Required:** OFF (switch is white)
- Client Protocol:** saml
- Client Template:** (dropdown menu)
- Include AuthnStatement:** ON (switch is blue)
- Sign Documents:** ON (switch is blue)
- Optimize REDIRECT signing key lookup:** OFF (switch is white)
- Sign Assertions:** OFF (switch is white)
- Signature Algorithm:** RSA\_SHA256
- SAML Signature Key Name:** KEY\_ID
- Canonicalization Method:** EXCLUSIVE
- Encrypt Assertions:** OFF (switch is white)

## Client ID

This value must match the issuer value sent with AuthNRequests. KeyCloak will pull the issuer from the Authn SAML request and match it to a client by this value.

## Name

This is the display name for the client whenever it is displayed in a KeyCloak UI screen. You can localize the value of this field by setting up a replacement string value i.e. \${myapp}. See the [{developerguide!}](#) for more information.

## Description

This specifies the description of the client. This can also be localized.

## **Enabled**

If this is turned off, the client will not be allowed to request authentication.

## **Consent Required**

If this is on, then users will get a consent page which asks the user if they grant access to that application. It will also display the metadata that the client is interested in so that the user knows exactly what information the client is getting access to. If you've ever done a social login to Google, you'll often see a similar page. Keycloak provides the same functionality.

## **Include AuthnStatement**

SAML login responses may specify the authentication method used (password, etc.) as well as a timestamp of the login. Setting this to on will include that statement in the response document.

## **Sign Documents**

When turned on, Keycloak will sign the document using the realm's private key.

## **Optimize REDIRECT signing key lookup**

When turned on, the SAML protocol messages will include Keycloak native extension that contains a hint with signing key ID. When the SP understands this extension, it can use it for signature validation instead of attempting to validate signature with all known keys.

This option only applies to REDIRECT bindings where the signature

is transferred in query parameters where there is no place with this information in the signature information (contrary to POST binding messages where key ID is always included in document signature). Currently this is relevant to situations where both IDP and SP are provided by Keycloak server and adapter. This option is only relevant when `Sign Documents` is switched on.

## Sign Assertions

The `Sign Documents` switch signs the whole document. With this setting the assertion is also signed and embedded within the SAML XML Auth response.

## Signature Algorithm

Choose between a variety of algorithms for signing SAML documents.

## SAML Signature Key Name

Signed SAML documents sent via POST binding contain identification of signing key in `KeyName` element. This by default contains Keycloak key ID. However various vendors might expect a different key name or no key name at all. This switch controls whether `KeyName` contains key ID (option `KEY_ID`), subject from certificate corresponding to the realm key (option `CERT_SUBJECT` - expected for instance by Microsoft Active Directory Federation Services), or that the key name hint is completely omitted from the SAML message (option `NONE`).

## Canonicalization Method

Canonicalization method for XML signatures.

## **Encrypt Assertions**

Encrypt assertions in SAML documents with the realm's private key. The AES algorithm is used with a key size of 128 bits.

## **Client Signature Required**

Expect that documents coming from a client are signed. KeyCloak will validate this signature using the client public key or cert set up in the `SAML Keys` tab.

## **Force POST Binding**

By default, KeyCloak will respond using the initial SAML binding of the original request. By turning on this switch, you will force KeyCloak to always respond using the SAML POST Binding even if the original request was the Redirect binding.

## **Front Channel Logout**

If true, this application requires a browser redirect to be able to perform a logout. For example, the application may require a cookie to be reset which could only be done via a redirect. If this switch is false, then KeyCloak will invoke a background SAML request to logout the application.

## **Force Name ID Format**

If the request has a name ID policy, ignore it and used the value configured in the admin console under Name ID Format

## **Name ID Format**

Name ID Format for the subject. If no name ID policy is specified in the request or if the Force Name ID Format attribute is true, this

value is used. Properties used for each of the respective formats are defined below.

### **Root URL**

If KeyCloak uses any configured relative URLs, this value is prepended to them.

### **Valid Redirect URIs**

This is an optional field. Enter in a URL pattern and click the + sign to add. Click the - sign next to URLs you want to remove. Remember that you still have to click the `Save` button! Wildcards (\*) are only allowed at the end of a URI, i.e. `http://host.com/*`. This field is used when the exact SAML endpoints are not registered and KeyCloak is pulling the Assertion Consumer URL from the request.

### **Base URL**

If KeyCloak needs to link to the client, this URL would be used.

### **Master SAML Processing URL**

This URL will be used for all SAML requests and the response will be directed to the SP. It will be used as the Assertion Consumer Service URL and the Single Logout Service URL. If a login request contains the Assertion Consumer Service URL, that will take precedence, but this URL must be validated by a registered Valid Redirect URI pattern

### **Assertion Consumer Service POST Binding URL**

POST Binding URL for the Assertion Consumer Service.

## **Assertion Consumer Service Redirect Binding URL**

Redirect Binding URL for the Assertion Consumer Service.

## **Logout Service POST Binding URL**

POST Binding URL for the Logout Service.

## **Logout Service Redirect Binding URL**

Redirect Binding URL for the Logout Service.

### **8.2.1. IDP Initiated Login**

IDP Initiated Login is a feature that allows you to set up an endpoint on the Keycloak server that will log you into a specific application/client.

In the `Settings` tab for your client, you need to specify the `IDP Initiated SSO URL Name`. This is a simple string with no whitespace in it. After this you can reference your client at the following URL:  
`root/auth/realms/{realm}/protocol/saml/clients/{url-name}`

The IDP initiated login implementation prefers *POST* over *REDIRECT* binding (check [saml bindings](#) for more information). Therefore the final binding and SP URL are selected in the following way:

1. If the specific `Assertion Consumer Service POST Binding URL` is defined (inside `Fine Grain SAML Endpoint Configuration` section of the client settings) *POST* binding is used through that URL.
2. If the general `Master SAML Processing URL` is specified then *POST* binding is used again through this general URL.
3. As the last resort, if the `Assertion Consumer Service Redi-`

rect Binding URL is configured (inside Fine Grain SAML Endpoint Configuration) *REDIRECT* binding is used with this URL.

If your client requires a special relay state, you can also configure this on the Settings tab in the IDP Initiated SSO Relay State field. Alternatively, browsers can specify the relay state in a Relay-State query parameter, i.e. root/auth/realms/{realm}/protocol/saml/clients/{url-name}?RelayState=thestate .

When using [identity brokering](#), it is possible to set up an IDP Initiated Login for a client from an external IDP. The actual client is set up for IDP Initiated Login at broker IDP as described above. The external IDP has to set up the client for application IDP Initiated Login that will point to a special URL pointing to the broker and representing IDP Initiated Login endpoint for a selected client at the brokering IDP. This means that in client settings at the external IDP:

- IDP Initiated SSO URL Name is set to a name that will be published as IDP Initiated Login initial point,
- Assertion Consumer Service POST Binding URL in the Fine Grain SAML Endpoint Configuration section has to be set to the following URL: broker-root/auth/realms/{broker-realm}/broker/{idp-name}/endpoint/clients/{client-id} , where:
  - broker-root is base broker URL
  - broker-realm is name of the realm at broker where external IDP is declared

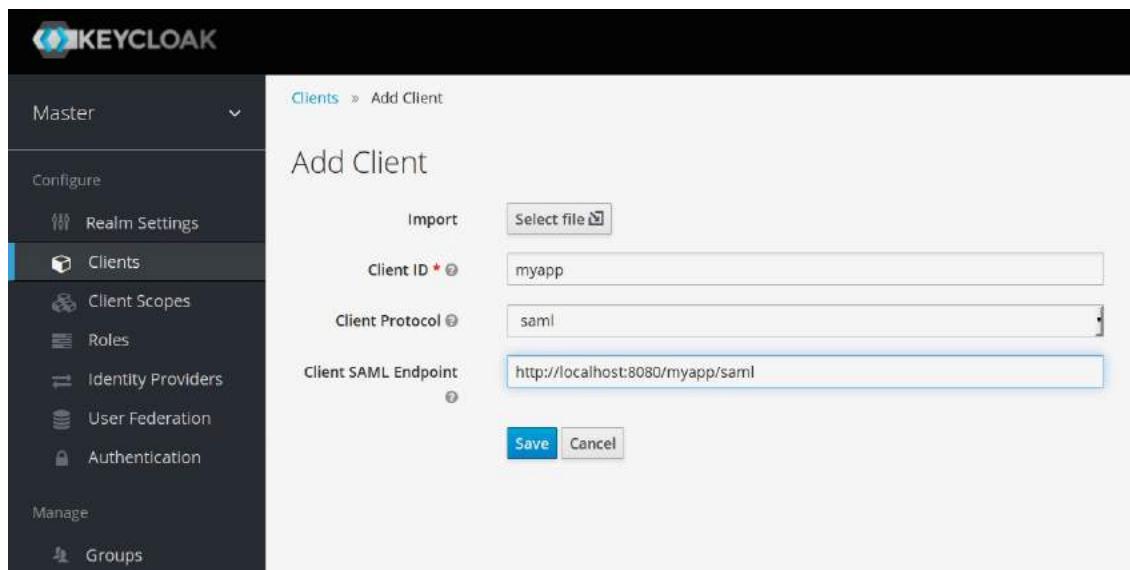
- *idp-name* is name of the external IDP at broker
- *client-id* is the value of `IDP Initiated SSO URL Name` attribute of the SAML client defined at broker. It is this client, which will be made available for IDP Initiated Login from the external IDP.

Please note that you can import basic client settings from the brokering IDP into client settings of the external IDP - just use [SP Descriptor](#) available from the settings of the identity provider in the brokering IDP, and add `clients/client-id` to the endpoint URL.

### 8.2.2. SAML Entity Descriptors

Instead of manually registering a SAML 2.0 client, you can import it via a standard SAML Entity Descriptor XML file. There is an `Import` option on the Add Client page.

#### Add Client



The screenshot shows the Keycloak 'Add Client' interface. On the left, there's a sidebar with 'Master' selected under 'Realm Settings'. The main area has 'Clients' selected under 'Configure'. The title bar says 'Clients > Add Client'. The 'Add Client' form is displayed with the following fields:

- Import:** A 'Select file' button.
- Client ID \***: The value 'myapp' is entered.
- Client Protocol**: The value 'saml' is selected from a dropdown.
- Client SAML Endpoint**: The value 'http://localhost:8080/myapp/saml' is entered.
- Buttons:** 'Save' and 'Cancel'.

Click the `Select File` button and load your entity descriptor file.  
You should review all the information there to make sure everything is

set up correctly.

Some SAML client adapters like *mod-auth-mellon* need the XML Entity Descriptor for the IDP. You can obtain this by going to this public URL: `root/auth/realms/{realm}/protocol/saml/descriptor`

## 8.3. Client Links

For scenarios where one wants to link from one client to another, KeyCloak provides a special redirect endpoint: `/realms/realm_name/clients/{client-id}/redirect`.

If a client accesses this endpoint via an `HTTP GET` request, KeyCloak returns the configured base URL for the provided Client and Realm in the form of an `HTTP 307` (Temporary Redirect) via the response's `Location` header.

Thus, a client only needs to know the Realm name and the Client ID in order to link to them. This indirection helps avoid hard-coding client base URLs.

As an example, given the realm `master` and the client-id `account`:

```
http://host:port/auth/realms/master/clients/account/redirect
```

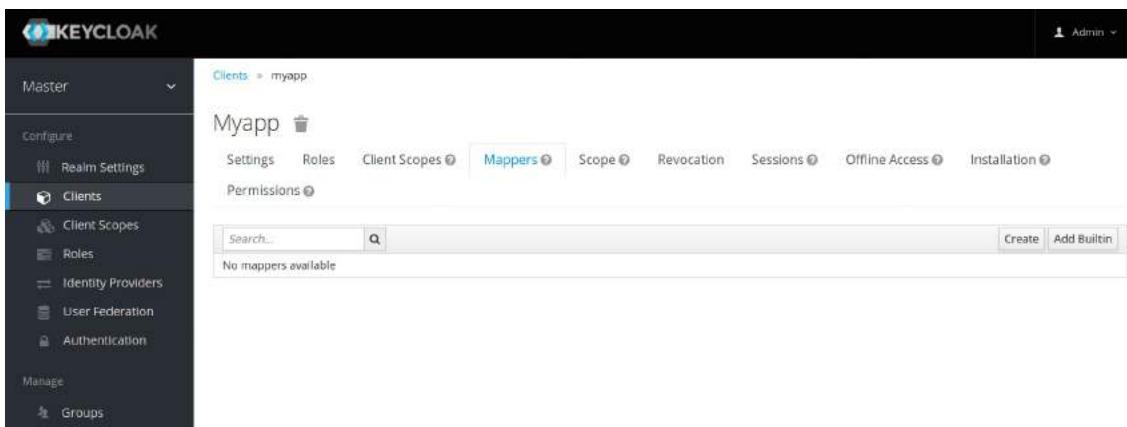
Would temporarily redirect to: <http://host:port/auth/realms/master/account>

## 8.4. OIDC Token and SAML Assertion Mappings

Applications that receive ID Tokens, Access Tokens, or SAML assertions may need or want different user metadata and roles. Keycloak allows you to define what exactly is transferred. You can hardcode roles, claims and custom attributes. You can pull user metadata into a token or assertion. You can rename roles. Basically you have a lot of control of what exactly goes back to the client.

Within the Admin Console, if you go to an application you've registered, you'll see a `Mappers` tab. Here's one for an OIDC based client.

### *Mappers Tab*



The new client does not have any built-in mappers, however it usually inherits some mappers from the client scopes as described in the [client scopes section](#). Protocol mappers map things like, for example, email address to a specific claim in the identity and access token. Their function should each be self explanatory from their name. There are additional pre-configured mappers that are not attached to the client that you can add by clicking the `Add Builtin` button.

Each mapper has common settings as well as additional ones depending on which type of mapper you are adding. Click the `Edit` button next to one of the mappers in the list to get to the config screen.

## Mapper Config

The screenshot shows the Keycloak configuration interface for a 'Mapper'. The left sidebar has 'Master' selected under 'Configure' and lists 'Clients', 'Client Templates', 'Roles', 'Identity Providers', 'User Federation', and 'Authentication'. Under 'Manage', it lists 'Groups', 'Users', 'Sessions', 'Events', and 'Import'. The main panel is titled 'Email' and shows the configuration for an 'Email' mapper. The fields are as follows:

- Protocol: openid-connect
- ID: 564bb27e-a21f-4e99-87fc-cf1252977798
- Name: email
- Consent Required: ON
- Consent Text: \${email}
- Mapper Type: User Property
- Property: email
- Token Claim Name: email
- Claim JSON Type: String
- Add to ID token: ON
- Add to access token: ON
- Add to userinfo: ON

The best way to learn about a config option is to hover over its tooltip.

Most OIDC mappers also allow you to control where the claim gets put. You can opt to include or exclude the claim from both the *id* and *access* tokens by fiddling with the `Add to ID token` and `Add to access token` switches.

Finally, you can also add other mapper types. If you go back to the `Mappers` tab, click the `Create` button.

## Add Mapper

The screenshot shows the Keycloak admin interface for creating a protocol mapper. The left sidebar is titled 'Master' and includes sections for 'Configure' (Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication), 'Manage' (Groups, Users, Sessions, Events, Import), and a 'Clients' section which is currently selected. The main content area is titled 'Create Protocol Mapper' and shows the configuration for an 'openid-connect' protocol. It includes fields for 'Name' (empty), 'Consent Required' (OFF), 'Mapper Type' (set to 'Group Membership'), 'Token Claim Name' (empty), and several optional checkboxes: 'Full group path' (OFF), 'Add to ID token' (OFF), 'Add to access token' (OFF), and 'Add to userinfo' (OFF). At the bottom are 'Save' and 'Cancel' buttons.

Pick a `Mapper Type` from the list box. If you hover over the tooltip, you'll see a description of what that mapper type does. Different config parameters will appear for different mapper types.

#### 8.4.1. Priority order

Mapper implementations have *priority order*. This priority order is not the configuration property of the mapper; rather, it is the property of the concrete implementation of the mapper.

Mappers are sorted in the admin console by the order in the list of mappers and the changes in the token or assertion will be applied using that order with the lowest being applied first. This means that implementations which are dependent on other implementations are processed in the needed order.

For example, when we first want to compute the roles which will be in-

cluded with a token, we first resolve audiences based on those roles. Then, we process a JavaScript script that uses the roles and audiences already available in the token.

#### 8.4.2. OIDC User Session Note Mappers

User session details are via mappers and depend on various criteria. User session details are automatically included when you use or enable a feature on a client. You can also click the `Add builtin` button to include session details.

Impersonated user sessions provide the following details:

- `IMPERSONATOR_ID` : The ID of an impersonating user
- `IMPERSONATOR_USERNAME` : The username of an impersonating user

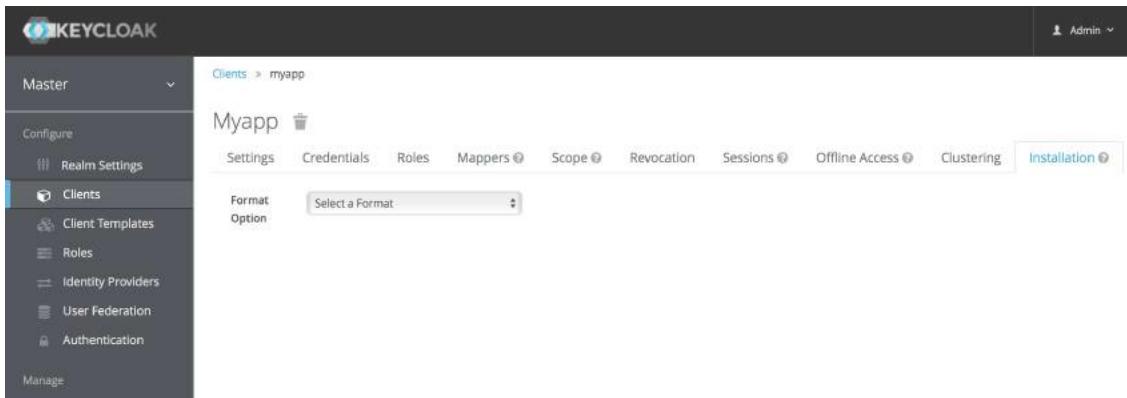
Service account sessions provide the following details:

- `clientId` : The client ID of the service account
- `clientAddress` : The remote host IP of the service account authenticated device
- `clientHost` : The remote host name of the service account authenticated device

### 8.5. Generating Client Adapter Config

The Keycloak can pre-generate configuration files that you can use to install a client adapter for in your application's deployment environment. A number of adapter types are supported for both OIDC and

SAML. Go to the `Installation` tab of the client you want to generate configuration for.

A screenshot of the Keycloak administration interface. The top navigation bar shows 'KEYCLOAK' and 'Admin'. The left sidebar is titled 'Master' and contains sections for 'Configure' (Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication), 'Manage', and 'Logout'. Under 'Clients', 'myapp' is selected. The main content area shows 'Myapp' with tabs for Settings, Credentials, Roles, Mappers, Scope, Revocation, Sessions, Offline Access, Clustering, and Installation. The 'Installation' tab is active. Below it is a 'Format Option' dropdown with the placeholder 'Select a Format'.

Select the `Format Option` you want configuration generated for. All KeyCloak client adapters for OIDC and SAML are supported. The mod-auth-mellon Apache HTTPD adapter for SAML is supported as well as standard SAML entity descriptor files.

## 8.6. Client Scopes

If you have many applications you need to secure and register within your organization, it can become tedious to configure the [protocol mappers](#) and [role scope mappings](#) for each of these clients. KeyCloak allows you to define a shared client configuration in an entity called a *client scope*.

Client scopes also provide support for the OAuth 2 `scope` parameter, which allows a client application to request more or fewer claims or roles in the access token, according to the application needs.

To create a client scope, follow these steps:

- Go to the `Client Scopes` left menu item. This initial screen

shows you a list of currently defined client scopes.

## Client Scopes List

Name	Protocol	Actions	
address	openid-connect	Edit	Delete
email	openid-connect	Edit	Delete
offline_access	openid-connect	Edit	Delete
phone	openid-connect	Edit	Delete
profile	openid-connect	Edit	Delete
role_list	saml	Edit	Delete

- Click the `Create` button. Name the client scope and save. A *client scope* will have similar tabs to a regular clients. You can define [protocol mappers](#) and [role scope mappings](#), which can be inherited by other clients, and which are configured to inherit from this client scope.

### 8.6.1. Protocol

When you are creating the client scope, you must choose the `Protocol`. Only the clients which use same protocol can then be linked with this client scope.

Once you have created new realm, you can see that there is a list of pre-defined (builtin) client scopes in the menu.

- For the SAML protocol, there is one builtin client scope, `roles-list`, which contains one protocol mapper for showing the roles list in the SAML assertion.
- For the OpenID Connect protocol, there are client scopes `profile`,

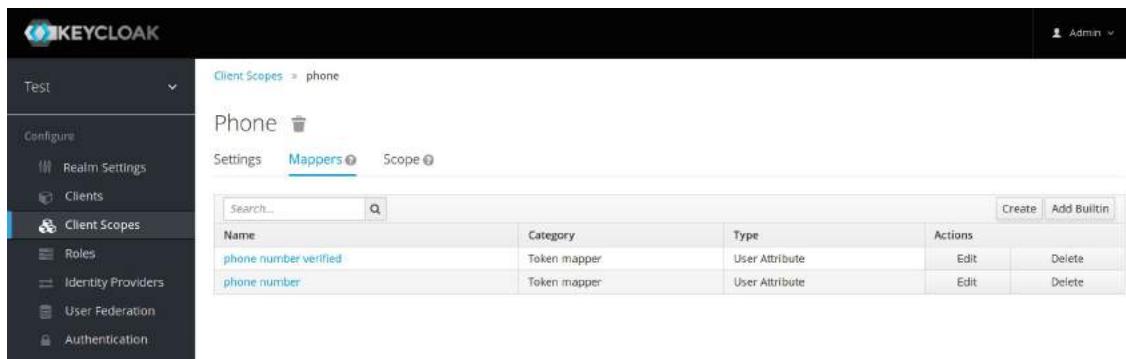
```
le, email, address, phone, offline_access, roles,  
web-origins and microprofile-jwt.
```

The client scope, `offline_access`, is useful when client wants to obtain offline tokens. Learn about offline tokens in the [Offline Access section](#) or in the [OpenID Connect specification](#), where scope parameter is defined with the value `offline_access`.

The client scopes `profile`, `email`, `address` and `phone` are also defined in the [OpenID Connect specification](#). These client scopes do not have any role scope mappings defined, but they have some protocol mappers defined, and these mappers correspond to the claims defined in the OpenID Connect specification.

For example, when you click to open the `phone` client scope and open the `Mappers` tab, you will see the protocol mappers, which correspond to the claims defined in the specification for the scope `phone`.

## *Client Scope Mappers*



The screenshot shows the Keycloak administration interface. The left sidebar is titled 'Test' and includes 'Configure' with sub-options: Realm Settings, Clients, Client Scopes (which is selected), Roles, Identity Providers, User Federation, and Authentication. The main content area has a header 'Client Scopes > phone'. Below the header, there are tabs: 'Settings' (selected), 'Mappers' (which is highlighted in blue), and 'Scope'. A search bar and a 'Create' button are at the top of the table. The table has columns: Name, Category, Type, and Actions (Edit and Delete). There are two rows: 'phone number verified' (Category: Token mapper, Type: User Attribute) and 'phone number' (Category: Token mapper, Type: User Attribute).

Name	Category	Type	Actions
phone number verified	Token mapper	User Attribute	Edit Delete
phone number	Token mapper	User Attribute	Edit Delete

When the `phone` client scope is linked to a client, that client automatically inherits all the protocol mappers defined in the `phone` client scope. Access tokens issued for this client will contain the phone number information about the user, assuming that the user has a defined

phone number.

Builtin client scopes contain exactly the protocol mappers as defined per the specification, however you are free to edit client scopes and create/update/remove any protocol mappers (or role scope mappings).

The client scope `roles` is not defined in the OpenID Connect specification and it is also not added automatically to the `scope` claim in the access token. This client scope has some mappers, which are used to add roles of the user to the access token and possibly add some audiences for the clients with at least one client role as described in the [Audience section](#).

The client scope `web-origins` is also not defined in the OpenID Connect specification and not added to the `scope` claim. This is used to add allowed web origins to the access token `allowed-origins` claim.

The client scope `microprofile-jwt` was created to handle the claims defined in the [MicroProfile/JWT Auth Specification](#). This client scope defines a user property mapper for the `upn` claim and also a realm role mapper for the `groups` claim. These mappers can be changed as needed so that different properties can be used to create the MicroProfile/JWT specific claims.

### 8.6.2. Consent related settings

Client scope contains options related to the consent screen. Those options are useful only if the linked client is configured to require consent (if the `Consent Required` switch is enabled on the client).

#### Display On Consent Screen

If on, and if this client scope is added to a client with consent required, then the text specified by `Consent Screen Text` will be displayed on the consent screen, which is shown once the user is authenticated and right before he is redirected from Keycloak to the client. If the switch is off, then this client scope will not be displayed on the consent screen.

### **Consent Screen Text**

The text shown on the consent screen when this client scope is added to some client with consent required defaults to the name of client scope. The value for this text is localizable by specifying a substitution variable with  `${var-name}` strings. The localized value is then configured within property files in your theme. See the [{developer-guide!}](#) for more information on localization.

#### **8.6.3. Link Client Scope with the Client**

Linking between client scope and client is configured in the `Client Scopes` tab of the particular client. There are 2 ways of linking between client scope and client.

### **Default Client Scopes**

This is applicable for both OpenID Connect and SAML clients. Default client scopes are always applied when issuing OpenID Connect tokens or SAML assertions for this client. The client will inherit Protocol mappers and Role Scope Mappings defined on the client scope. For the OpenID Connect Protocol, the Mappers and Role Scope Mappings are always applied, regardless of the value used for the scope parameter in the OpenID Connect authorization request.

## Optional Client Scopes

This is applicable only for OpenID Connect clients. Optional client scopes are applied when issuing tokens for this client, but only when they are requested by the `scope` parameter in the OpenID Connect authorization request.

### Example

For this example, we assume that the client has `profile` and `email` linked as default client scopes, and `phone` and `address` are linked as optional client scopes. The client will use the value of the scope parameter when sending a request to the OpenID Connect authorization endpoint:

```
scope=openid phone
```

The scope parameter contains the string, with the scope values divided by space (which is also the reason why a client scope name cannot contain a space character in it). The value `openid` is the meta-value used for all OpenID Connect requests, so we will ignore it for this example. The token will contain mappers and role scope mappings from the client scopes `profile`, `email` (which are default scopes) and `phone` (an optional client scope requested by the scope parameter).

#### 8.6.4. Evaluating Client Scopes

The tabs `Mappers` and `Scope` of the client contain the protocol mappers and role scope mappings declared solely for this client. They do not contain the mappers and scope mappings inherited from client scopes. However, it may be useful to see what the effective protocol mappers will be (protocol mappers defined on the client itself as well as

inherited from the linked client scopes) and the effective role scope mappings used when you generate the token for the particular client.

You can see all of these when you click the `Client Scopes` tab for the client and then open the sub-tab `Evaluate`. From here you can select the optional client scopes that you want to apply. This will also show you the value of the `scope` parameter, which needs to be sent from the application to the Keycloak OpenID Connect authorization endpoint.

### Evaluating Client Scopes

The screenshot shows the Keycloak Admin UI for a client. The left sidebar has sections for Configure, Realm Settings, Clients, Client Scopes (which is selected), Roles, Identity Providers, User Federation, and Authentication. Under Manage, there are Groups, Users, Sessions, Events, Import, and Export. The main content area has tabs for Settings, Credentials, Roles, Client Scopes (selected), Mappers, Scope, Revocation, Sessions, and Offline Access. The Client Scopes tab has sub-tabs for Setup and Evaluate (selected). The Evaluate tab has fields for Scope Parameter (set to openid phone), Client Scopes (Available Optional Client Scopes: address, offline\_access; Selected Optional Client Scopes: phone), User (john), and an Evaluate button. Below these are tabs for Effective Protocol Mappers, Effective Role Scope Mappings, and Generated Access Token (selected). The Generated Access Token section displays a JSON token object:

```
{ "jti": "71c42bb1-e6c6-4a78-bc60-dc899bf23ddd", "exp": 1528095076, "nbf": 0, "id": 1528094776, "iss": "http://localhost:8081/auth/realm/test", "aud": "account", }
```

If you want to see how you can send a custom value for a `scope` parameter from your application, see the [parameters forwarding section](#), if your application uses the servlet adapter, or the [javascript adapter section](#), if your application uses the javascript adapter.

### Generating Example Tokens

To see an example of a real access token, generated for the particular user and issued for the particular client, with the specified value of `scope` parameter, select the user from the `Evaluate` screen. This will generate an example token that includes all of the claims and role mappings used.

### 8.6.5. Client Scopes Permissions

When issuing tokens for a particular user, the client scope is applied only if the user is permitted to use it. In the case that a client scope does not have any role scope mappings defined on itself, then each user is automatically permitted to use this client scope. However, when a client scope has any role scope mappings defined on itself, then the user must be a member of at least one of the roles. In other words, there must be an intersection between the user roles and the roles of the client scope. Composite roles are taken into account when evaluating this intersection.

If a user is not permitted to use the client scope, then no protocol mappers or role scope mappings will be used when generating tokens and the client scope will not appear in the `scope` value in the token.

### 8.6.6. Realm Default Client Scopes

The `Realm Default Client Scopes` allow you to define set of client scopes, which will be automatically linked to newly created clients.

Open the left menu item `Client Scopes` and then select `Default Client Scopes`.

From here, select the client scopes that you want to add as `Default Client Scopes` to newly created clients and `Optional Client`

Scopes to newly created clients.

## Default Client Scopes

The screenshot shows the Keycloak administration interface under the 'Test' realm. On the left, a sidebar menu includes 'Configure' (Realm Settings, Clients, Client Scopes, Roles, Identity Providers, User Federation, Authentication) and 'Manage' (Groups, Users). The main content area is titled 'Default Client Scopes'. It features two tabs: 'Client Scopes' (disabled) and 'Default Client Scopes' (selected). Below the tabs are two sections: 'Default Client Scopes' and 'Optional Client Scopes'. Each section has an 'Available Client Scopes' list (empty in both cases) and an 'Assigned Client Scopes' list. In the 'Assigned Default Client Scopes' list, there are three items: 'role\_list', 'email', and 'profile'. Below each assigned list is a 'Remove selected' button.

Once the client is created, you can unlink the default client scopes, if needed. This is similar to how you remove [Default Roles](#).

### 8.6.7. Scopes explained

The term `scope` is used in KeyCloak on few places. Various occurrences of scopes are related to each other, but may have a different context and meaning. To clarify, here we explain the various `scopes` used in KeyCloak.

#### Client scope

Referenced in this chapter. Client scopes are entities in KeyCloak, which are configured at the realm level and they can be linked to clients. The client scopes are referenced by their name when a request is sent to the KeyCloak authorization endpoint with a corresponding value of the `scope` parameter. The details are described in the [section about client scopes linking](#).

## **Role scope mapping**

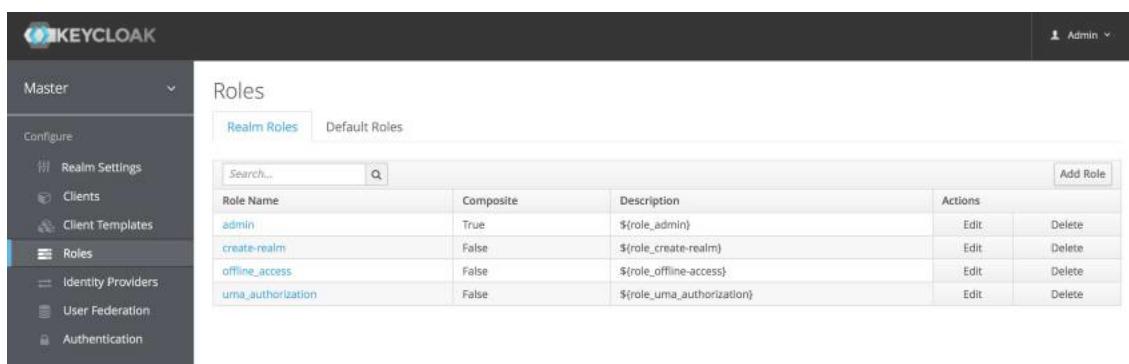
This can be seen when you open tab `Scope` of a client or client scope. Role scope mapping allows you to limit the roles which can be used in the access tokens. The details are described in the [Role Scope Mappings section](#).

# 9. Roles

Roles identify a type or category of user. `Admin`, `user`, `manager`, and `employee` are all typical roles that may exist in an organization. Applications often assign access and permissions to specific roles rather than individual users as dealing with users can be too fine grained and hard to manage. For example, the Admin Console has specific roles which give permission to users to access parts of the Admin Console UI and perform certain actions. There is a global namespace for roles and each client also has its own dedicated namespace where roles can be defined.

## 9.1. Realm Roles

Realm-level roles are a global namespace to define your roles. You can see the list of built-in and created roles by clicking the `Roles` left menu item.



The screenshot shows the Keycloak Admin Console interface. The top navigation bar includes the Keycloak logo, a search bar, and a user dropdown labeled "Admin". The left sidebar is titled "Master" and contains the following navigation items: "Configure" (with "Realm Settings", "Clients", "Client Templates", and "Roles" selected), "Identity Providers", "User Federation", and "Authentication". The main content area is titled "Roles" and has two tabs: "Realm Roles" (selected) and "Default Roles". Below the tabs is a search bar with placeholder text "Search...". A table lists the available roles:

Role Name	Composite	Description	Actions
admin	True	\${role_admin}	Edit Delete
create-realm	False	\${role_create-realm}	Edit Delete
offline_access	False	\${role_offline-access}	Edit Delete
uma_authorization	False	\${role_uma_authorization}	Edit Delete

To create a role, click **Add Role** on this page, enter in the name and description of the role, and click **Save**.

*Add Role*

The screenshot shows the Keycloak administration interface. On the left, a dark sidebar menu includes 'Master' (selected), 'Configure' (with 'Realm Settings', 'Clients', 'Client Scopes', and 'Roles' selected), 'Identity Providers', 'User Federation', and 'Authentication'. Under 'Manage', there are 'Groups', 'Users', and 'Sessions'. The main content area is titled 'Developer' and shows the 'developer' role configuration. It has tabs for 'Details', 'Permissions', and 'Users in Role'. The 'Details' tab shows 'Role Name' as 'developer', 'Description' as an empty text area, and 'Composite Roles' as 'OFF'. At the bottom are 'Save' and 'Cancel' buttons.

The value for the `description` field is localizable by specifying a substitution variable with  `${var-name}`  strings. The localized value is then configured within property files in your theme. See the [{developer-guide!}](#) for more information on localization.

## 9.2. Client Roles

Client roles are basically a namespace dedicated to a client. Each client gets its own namespace. Client roles are managed under the `Roles` tab under each individual client. You interact with this UI the same way you do for realm-level roles.

## 9.3. Composite Roles

Any realm or client level role can be turned into a *composite role*. A *composite role* is a role that has one or more additional roles associated with it. When a composite role is mapped to the user, the user also gains the roles associated with that composite. This inheritance is recursive so any composite of composites also gets inherited.

To turn a regular role into a composite role, go to the role detail page and flip the `Composite Role` switch on.

## Composite Role

The screenshot shows the 'Developer' role configuration page in Keycloak. The left sidebar is dark with white text, showing 'Master' at the top, followed by 'Configure' with sub-options 'Realm Settings', 'Clients', and 'Client Templates'. Below 'Configure' is a blue-highlighted 'Roles' section with sub-options 'Identity Providers', 'User Federation', and 'Authentication'. Under 'Manage' are 'Groups', 'Users', 'Sessions', 'Events', and 'Import'. The main content area has a header 'Developer' with a trash icon. Below it is a 'role' section with fields: 'Role Name' (developer), 'Description' (empty), 'Scope Param Required' (OFF), and 'Composite Roles' (ON). At the bottom are 'Save' and 'Cancel' buttons. A section titled 'Composite Roles' is expanded, showing 'Realm Roles' (admin, create-realm, offline\_access, uma\_authorization) and 'Associated Roles' (employee). There are 'Add selected' and 'Remove selected' buttons between the two lists. A 'Client Roles' dropdown is at the bottom left, and a note 'Select client to view roles for client' is at the bottom right.

Once you flip this switch the role selection UI will be displayed lower on the page and you'll be able to associate realm level and client level roles to the composite you are creating. In this example, the `employee` realm-level role was associated with the `developer` composite role. Any user with the `developer` role will now also inherit the `employee` role too.

When tokens and SAML assertions are created, any composite will also have its associated roles added to the claims and assertions of the authentication response

sent back to the client.

## 9.4. User Role Mappings

User role mappings can be assigned individually to each user through the `Role Mappings` tab for that single user.

### *Role Mappings*

The screenshot shows the Keycloak administration interface. On the left, a sidebar menu includes 'Configure' (with 'Realm Settings', 'Clients', 'Client Templates', 'Roles', 'Identity Providers', 'User Federation', 'Authentication') and 'Manage' (with 'Groups', 'Users', 'Sessions', 'Events', 'Import'). The 'Users' option is currently selected. In the main content area, the URL is 'Users > johndoe'. The 'Role Mappings' tab is active. The 'Available Roles' section contains 'admin', 'create-realm', 'developer' (which is highlighted with a blue border), and 'employee'. Below this is a button 'Add selected >'. The 'Assigned Roles' section contains 'offline\_access' and 'uma\_authorization'. Below this is a button '< Remove selected'. The 'Effective Roles' section also contains 'offline\_access' and 'uma\_authorization'. Below this is a button 'Select client to view roles for client'.

In the above example, we are about to assign the composite role `developer` that was created in the [Composite Roles](#) chapter.

### *Effective Role Mappings*

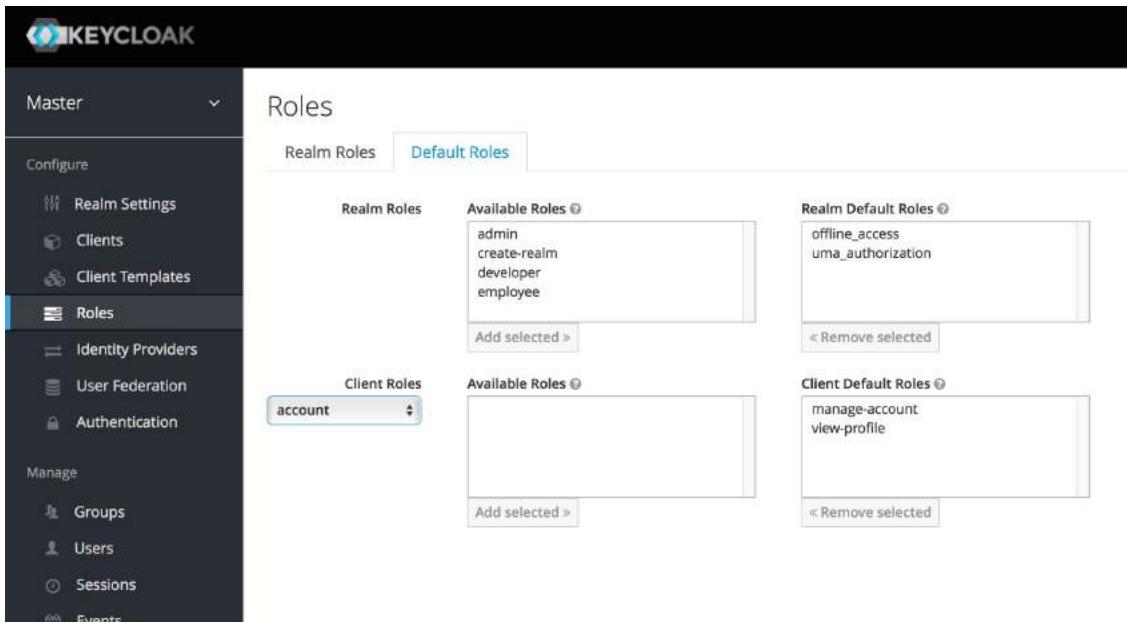
The screenshot shows the Keycloak administration interface. On the left, there's a sidebar with a 'Master' dropdown and sections for 'Configure' (Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication) and 'Manage' (Groups, Users, Sessions, Events). The 'Users' section is currently selected. In the main area, the URL is 'Users > johndoe'. The user details for 'johndoe' are shown, including a 'Role Mappings' tab which is active. Under 'Role Mappings', there are four panels: 'Realm Roles' (available: admin, create-realm), 'Assigned Roles' (developer, offline\_access, uma\_authorization), and 'Effective Roles' (developer, employee, offline\_access, uma\_authorization). Below these panels is a note: 'Select client to view roles for client'.

Once the `developer` role is assigned, you see that the `employee` role that is associated with the `developer` composite shows up in the `Effective Roles`. `Effective Roles` are all roles that are explicitly assigned to the user as well as any roles that are inherited from composites.

#### 9.4.1. Default Roles

Default roles allow you to automatically assign user role mappings when any user is newly created or imported through [Identity Brokering](#). To specify default roles go to the `Roles` left menu item, and click the `Default Roles` tab.

#### *Default Roles*



As you can see from the screenshot, there are already a number of *default roles* set up by default.

## 9.5. Role Scope Mappings

When an OIDC access token or SAML assertion is created, all the user role mappings of the user are, by default, added as claims within the token or assertion. Applications use this information to make access decisions on the resources controlled by that application. In KeyCloak, access tokens are digitally signed and can actually be re-used by the application to invoke on other remotely secured REST services. This means that if an application gets compromised or there is a rogue client registered with the realm, attackers can get access tokens that have a broad range of permissions and your whole network is compromised. This is where *role scope mappings* becomes important.

*Role Scope Mappings* is a way to limit the roles that get declared inside an access token. When a client requests that a user be authenticated, the access token they receive back will only contain the role mappings

you've explicitly specified for the client's scope. This allows you to limit the permissions each individual access token has rather than giving the client access to all of the user's permissions. By default, each client gets all the role mappings of the user. You can view this in the `Scope` tab of each client.

### *Full Scope*

The screenshot shows the Keycloak administration interface. The left sidebar is titled 'Master' and contains a 'Clients' section with 'Myapp' selected. The main content area shows the 'Scope' tab for 'myapp'. At the top of this tab, there is a switch labeled 'Full Scope Allowed' which is currently set to 'ON'. Below the switch, there is a heading 'myapp Scope Mappings'.

You can see from the picture that the effective roles of the scope are every declared role in the realm. To change this default behavior, you must explicitly turn off the `Full Scope Allowed` switch and declare the specific roles you want in each individual client. Alternatively, you can also use [client scopes](#) to define the same role scope mappings for a whole set of clients.

### *Partial Scope*

KEYCLOAK

Clients > myapp

Myapp

Settings Credentials Roles Mappers Scope Revocation Sessions Offline Access Cluster

myapp Scope Mappings

Full Scope Allowed OFF

Realm Roles Available Roles Assigned Roles Effective Roles

admin create-realm developer employee offline\_access

Add selected << Remove selected

Select client to view roles for client:

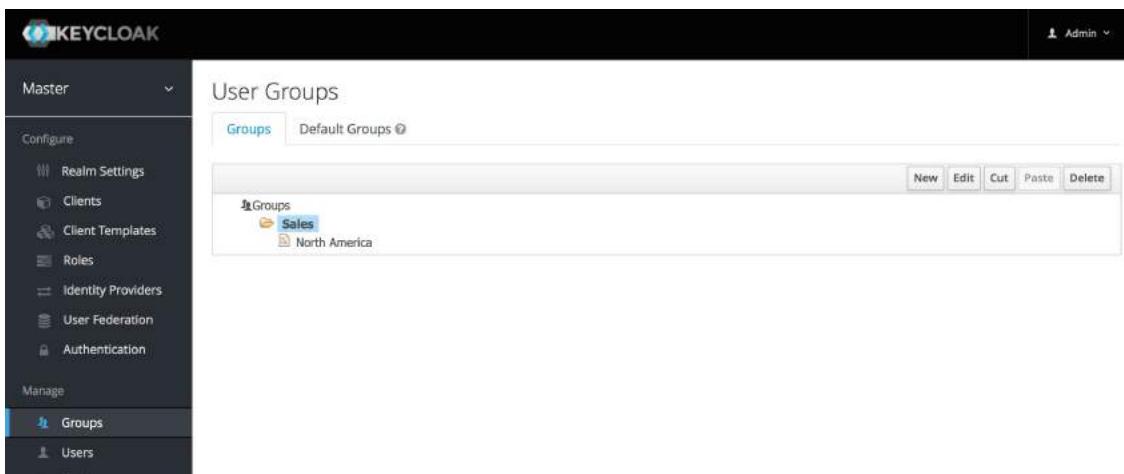
Client Roles

Developer

# 10. Groups

Groups in Keycloak allow you to manage a common set of attributes and role mappings for a set of users. Users can be members of zero or more groups. Users inherit the attributes and role mappings assigned to each group. To manage groups go to the `Groups` left menu item.

## *Groups*



The screenshot shows the Keycloak administration interface. The top navigation bar has the Keycloak logo and the word "KEYCLOAK". On the right, there is a user icon and the text "Admin". The left sidebar is titled "Master" and contains several sections: "Configure" (with sub-options like Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication), "Manage" (with sub-options like Groups and Users). The "Groups" option under "Manage" is currently selected and highlighted in blue. The main content area is titled "User Groups" and shows two tabs: "Groups" (which is active) and "Default Groups". Below the tabs, there is a tree view showing a single node named "Sales" which has a child node named "North America". At the bottom of the main content area, there is a toolbar with buttons for New, Edit, Cut, Paste, and Delete.

Groups are hierarchical. A group can have many subgroups, but a group can only have one parent. Subgroups inherit the attributes and role mappings from the parent. This applies to the user as well. So, if you have a parent group and a child group and a user that only belongs to the child group, the user inherits the attributes and role mappings of both the parent and child. In this example, we have a top level `Sales` group and a child `North America` subgroup. To add a group, click on the parent you want to add a new child to and click `New` button. Select the `Groups` icon in the tree to make a top-level group. Entering in a group name in the `Create Group` screen and hitting `Save` will bring you to the individual group management page.

## *Group*

The screenshot shows the Keycloak administration interface. The left sidebar is titled 'Master' and contains sections for 'Configure' (Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication) and 'Manage' (Groups, Users). The 'Groups' section is currently selected. The main content area shows a group named 'Europe'. The top navigation bar for the group page includes 'Groups > Europe'. Below the group name, there are tabs for 'Settings' (which is active), 'Attributes', 'Role Mappings', and 'Members'. A form is displayed with the field 'Name \*' containing 'Europe', and buttons for 'Save' and 'Cancel'.

The `Attributes` and `Role Mappings` tab work exactly as the tabs with similar names under a user. Any attributes and role mappings you define will be inherited by the groups and users that are members of this group.

To add a user to a group you need to go all the way back to the user detail page and click on the `Groups` tab there.

## *User Groups*

Select a group from the `Available Groups` tree and hit the `join` button to add the user to a group. Vice versa to remove a group. Here we've added the user *Jim* to the *North America* sales group. If you go back to the detail page for that group and select the `Membership` tab, *Jim* is now displayed there.

### *Group Membership*

Username	Last Name	First Name	Email	Action
johndoe	Doe	John	johndoe@example.com	Edit

## 10.1. Groups vs. Roles

In the IT world the concepts of Group and Role are often blurred and

interchangeable. In Keycloak, Groups are just a collection of users that you can apply roles and attributes to in one place. Roles define a type of user and applications assign permission and access control to roles

Aren't [Composite Roles](#) also similar to Groups? Logically they provide the same exact functionality, but the difference is conceptual. Composite roles should be used to apply the permission model to your set of services and applications. Groups should focus on collections of users and their roles in your organization. Use groups to manage users. Use composite roles to manage applications and services.

## 10.2. Default Groups

Default groups allow you to automatically assign group membership whenever any new user is created or imported through [Identity Brokering](#). To specify default groups go to the `Groups` left menu item, and click the `Default Groups` tab.

### *Default Groups*

The screenshot shows the Keycloak administration interface. The top navigation bar has the Keycloak logo and the text "KEYCLOAK". Below it, a dropdown menu shows "Master" and a list of configuration options: Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, and Authentication. The "Groups" option under the "Manage" section is highlighted with a blue bar at the bottom of the sidebar.

The main content area is titled "User Groups". At the top of this section, there are two tabs: "Groups" and "Default Groups", with "Default Groups" being the active tab. Below the tabs are two buttons: "Default Groups" (with a "Remove" button) and "Available Groups".

The "Default Groups" section contains a dropdown menu labeled "Select a type". The "Available Groups" section lists a group named "Sales" which contains three sub-groups: "Europe" and "North America". There is also an "Add" button next to the "Available Groups" list.

---

# 11. Admin Console Access Control and Permissions

Each realm created on the Keycloak has a dedicated Admin Console from which that realm can be managed. The `master` realm is a special realm that allows admins to manage more than one realm on the system. You can also define fine-grained access to users in different realms to manage the server. This chapter goes over all the scenarios for this.

## 11.1. Master Realm Access Control

The `master` realm in Keycloak is a special realm and treated differently than other realms. Users in the Keycloak `master` realm can be granted permission to manage zero or more realms that are deployed on the Keycloak server. When a realm is created, Keycloak automatically creates various roles that grant fine-grain permissions to access that new realm. Access to The Admin Console and Admin REST endpoints can be controlled by mapping these roles to users in the `master` realm. It's possible to create multiple super users, as well as users that can only manage specific realms.

### 11.1.1. Global Roles

There are two realm-level roles in the `master` realm. These are:

- `admin`
- `create-realm`

Users with the `admin` role are super users and have full access to ma-

nage any realm on the server. Users with the `create-realm` role are allowed to create new realms. They will be granted full access to any new realm they create.

### 11.1.2. Realm Specific Roles

Admin users within the `master` realm can be granted management privileges to one or more other realms in the system. Each realm in KeyCloak is represented by a client in the `master` realm. The name of the client is `<realm name>-realm`. These clients each have client-level roles defined which define varying level of access to manage an individual realm.

The roles available are:

- `view-realm`
- `view-users`
- `view-clients`
- `view-events`
- `manage-realm`
- `manage-users`
- `create-client`
- `manage-clients`
- `manage-events`
- `view-identity-providers`
- `manage-identity-providers`

- impersonation

Assign the roles you want to your users and they will only be able to use that specific part of the administration console.

Admins with the `manage-users` role will only be able to assign admin roles to users that they themselves have. So, if an admin has the `manage-users` role but doesn't have the `manage-realm` role, they will not be able to assign this role.

## 11.2. Dedicated Realm Admin Consoles

Each realm has a dedicated Admin Console that can be accessed by going to the url `/auth/admin/{realm-name}/console`. Users within that realm can be granted realm management permissions by assigning specific user role mappings.

Each realm has a built-in client called `realm-management`. You can view this client by going to the `Clients` left menu item of your realm. This client defines client-level roles that specify permissions that can be granted to manage the realm.

- `view-realm`
- `view-users`
- `view-clients`
- `view-events`
- `manage-realm`

- manage-users
- create-client
- manage-clients
- manage-events
- view-identity-providers
- manage-identity-providers
- impersonation

Assign the roles you want to your users and they will only be able to use that specific part of the administration console.

### 11.3. Fine Grain Admin Permissions

Sometimes roles like `manage-realm` or `manage-users` are too coarse grain and you want to create restricted admin accounts that have more fine grain permissions. Keycloak allows you to define and assign restricted access policies for managing a realm. Things like:

- Managing one specific client
- Managing users that belong to a specific group
- Managing membership of a group
- Limited user management.
- Fine grain impersonation control
- Being able to assign a specific restricted set of roles to users.
- Being able to assign a specific restricted set of roles to a composite

role.

- Being able to assign a specific restricted set of roles to a client's scope.
- New general policies for viewing and managing users, groups, roles, and clients.

There's some important things to note about fine grain admin permissions:

- Fine grain admin permissions were implemented on top of [Authorization Services](#). It is highly recommended that you read up on those features before diving into fine grain permissions.
- Fine grain permissions are only available within [dedicated admin consoles](#) and admins defined within those realms. You cannot define cross-realm fine grain permissions.
- Fine grain permissions are used to grant additional permissions. You cannot override the default behavior of the built in admin roles.

### 11.3.1. Managing One Specific Client

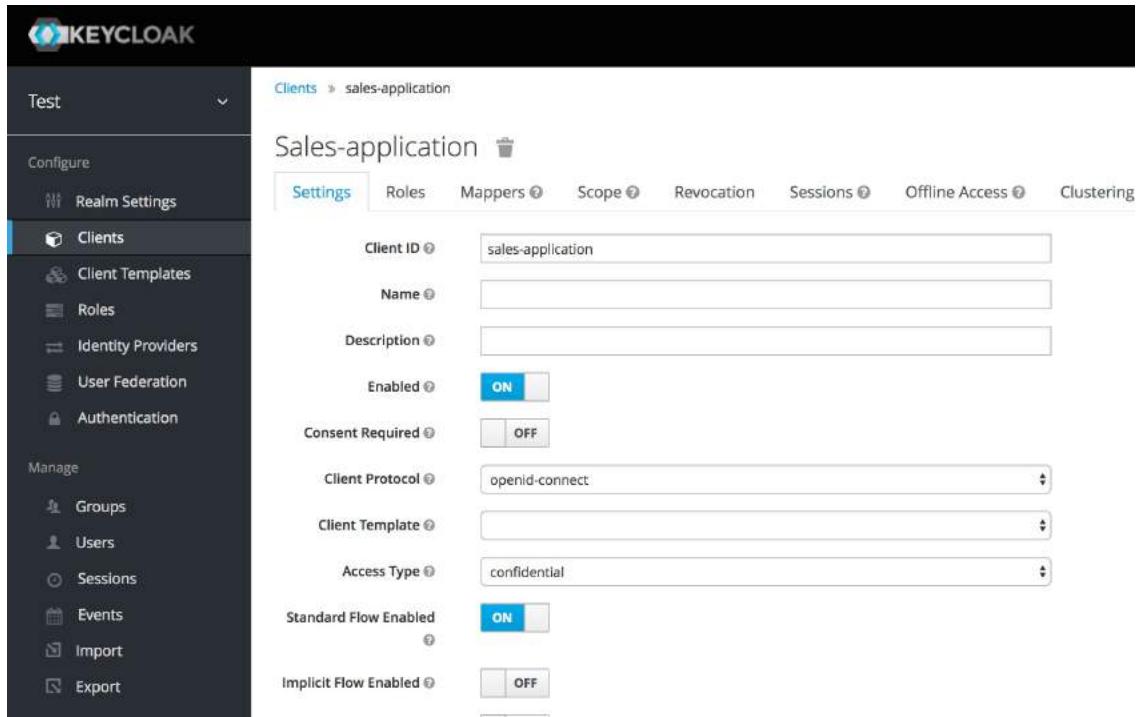
Let's look first at allowing an admin to manage one client and one client only. In our example we have a realm called `test` and a client called `sales-application`. In realm `test` we will give a user in that realm permission to only manage that application.

You cannot do cross realm fine grain permissions. Admins in the `master` realm are limited to the predefined admin roles defined in previous chapters.

## Permission Setup

The first thing we must do is login to the Admin Console so we can set up permissions for that client. We navigate to the management section of the client we want to define fine-grain permissions for.

### *Client Management*

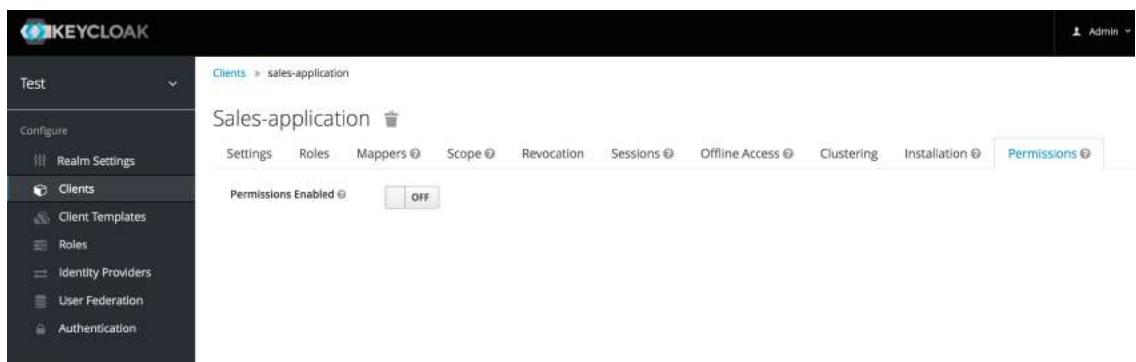


The screenshot shows the Keycloak Admin Console interface. The left sidebar is titled 'Test' and contains sections for 'Configure' (Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication) and 'Manage' (Groups, Users, Sessions, Events, Import, Export). The main content area is titled 'Clients > sales-application' and shows the 'Sales-application' client details. The 'Settings' tab is selected. The client configuration includes:

- Client ID: sales-application
- Name: (empty)
- Description: (empty)
- Enabled: ON
- Consent Required: OFF
- Client Protocol: openid-connect
- Client Template: (empty)
- Access Type: confidential
- Standard Flow Enabled: ON
- Implicit Flow Enabled: OFF

You should see a tab menu item called `Permissions`. Click on that tab.

### *Client Permissions Tab*



The screenshot shows the same Keycloak Admin Console interface as before, but the 'Permissions' tab is now selected in the top navigation bar. The main content area shows the 'Sales-application' client details, including the 'Permissions Enabled' setting which is currently OFF.

By default, each client is not enabled to do fine grain permissions. So

turn the `Permissions Enabled` switch to on to initialize permissions.

If you turn the `Permissions Enabled` switch to off, it will delete any and all permissions you have defined for this client.

## *Client Permissions Tab*

The screenshot shows the Keycloak administration interface. The left sidebar is titled 'Test' and includes sections for 'Configure' (Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication), 'Manage' (Groups, Users), and 'Logs'. The 'Clients' section is currently selected. The main content area shows the 'Sales-application' client details. At the top, there are tabs for 'Settings', 'Roles', 'Mappers', 'Scope', 'Revocation', 'Sessions', 'Offline Access', 'Clustering', 'Installation', and 'Permissions'. The 'Permissions' tab is selected. Below the tabs, a 'Permissions Enabled' switch is set to 'ON'. A table lists various permission objects:

scope-name	Description	Actions
view	Policies that decide if an admin can view this client.	Edit
map-roles-client-scope	Policies that decide if an admin can apply roles defined by this client to the client scope of another client.	Edit
configure	Reduced management permissions for admin. Cannot set scope, template, or protocol mappers.	Edit
map-roles-composite	Policies that decide if an admin can apply roles defined by this client as a composite to another role	Edit
map-roles	Policies that decide if an admin can map roles defined by this client	Edit
manage	Policies that decide if an admin can manage this client	Edit

When you switch `Permissions Enabled` to on, it initializes various permission objects behind the scenes using [Authorization Services](#). For this example, we're interested in the `manage` permission for the client. Clicking on that will redirect you to the permission that handles the `manage` permission for the client. All authorization objects are contained in the `realm-management` client's `Authorization` tab.

## *Client Manage Permission*

The screenshot shows the Keycloak administration interface. The left sidebar is titled 'Test' and contains sections for 'Configure' (Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication) and 'Manage' (Groups, Users, Sessions). The 'Clients' section is currently selected. The main content area shows a form for managing a client permission. The form fields are: Name (manage.permission.client.4535c9b1-ad7b-4106-b509-1834d38cdc4b), Description (empty), Resource (client.resource.4535c9b1-ad7b-4106-b509-1834d38cdc4b), Scopes (manage), Apply Policy (Select a policy...), and Decision Strategy (Unanimous). There are 'Save' and 'Cancel' buttons at the bottom.

When first initialized the `manage` permission does not have any policies associated with it. You will need to create one by going to the policy tab. To get there fast, click on the `Authorization` link shown in the above image. Then click on the policies tab.

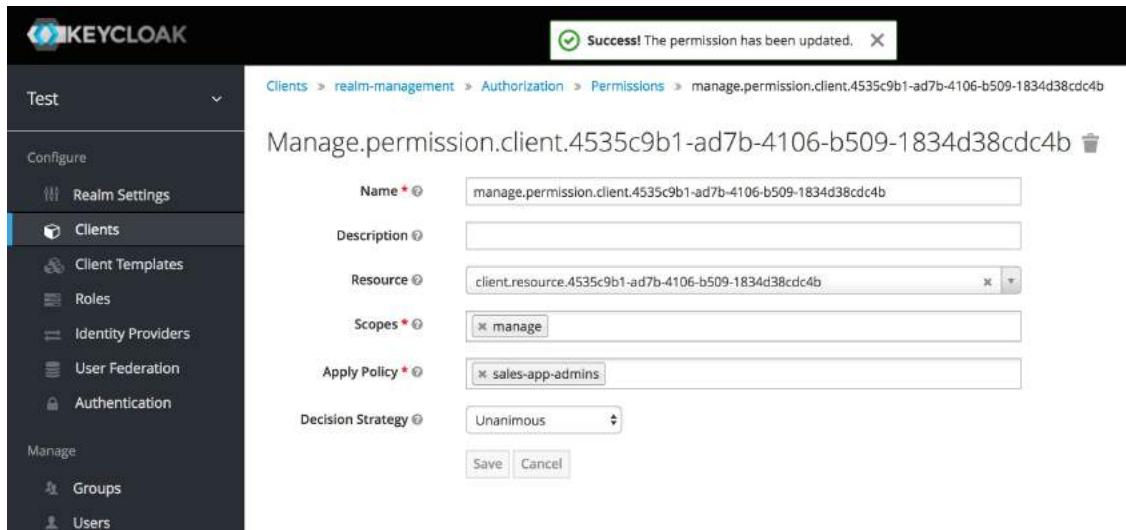
There's a pull down menu on this page called `Create policy`. There's a multitude of policies you can define. You can define a policy that is associated with a role or a group or even define rules in JavaScript. For this simple example, we're going to create a `User Policy`.

### *User Policy*

The screenshot shows the Keycloak administration interface. The left sidebar is titled 'Test' and contains sections for 'Configure' (Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication) and 'Manage' (Groups, Users, Sessions). The 'Clients' section is currently selected. The main content area shows a form for adding a new user policy. The form fields are: Name (sales-app-admins), Description (empty), Users (Select an user...), Logic (Posit), and a table for users (Username: sales-admin, Actions: Remove). There are 'Save' and 'Cancel' buttons at the bottom.

This policy will match a hard-coded user in the user database. In this case it is the `sales-admin` user. We must then go back to the `sales-application` client's `manage` permission page and assign the policy to the permission object.

### Assign User Policy



The screenshot shows the Keycloak administration interface. The left sidebar is titled 'Test' and contains sections for 'Configure' (Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication) and 'Manage' (Groups, Users). The 'Clients' section is currently selected. The main content area shows a success message: 'Success! The permission has been updated.' Below this, the details for a new permission object are displayed:

- Name \***: manage.permission.client.4535c9b1-ad7b-4106-b509-1834d38cdc4b
- Description**: (empty)
- Resource**: client.resource.4535c9b1-ad7b-4106-b509-1834d38cdc4b
- Scopes \***: manage
- Apply Policy \***: sales-app-admins
- Decision Strategy**: Unanimous

At the bottom are 'Save' and 'Cancel' buttons.

The `sales-admin` user can now has permission to manage the `sales-application` client.

There's one more thing we have to do. Go to the `Role Mappings` tab and assign the `query-clients` role to the `sales-admin`.

### Assign `query-clients`

Why do you have to do this? This role tells the Admin Console what menu items to render when the `sales-admin` visits the Admin Console. The `query-clients` role tells the Admin Console that it should render client menus for the `sales-admin` user.

**IMPORTANT** If you do not set the `query-clients` role, restricted admins like `sales-admin` will not see any menu options when they log into the Admin Console

## Testing It Out.

Next we log out of the master realm and re-login to the [dedicated admin console](#) for the `test` realm using the `sales-admin` as a username. This is located under `/auth/admin/test/console`.

### Sales Admin Login

Client ID	Enabled	Base URL	Actions		
sales-application	True	Not defined	Edit	Export	Delete

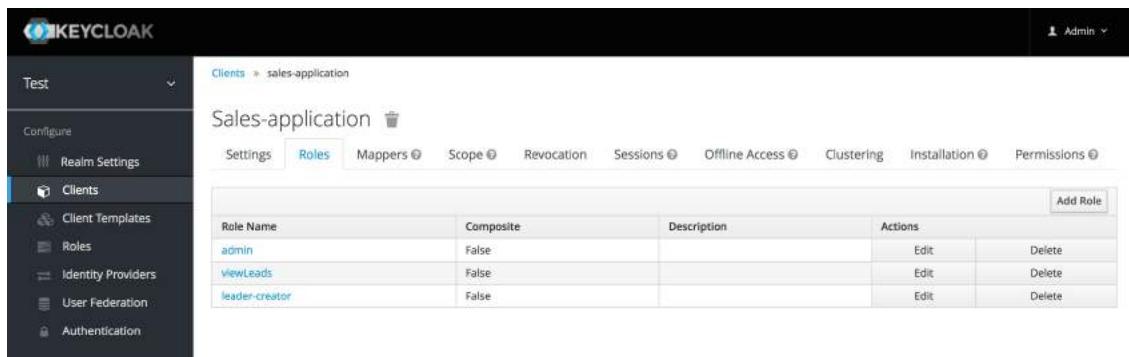
This admin is now able to manage this one client.

### 11.3.2. Restrict User Role Mapping

Another thing you might want to do is to restrict the set of roles an admin is allowed to assign to a user. Continuing our last example, let's expand the permission set of the 'sales-admin' user so that he can also control which users are allowed to access this application. Through fine grain permissions we can enable it so that the `sales-admin` can only assign roles that grant specific access to the `sales-application`. We can also restrict it so that the admin can only map roles and not perform any other types of user administration.

The `sales-application` has defined three different client roles.

#### *Sales Application Roles*



The screenshot shows the Keycloak administration interface. The left sidebar is titled 'Test' and contains 'Configure' and 'Clients' sections. Under 'Clients', 'Client Templates', 'Roles', 'Identity Providers', 'User Federation', and 'Authentication' are listed. The 'Clients' section is currently selected. The main content area shows the 'Sales-application' client details. The 'Roles' tab is active, displaying a table of roles:

Role Name	Composite	Description	Actions
admin	False		Edit Delete
viewLeads	False		Edit Delete
leader-creator	False		Edit Delete

An 'Add Role' button is located at the top right of the table.

We want the `sales-admin` user to be able to map these roles to any user in the system. The first step to do this is to allow the role to be mapped by the admin. If we click on the `viewLeads` role, you'll see that there is a `Permissions` tab for this role.

#### *View Leads Role Permission Tab*

The screenshot shows the Keycloak admin interface for managing roles. On the left, a sidebar menu is visible with sections like 'Configure', 'Clients', 'Realm Settings', 'Roles', 'Identity Providers', 'User Federation', and 'Authentication'. Under 'Clients', 'Sales-application' is selected. In the main content area, the path 'Clients > sales-application > Roles > viewLeads' is shown. The 'ViewLeads' role is displayed with its details. The 'Details' tab is active, showing the 'Role Name' as 'viewLeads'. There is a large empty 'Description' field. Below it are two toggle switches: 'Scope Param Required' is set to 'OFF', and 'Composite Roles' is also set to 'OFF'. At the bottom right are 'Save' and 'Cancel' buttons.

If we click on that tab and turn the `Permissions Enabled` on, you'll see that there are a number of actions we can apply policies to.

### *View Leads Permissions*

The screenshot shows the 'Permissions' tab for the 'ViewLeads' role. The 'Permissions Enabled' switch is turned 'ON'. A table lists three permissions:

scope-name	Description	Actions
map-role-client-scope	Policies that decide if an admin can apply this role to the client scope of a client	Edit
map-role-composite	Policies that decide if an admin can apply this role as a composite to another role	Edit
map-role	Policies that decide if an admin can map role this role to a user or group	Edit

The one we are interested in is `map-role`. Click on this permission and add the same User Policy that was created in the earlier example.

### *Map-roles Permission*

The screenshot shows the Keycloak admin interface under the 'Clients' section. A new permission is being created with the following details:

- Name:** map-roles.permission.client.bbfafae0b7-7297-40b4-87f3-94604335461f
- Description:** (empty)
- Resource:** client.resource.bbfafae0b7-7297-40b4-87f3-94604335461f
- Scopes:** map-roles
- Apply Policy:** sales-app-admins
- Decision Strategy:** Unanimous

What we've done is say that the `sales-admin` can map the `view-Leads` role. What we have not done is specify which users the admin is allowed to map this role too. To do that we must go to the `Users` section of the admin console for this realm. Clicking on the `Users` left menu item brings us to the users interface of the realm. You should see a `Permissions` tab. Click on that and enable it.

## Users Permissions

The screenshot shows the 'Users' section of the Keycloak admin console with the 'Permissions' tab selected. The 'Permissions Enabled' switch is turned on. A table lists the available permissions:

scope-name	Description	Actions
user-impersonated	Policies that decide which users can be impersonated. These policies are applied to the user being impersonated.	Edit
view	Policies that decide if an admin can view all users in realm	Edit
manage-group-membership	Policies that decide if an admin can manage group membership for all users in the realm. This is used in conjunction with specific group policy	Edit
impersonate	Policies that decide if admin can impersonate other users	Edit
manage	Policies that decide if an admin can manage all users in the realm	Edit
map-roles	Policies that decide if admin can map roles for all users	Edit

The permission we are interested in is `map-roles`. This is a restrictive policy in that it only allows admins the ability to map roles to a user. If we click on the `map-roles` permission and again add the User Policy we created for this, our `sales-admin` will be able to map roles to any

user.

The last thing we have to do is add the `view-users` role to the `sales-admin`. This will allow the admin to view users in the realm he wants to add the `sales-application` roles to.

### Add `view-users`

The screenshot shows the Keycloak Admin UI for the `sales-admin` realm. On the left, the sidebar has sections for `Configure` (Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication) and `Manage` (Groups, Users, Sessions, Events). The `Users` section is selected. The main area shows the `Sales-admin` realm with tabs for Details, Attributes, Credentials, Role Mappings (which is active), Groups, Consents, and Sessions. Under the `Role Mappings` tab, there are four panels: `Realm Roles` (containing `realm-role`), `Available Roles` (containing `realm-role`), `Assigned Roles` (containing `offline_access` and `uma_authorization`), and `Effective Roles` (containing `offline_access` and `uma_authorization`). Below these, under `Client Roles` (`realm-management` dropdown), there are two panels: `Available Roles` (containing `view-clients`, `view-events`, `view-identity-providers`, `view-realm`, and `view-users`) and `Assigned Roles` (empty). The `view-users` role is highlighted in the `Available Roles` list.

### Testing It Out.

Next we log out of the master realm and re-login to the [dedicated admin console](#) for the `test` realm using the `sales-admin` as a username. This is located under `/auth/admin/test/console`.

You will see that now the `sales-admin` can view users in the system. If you select one of the users you'll see that each user detail page is read only, except for the `Role Mappings` tab. Going to these tab you'll find that there are no `Available` roles for the admin to map to the user except when we browse the `sales-application` roles.

### Add `viewLeads`

We've only specified that the `sales-admin` can map the `viewLeads` role.

## Per Client map-roles Shortcut

It would be tedious if we had to do this for every client role that the `sales-application` published. To make things easier, there's a way to specify that an admin can map any role defined by a client. If we log back into the admin console to our master realm admin and go back to the `sales-application` permissions page, you'll see the `map-roles` permission.

## *Client map-roles Permission*

scope-name	Description	Actions
view	Policies that decide if an admin can view this client	Edit
map-roles-client-scope	Policies that decide if an admin can apply roles defined by this client to the client scope of another client	Edit
configure	Reduced management permissions for admin. Cannot set scope, template, or protocol mappers.	Edit
map-roles-composite	Policies that decide if an admin can apply roles defined by this client as a composite to another role	Edit
map-roles	Policies that decide if an admin can map roles defined by this client	Edit
manage	Policies that decide if an admin can manage this client	Edit

If you grant access to this particular permission to an admin, that admin will be able map any role defined by the client.

### 11.3.3. Full List of Permissions

You can do a lot more with fine grain permissions beyond managing a specific client or the specific roles of a client. This chapter defines the whole list of permission types that can be described for a realm.

#### Role

When going to the `Permissions` tab for a specific role, you will see these permission types listed.

#### map-role

Policies that decide if an admin can map this role to a user. These policies only specify that the role can be mapped to a user, not that the admin is allowed to perform user role mapping tasks. The admin will also have to have manage or role mapping permissions. See [Users Permissions](#) for more information.

#### map-role-composite

Policies that decide if an admin can map this role as a composite to another role. An admin can define roles for a client if he has manage permissions for that client but he will not be able to add composites to those roles unless he has the `map-role-composite` privileges for the role he wants to add as a composite.

#### map-role-client-scope

Policies that decide if an admin can apply this role to the scope of a client. Even if the admin can manage the client, he will not have per-

mission to create tokens for that client that contain this role unless this privilege is granted.

## Client

When going to the `Permissions` tab for a specific client, you will see these permission types listed.

### view

Policies that decide if an admin can view the client's configuration.

### manage

Policies that decide if an admin can view and manage the client's configuration. There is some issues with this in that privileges could be leaked unintentionally. For example, the admin could define a protocol mapper that hardcoded a role even if the admin does not have privileges to map the role to the client's scope. This is currently the limitation of protocol mappers as they don't have a way to assign individual permissions to them like roles do.

### configure

Reduced set of privileges to manage the client. Its like the `manage` scope except the admin is not allowed to define protocol mappers, change the client template, or the client's scope.

### map-roles

Policies that decide if an admin can map any role defined by the client to a user. This is a shortcut, easy-of-use feature to avoid having to define policies for each and every role defined by the client.

## **map-roles-composite**

Policies that decide if an admin can map any role defined by the client as a composite to another role. This is a shortcut, easy-of-use feature to avoid having to define policies for each and every role defined by the client.

## **map-roles-client-scope**

Policies that decide if an admin can map any role defined by the client to the scope of another client. This is a shortcut, easy-of-use feature to avoid having to define policies for each and every role defined by the client.

## **Users**

When going to the `Permissions` tab for all users, you will see these permission types listed.

### **view**

Policies that decide if an admin can view all users in the realm.

### **manage**

Policies that decide if an admin can manage all users in the realm.

This permission grants the admin the privilege to perform user role mappings, but it does not specify which roles the admin is allowed to map. You'll need to define the privilege for each role you want the admin to be able to map.

### **map-roles**

This is a subset of the privileges granted by the `manage` scope. In this case the admin is only allowed to map roles. The admin is not al-

lowed to perform any other user management operation. Also, like `manage`, the roles that the admin is allowed to apply must be specified per role or per set of roles if dealing with client roles.

### **manage-group-membership**

Similar to `map-roles` except that it pertains to group membership: which groups a user can be added or removed from. These policies just grant the admin permission to manage group membership, not which groups the admin is allowed to manage membership for.

You'll have to specify policies for each group's `manage-members` permission.

### **impersonate**

Policies that decide if the admin is allowed to impersonate other users. These policies are applied to the admin's attributes and role mappings.

### **user-impersonated**

Policies that decide which users can be impersonated. These policies will be applied to the user being impersonated. For example, you might want to define a policy that will forbid anybody from impersonating a user that has admin privileges.

## **Group**

When going to the `Permissions` tab for a specific group, you will see these permission types listed.

### **view**

Policies that decide if the admin can view information about the

group.

### **manage**

Policies that decide if the admin can manage the configuration of the group.

### **view-members**

Policies that decide if the admin can view the user details of members of the group.

### **manage-members**

Policies that decide if the admin can manage the users that belong to this group.

### **manage-membership**

Policies that decide if an admin can change the membership of the group. Add or remove members from the group.

## **11.4. Realm Keys**

The authentication protocols that are used by KeyCloak require cryptographic signatures and sometimes encryption. KeyCloak uses asymmetric key pairs, a private and public key, to accomplish this.

KeyCloak has a single active keypair at a time, but can have several passive keys as well. The active keypair is used to create new signatures, while the passive keypairs can be used to verify previous signatures. This makes it possible to regularly rotate the keys without any downtime or interruption to users.

When a realm is created a key pair and a self-signed certificate is automatically generated.

To view the active keys for a realm select the realm in the admin console click on `Realm settings` then `Keys`. This will show the currently active keys for the realm. Keycloak currently only supports RSA signatures so there is only one active keypair. In the future as more signature algorithms are added there will be more active keypairs.

To view all available keys select `All`. This will show all active, passive and disabled keys. A keypair can have the status `Active`, but still not be selected as the currently active keypair for the realm. The selected active pair which is used for signatures is selected based on the first key provider sorted by priority that is able to provide an active keypair.

#### 11.4.1. Rotating keys

It's recommended to regularly rotate keys. To do so you should start by creating new keys with a higher priority than the existing active keys. Or create new keys with the same priority and making the previous keys passive.

Once new keys are available all new tokens and cookies will be signed with the new keys. When a user authenticates to an application the SSO cookie is updated with the new signature. When OpenID Connect tokens are refreshed new tokens are signed with the new keys. This means that over time all cookies and tokens will use the new keys and after a while the old keys can be removed.

How long you wait to delete old keys is a tradeoff between security and making sure all cookies and tokens are updated. In general it should be

acceptable to drop old keys after a few weeks. Users that have not been active in the period between the new keys where added and the old keys removed will have to re-authenticate.

This also applies to offline tokens. To make sure they are updated the applications need to refresh the tokens before the old keys are removed.

As a guideline it may be a good idea to create new keys every 3-6 months and delete old keys 1-2 months after the new keys where created.

#### 11.4.2. Adding a generated keypair

To add a new generated keypair select `Providers` and choose `rsa-generated` from the dropdown. You can change the priority to make sure the new keypair becomes the active keypair. You can also change the `keysize` if you want smaller or larger keys (default is 2048, supported values are 1024, 2048 and 4096).

Click `Save` to add the new keys. This will generate a new keypair including a self-signed certificate.

Changing the priority for a provider will not cause the keys to be regenerated, but if you want to change the keysize you can edit the provider and new keys will be generated.

#### 11.4.3. Adding an existing keypair and certificate

To add a keypair and certificate obtained elsewhere select `Providers` and choose `rsa` from the dropdown. You can change the priority to make sure the new keypair becomes the active keypair.

Click on `Select file` for `Private RSA Key` to upload your private key. The file should be encoded in PEM format. You don't need to upload the public key as it is automatically extracted from the private key.

If you have a signed certificate for the keys click on `Select file` next to `X509 Certificate`. If you don't have one you can skip this and a self-signed certificate will be generated.

#### 11.4.4. Loading keys from a Java Keystore

To add a keypair and certificate stored in a Java Keystore file on the host select `Providers` and choose `java-keystore` from the drop-down. You can change the priority to make sure the new keypair becomes the active keypair.

Fill in the values for `Keystore`, `Keystore Password`, `Key Alias` and `Key Password` and click on `Save`.

#### 11.4.5. Making keys passive

Locate the keypair in `Active` or `All` then click on the provider in the `Provider` column. This will take you to the configuration screen for the key provider for the keys. Click on `Active` to turn it `OFF`, then click on `Save`. The keys will no longer be active and can only be used for verifying signatures.

#### 11.4.6. Disabling keys

Locate the keypair in `Active` or `All` then click on the provider in the `Provider` column. This will take you to the configuration screen for the key provider for the keys. Click on `Enabled` to turn it `OFF`, then click on `Save`. The keys will no longer be enabled.

Alternatively, you can delete the provider from the `Providers` table.

#### 11.4.7. Compromised keys

KeyCloak has the signing keys stored just locally and they are never shared with the client applications, users or other entities. However if you think that your realm signing key was compromised, you should first generate new keypair as described above and then immediately remove the compromised keypair.

Then to ensure that client applications won't accept the tokens signed by the compromised key, you should update and push not-before policy for the realm, which is doable from the admin console. Pushing new policy will ensure that client applications won't accept the existing tokens signed by the compromised key, but also the client application will be forced to download new keypair from the KeyCloak, hence the tokens signed by the compromised key won't be valid anymore. Note that your REST and confidential clients must have set `Admin URL`, so that KeyCloak is able to send them the request about pushed not-before policy.

---

## 12. Identity Brokering

An Identity Broker is an intermediary service that connects multiple service providers with different identity providers. As an intermediary service, the identity broker is responsible for creating a trust relationship with an external identity provider in order to use its identities to access internal services exposed by service providers.

From a user perspective, an identity broker provides a user-centric and centralized way to manage identities across different security domains or realms. An existing account can be linked with one or more identities from different identity providers or even created based on the identity information obtained from them.

An identity provider is usually based on a specific protocol that is used to authenticate and communicate authentication and authorization information to their users. It can be a social provider such as Facebook, Google or Twitter. It can be a business partner whose users need to access your services. Or it can be a cloud-based identity service that you want to integrate with.

Usually, identity providers are based on the following protocols:

- SAML v2.0
- OpenID Connect v1.0
- OAuth v2.0

In the next sections we'll see how to configure and use KeyCloak as an identity broker, covering some important aspects such as:

- Social Authentication
- OpenID Connect v1.0 Brokering
- SAML v2.0 Brokering
- Identity Federation

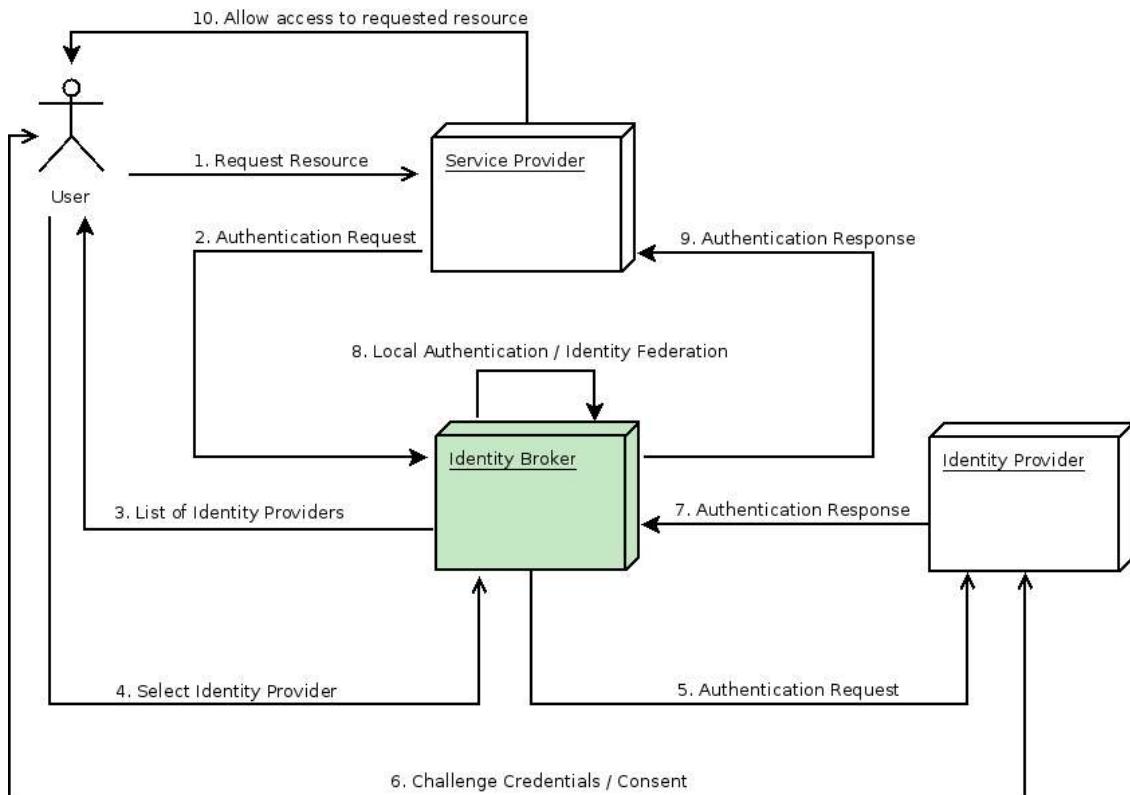
## 12.1. Brokering Overview

When using KeyCloak as an identity broker, users are not forced to provide their credentials in order to authenticate in a specific realm. Instead, they are presented with a list of identity providers from which they can authenticate.

You can also configure a default identity provider. In this case the user will not be given a choice, but will instead be redirected directly to the default provider.

The following diagram demonstrates the steps involved when using KeyCloak to broker an external identity provider:

### *Identity Broker Flow*



1. User is not authenticated and requests a protected resource in a client application.
2. The client applications redirects the user to KeyCloak to authenticate.
3. At this point the user is presented with the login page where there is a list of identity providers configured in a realm.
4. User selects one of the identity providers by clicking on its respective button or link.
5. KeyCloak issues an authentication request to the target identity provider asking for authentication and the user is redirected to the login page of the identity provider. The connection properties and other configuration options for the identity provider were previously set by the administrator in the Admin Console.

6. User provides his credentials or consent in order to authenticate with the identity provider.
7. Upon a successful authentication by the identity provider, the user is redirected back to KeyCloak with an authentication response. Usually this response contains a security token that will be used by KeyCloak to trust the authentication performed by the identity provider and retrieve information about the user.
8. Now KeyCloak is going to check if the response from the identity provider is valid. If valid, it will import and create a new user or just skip that if the user already exists. If it is a new user, KeyCloak may ask the identity provider for information about the user if that info doesn't already exist in the token. This is what we call *identity federation*. If the user already exists KeyCloak may ask him to link the identity returned from the identity provider with the existing account. We call this process *account linking*. What exactly is done is configurable and can be specified by setup of [First Login Flow](#). At the end of this step, KeyCloak authenticates the user and issues its own token in order to access the requested resource in the service provider.
9. Once the user is locally authenticated, KeyCloak redirects the user to the service provider by sending the token previously issued during the local authentication.
10. The service provider receives the token from KeyCloak and allows access to the protected resource.

There are some variations of this flow that we will talk about later. For instance, instead of presenting a list of identity providers, the client app-

lication can request a specific one. Or you can tell KeyCloak to force the user to provide additional information before federating his identity.

Different protocols may require different authentication flows. At this moment, all the identity providers supported by KeyCloak use a flow just like described above. However, regardless of the protocol in use, user experience should be pretty much the same.

As you may notice, at the end of the authentication process KeyCloak will always issue its own token to client applications. What this means is that client applications are completely decoupled from external identity providers. They don't need to know which protocol (eg.: SAML, OpenID Connect, OAuth, etc) was used or how the user's identity was validated. They only need to know about KeyCloak.

## 12.2. Default Identity Provider

It is possible to automatically redirect to a identity provider instead of displaying the login form. To enable this go to the `Authentication` page in the administration console and select the `Browser` flow. Then click on config for the `Identity Provider Redirector` authenticator. Set `Default Identity Provider` to the alias of the identity provider you want to automatically redirect users to.

If the configured default identity provider is not found the login form will be displayed instead.

This authenticator is also responsible for dealing with the `kc_id-p_hint` query parameter. See [client suggested identity provider](#) section

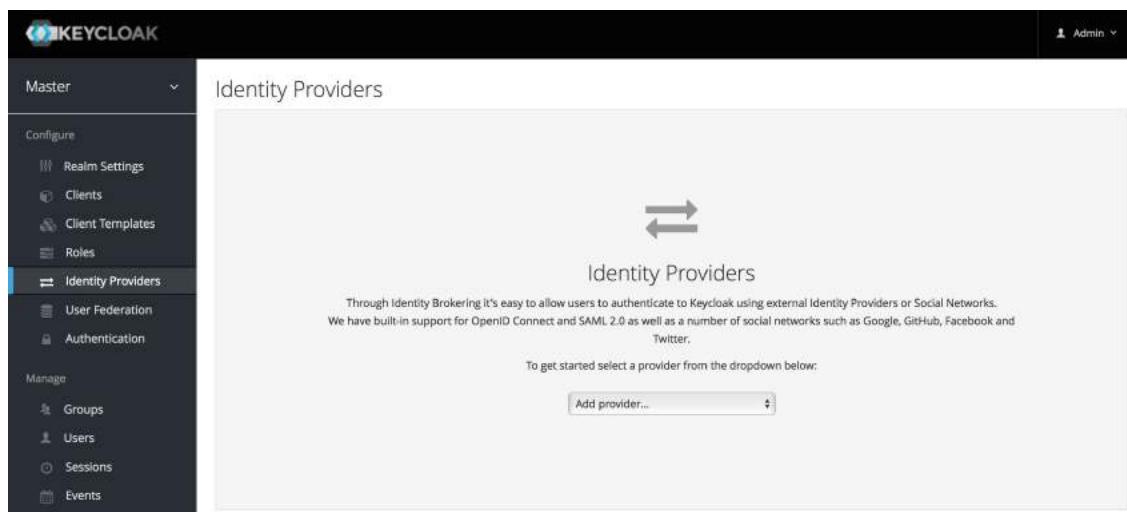
for more details.

## 12.3. General Configuration

The identity broker configuration is all based on identity providers. Identity providers are created for each realm and by default they are enabled for every single application. That means that users from a realm can use any of the registered identity providers when signing in to an application.

In order to create an identity provider click the **Identity Providers** left menu item.

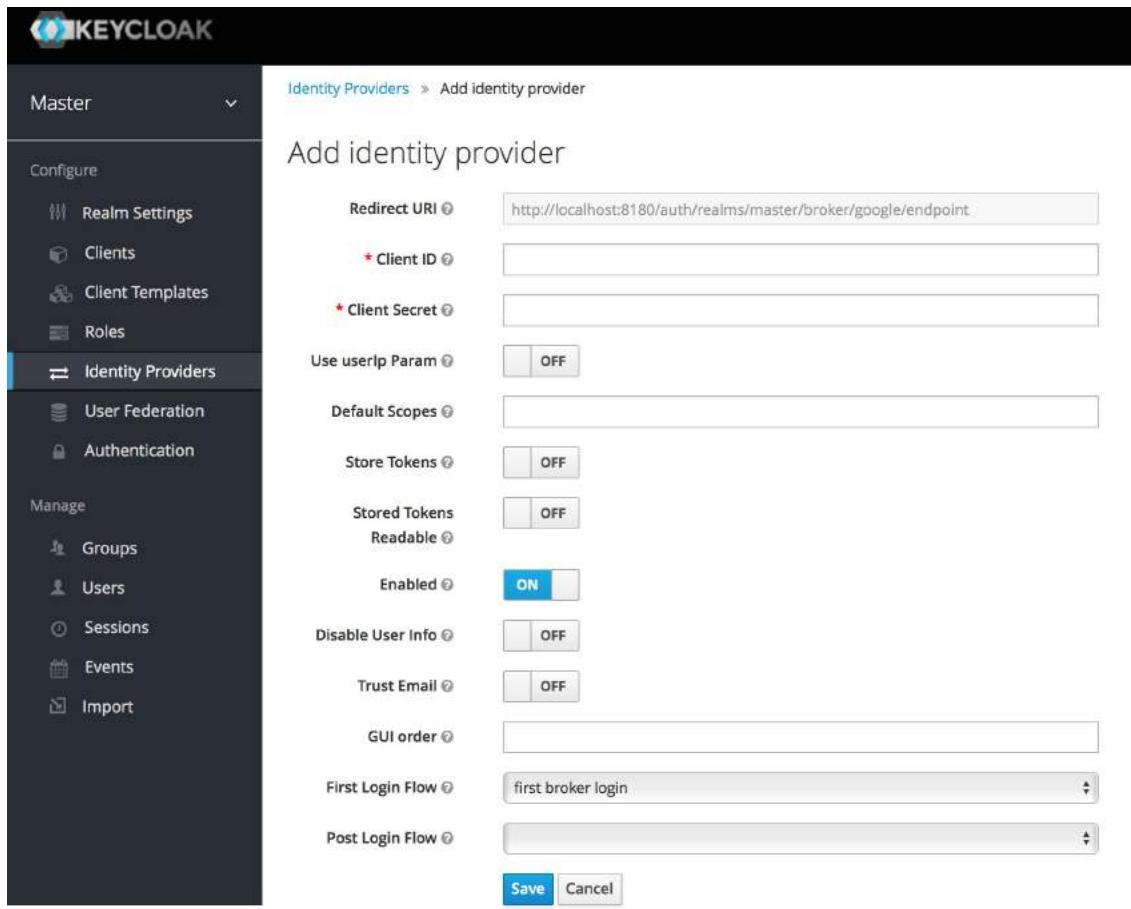
### *Identity Providers*



The screenshot shows the Keycloak administration interface. The top navigation bar has the Keycloak logo and the word "KEYCLOAK". On the far right, there is a user icon and the text "Admin". The left sidebar contains a navigation menu with the following items: "Master" (dropdown), "Configure" (dropdown) which includes "Realm Settings", "Clients", "Client Templates", "Roles", "Identity Providers" (selected and highlighted in blue), "User Federation", and "Authentication"; and "Manage" (dropdown) which includes "Groups", "Users", "Sessions", and "Events". The main content area is titled "Identity Providers". It features a large double-headed arrow icon. Below the icon, the text "Identity Providers" is displayed. A descriptive paragraph explains that through identity brokering, users can authenticate to Keycloak using external identity providers or social networks, mentioning support for OpenID Connect and SAML 2.0, and various social networks like Google, GitHub, Facebook, and Twitter. A call-to-action text "To get started select a provider from the dropdown below:" is followed by a dropdown menu labeled "Add provider...".

In the drop down list box, choose the identity provider you want to add. This will bring you to the configuration page for that identity provider type.

### *Add Identity Provider*



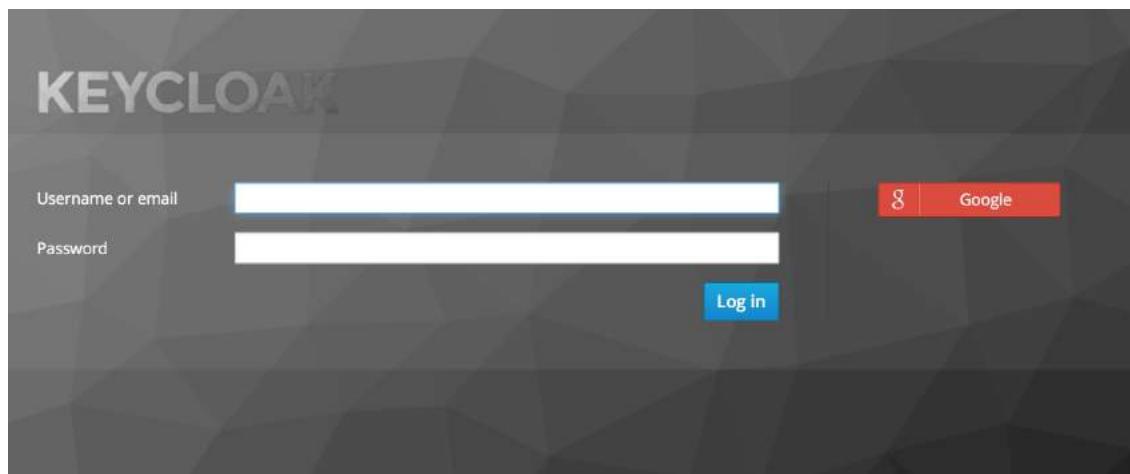
The screenshot shows the Keycloak Admin UI for adding a new identity provider. The left sidebar is dark-themed with white text, showing the 'Master' realm selected. Under 'Identity Providers', the 'Identity Providers' option is highlighted. The main content area has a light background and displays the 'Add identity provider' form. The form fields include:

- Redirect URI: http://localhost:8180/auth/realms/master/broker/google/endpoint
- \* Client ID: (empty input field)
- \* Client Secret: (empty input field)
- Use userip Param: OFF (radio button)
- Default Scopes: (empty input field)
- Store Tokens: OFF (radio button)
- Stored Tokens Readable: OFF (radio button)
- Enabled: ON (radio button)
- Disable User Info: OFF (radio button)
- Trust Email: OFF (radio button)
- GUI order: (empty input field)
- First Login Flow: first broker login (dropdown menu)
- Post Login Flow: (empty input field)

At the bottom right are 'Save' and 'Cancel' buttons.

Above is an example of configuring a Google social login provider. Once you configure an IDP, it will appear on the Keycloak login page as an option.

### *IDP login page*



## Social

Social providers allow you to enable social authentication in your realm. Keycloak makes it easy to let users log in to your application using an existing account with a social network. Currently supported providers include: Twitter, Facebook, Google, LinkedIn, Instagram, Microsoft, PayPal, Openshift v3, GitHub, GitLab, Bitbucket, and Stack Overflow.

## Protocol-based

Protocol-based providers are those that rely on a specific protocol in order to authenticate and authorize users. They allow you to connect to any identity provider compliant with a specific protocol. Keycloak provides support for SAML v2.0 and OpenID Connect v1.0 protocols. It makes it easy to configure and broker any identity provider based on these open standards.

Although each type of identity provider has its own configuration options, all of them share some very common configuration. Regardless of which identity provider you are creating, you'll see the following configuration options available:

*Table 1. Common Configuration*

Configuration	Description
Alias	The alias is a unique identifier for an identity provider. It is used to reference an identity provider internally. Some protocols such as OpenID Connect require a redirect URI or call-

	back url in order to communicate with an identity provider. In this case, the alias is used to build the redirect URI. Every single identity provider must have an alias. Examples are <code>facebook</code> , <code>google</code> , <code>idp.acme.com</code> , etc.
Enabled	Turn the provider on/off.
Hide on Login Page	When this switch is on, this provider will not be shown as a login option on the login page. Clients can still request to use this provider by using the ' <code>kc_idp_hint</code> ' parameter in the URL they use to request a login.
Account Linking Only	When this switch is on, this provider cannot be used to login users and will not be shown as an option on the login page. Existing accounts can still be linked with this provider though.
Store Tokens	Whether or not to store the token received from the identity provider.

Stored Tokens Readable	Whether or not users are allowed to retrieve the stored identity provider token. This also applies to the <i>broker</i> client-level role <i>read token</i> .
Trust Email	If the identity provider supplies an email address this email address will be trusted. If the realm required email validation, users that log in from this IDP will not have to go through the email verification process.
GUI Order	The order number that sorts how the available IDPs are listed on the login page.
First Login Flow	This is the authentication flow that will be triggered for users that log into Keycloak through this IDP for the first time ever.
Post Login Flow	Authentication flow that is triggered after the user finishes logging in with the external identity provider.

## 12.4. Social Identity Providers

For Internet facing applications, it is quite burdensome for users to have to register at your site to obtain access. It requires them to remember yet another username and password combination. Social identity providers allow you to delegate authentication to a semi-trusted and respected entity where the user probably already has an account. Keycloak provides built-in support for the most common social networks out there, such as Google, Facebook, Twitter, GitHub, LinkedIn, Microsoft and Stack Overflow.

#### 12.4.1. Bitbucket

There are a number of steps you have to complete to be able to enable login with Bitbucket.

First, open the `Identity Providers` left menu item and select `Bitbucket` from the `Add provider` drop down list. This will bring you to the `Add identity provider` page.

*Add Identity Provider*

The screenshot shows the Keycloak administration interface. The left sidebar is dark-themed and includes sections for 'Configure' (Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication) and 'Manage' (Groups, Users, Sessions, Events, Import, Export). The 'Identity Providers' section is currently selected. The main content area has a title 'Add identity provider'. It contains several configuration fields: 'Redirect URI' (set to 'http://localhost:8081/auth/realms/master/broker/gitlab/endpoint'), 'Application Id' (empty), 'Application Secret' (empty), 'Default Scopes' (empty), 'Store Tokens' (set to 'OFF'), 'Stored Tokens Readable' (set to 'OFF'), 'Enabled' (set to 'ON'), 'Trust Email' (set to 'OFF'), 'Account Linking Only' (set to 'OFF'), 'Hide on Login Page' (set to 'OFF'), 'GUI order' (empty), 'First Login Flow' (set to 'first broker login'), and 'Post Login Flow' (empty). At the bottom right are 'Save' and 'Cancel' buttons.

Before you can click `Save`, you must obtain a `Client ID` and `client Secret` from Bitbucket.

You will use the `Redirect URI` from this page in a later step, which you will provide to Bitbucket when you register KeyCloak as a client there.

### Add a New App

To enable login with Bitbucket you must first register an application project in [OAuth on Bitbucket Cloud](#).

Bitbucket often changes the look and feel of application registration, so what you see on the Bitbucket site may differ. If in doubt, see the Bitbucket documentation.

## OAuth integrated applications

You have **authorized** the following applications to interact with your Bitbucket repositories.

Name	Description	Last accessed	
Keycloak		2018-03-28T07:53:10.673293+00:00	<a href="#">Revoke</a>

You have **denied** the following applications access to interact with your Bitbucket repositories.

By clicking **Remove**, you will remove this denial and will again be prompted for access the next time you use the application.

Name	Description
<i>You have not denied access to any applications</i>	

## OAuth consumers

Generate your own OAuth consumer key and secret to [build your own custom integration with Bitbucket](#).

[Add consumer](#)

Name	Description
You have no consumers configured.	

Click the `Add consumer` button.

*Register App*

## Add OAuth consumer

**Details**

Name *	Keycloak
Description	
Callback URL	ms/master/broker/bitbucket/endp
URL users will be redirected to after access authorization. Required for OAuth 2.	
URL	
Optional URL where users can learn more about your application.	
<input checked="" type="checkbox"/> This is a private consumer	
Installable applications that ship their OAuth consumer credentials as part of the application should not be marked as private.	

Copy the `Redirect URI` from the KeyCloak `Add Identity Provider` page and enter it into the Callback URL field on the Bitbucket Add OAuth Consumer page.

On the same page, mark the `Email` and `Read` boxes under `Account` to allow your application to read user email.

*Bitbucket App Page*

## OAuth consumers

Generate your own OAuth consumer key and secret to [build your own custom integration with Bitbucket](#).

[Add consumer](#)

Name	Description	...
▼ Keycloak		
<b>Key</b>	AgxwJL4yChLnpDNuB4	
<b>Secret</b>	n7QwZypgzC5ruTLCfsFCGB7tJ3X3txkt	

When you are done registering, click `Save`. This will open the application management page in Bitbucket. Find the client ID and secret from this page so you can enter them into the KeyCloak `Add identity provider` page. Click `Save`.

### 12.4.2. Facebook

There are a number of steps you have to complete to be able to enable login with Facebook. First, go to the `Identity Providers` left menu item and select `Facebook` from the `Add provider` drop down list. This will bring you to the `Add identity provider` page.

#### *Add Identity Provider*

The screenshot shows the Keycloak administration interface. The left sidebar is titled 'Master' and contains sections for 'Configure' (Realm Settings, Clients, Client Templates, Roles, Identity Providers), 'User Federation', 'Authentication', and 'Manage' (Groups, Users, Sessions, Events, Import). The 'Identity Providers' section is currently selected. The main content area is titled 'Add identity provider' and is specifically for adding a Facebook identity provider. It includes fields for 'Redirect URI' (set to 'http://localhost:8180/auth/realms/master/broker/facebook/endpoint'), 'Client ID' (marked with a red asterisk), 'Client Secret' (marked with a red asterisk), 'Default Scopes' (empty), 'Store Tokens' (OFF), 'Stored Tokens' (Readable OFF), 'Enabled' (ON), 'Disable User Info' (OFF), 'Trust Email' (OFF), 'GUI order' (empty), 'First Login Flow' (set to 'first broker login'), and 'Post Login Flow' (empty). At the bottom are 'Save' and 'Cancel' buttons.

You can't click save yet, as you'll need to obtain a `Client ID` and `Client Secret` from Facebook. One piece of data you'll need from this page is the `Redirect URI`. You'll have to provide that to Facebook when you register KeyCloak as a client there, so copy this URI to your clipboard.

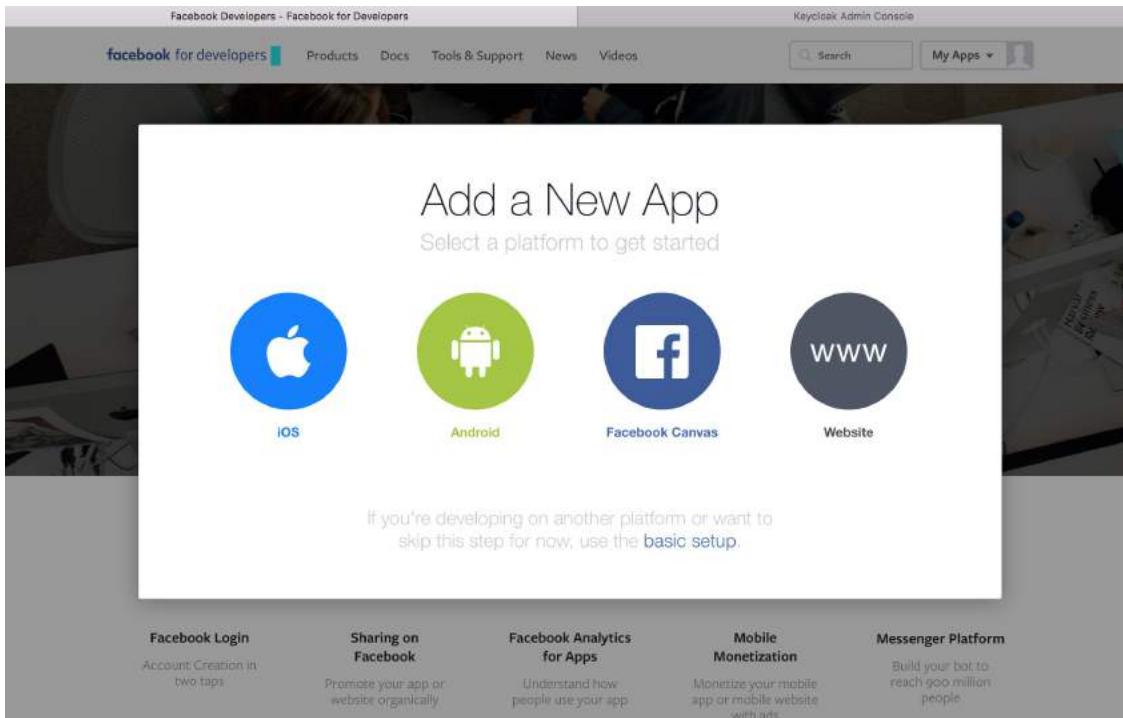
To enable login with Facebook you first have to create a project and a client in the [Facebook Developer Console](#).

Facebook often changes the look and feel of the Facebook Developer Console, so these directions might not always be up to date and the configuration steps might be slightly different.

Once you've logged into the console there is a pull down menu in the

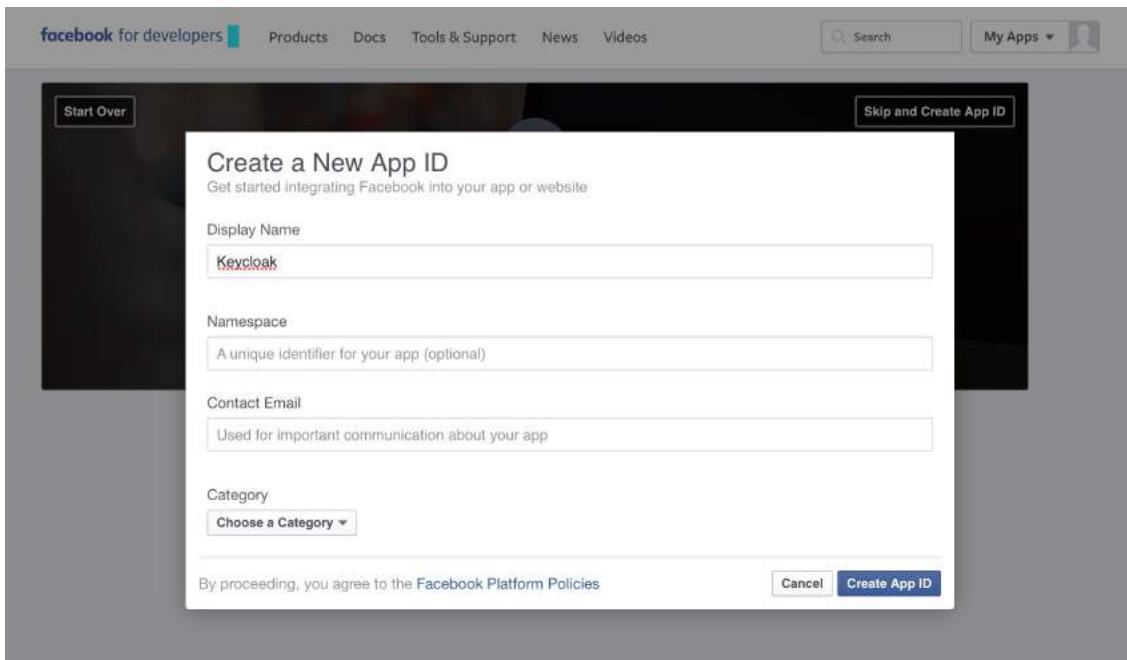
top right corner of the screen that says `My Apps`. Select the `Add a New App` menu item.

### *Add a New App*



Select the `Website` icon. Click the `Skip and Create App ID` button.

### *Create a New App ID*



The email address and app category are required fields. Once you're done with that, you will be brought to the dashboard for the application. Click the `Settings` left menu item.

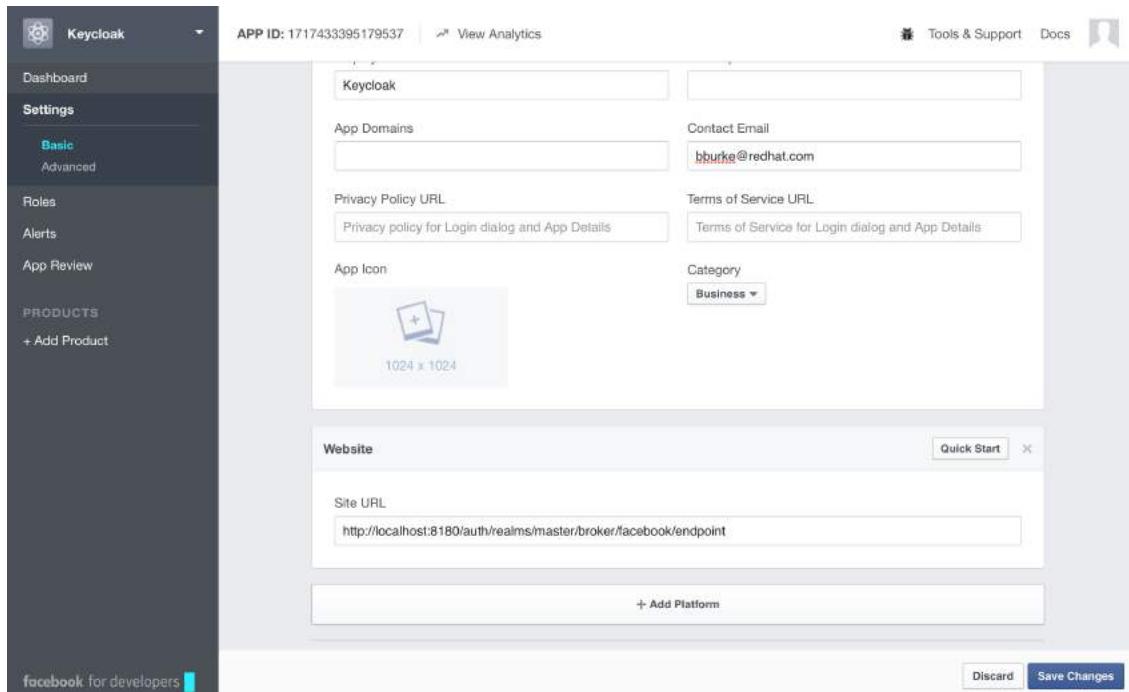
### Create a New App ID

A screenshot of the Facebook App Dashboard settings page for the 'Keycloak' app. The left sidebar shows the 'Settings' section with 'Basic' selected. The main area displays the app's basic information: App ID (171743395179537), App Secret (a7d28f77939d00b5d48ada04947d737e), Display Name (Keycloak), Namespace (empty), App Domains (empty), Contact Email (Used for important communication about your app), Privacy Policy URL (Privacy policy for Login dialog and App Details), Terms of Service URL (Terms of Service for Login dialog and App Details), App Icon (a placeholder icon), and Category (Business). At the bottom, there is a '+ Add Platform' button, a 'Discard' button, and a 'Save Changes' button.

Click on the `+ Add Platform` button at the end of this page and select

the `Website` icon. Copy and paste the `Redirect URI` from the Keycloak `Add identity provider` page into the `Site URL` of the Facebook `Website` settings block.

## Specify Website



After this it is necessary to make the Facebook app public. Click `App Review` left menu item and switch button to "Yes".

You will need also to obtain the App ID and App Secret from this page so you can enter them into the Keycloak `Add identity provider` page. To obtain this click on the `Dashboard` left menu item and click on `Show` under `App Secret`. Go back to Keycloak and specify those items and finally save your Facebook Identity Provider.

One config option to note on the `Add identity provider` page for Facebook is the `Default Scopes` field. This field allows you to manually specify the scopes that users must authorize when authenticating with this provider. For a complete list of scopes, please take a look at

<https://developers.facebook.com/docs/graph-api>. By default, KeyCloak uses the following scopes: `email`.

### 12.4.3. GitHub

There are a number of steps you have to complete to be able to enable login with GitHub. First, go to the `Identity Providers` left menu item and select `GitHub` from the `Add provider` drop down list. This will bring you to the `Add identity provider` page.

#### *Add Identity Provider*

The screenshot shows the Keycloak administration interface. The left sidebar has a dark theme with white text. The 'Identity Providers' option is highlighted with a blue bar at the top. The main content area has a light background. At the top, it says 'Identity Providers > Add identity provider'. Below that is a form titled 'Add identity provider'. The form fields are as follows:

- Redirect URI:** `http://localhost:8180/auth/realms/master/broker/github/endpoint`
- \* Client ID:** (empty input field)
- \* Client Secret:** (empty input field)
- Default Scopes:** (empty input field)
- Store Tokens:** `OFF` (radio button)
- Stored Tokens:** `OFF` (radio button)
- Readable:** (checkbox)
- Enabled:** `ON` (radio button)
- Disable User Info:** `OFF` (radio button)
- Trust Email:** `OFF` (radio button)
- GUI order:** (empty input field)
- First Login Flow:** `first broker login` (dropdown menu)
- Post Login Flow:** (empty input field)

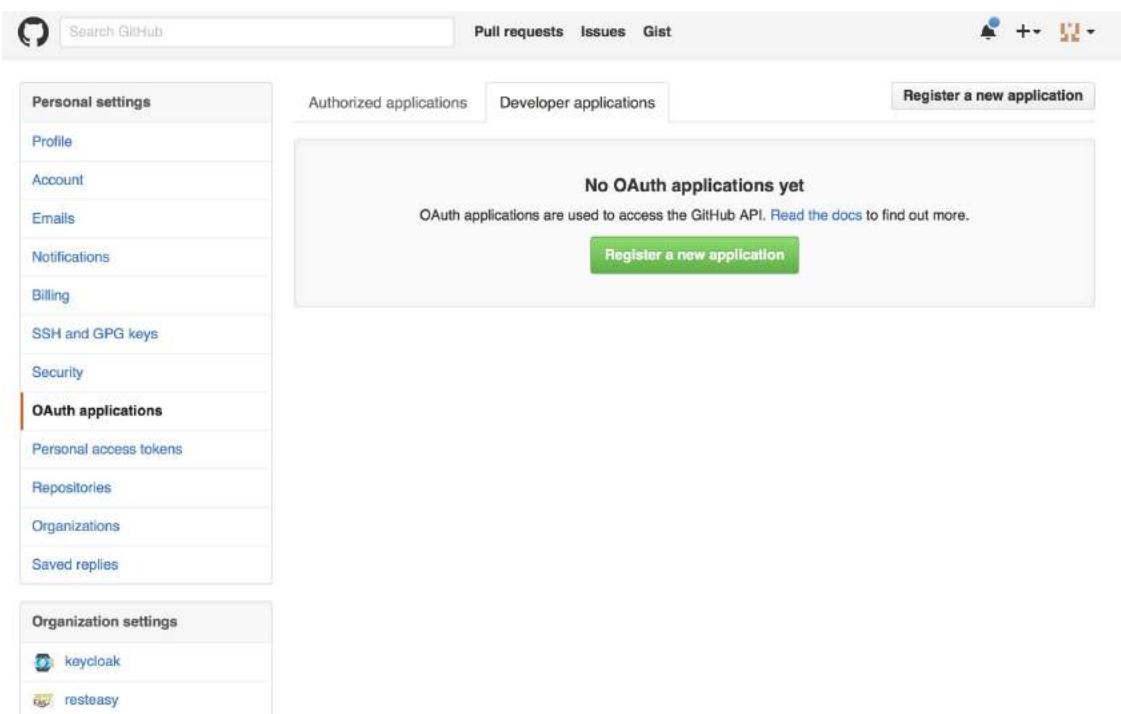
At the bottom right are two buttons: `Save` (blue) and `Cancel`.

You can't click `Save` yet, as you'll need to obtain a `Client ID` and `Client Secret` from GitHub. One piece of data you'll need from this page is the `Redirect URI`. You'll have to provide that to GitHub when you register KeyCloak as a client there, so copy this URI to your clipboard.

To enable login with GitHub you first have to register an application project in [GitHub Developer applications](#).

GitHub often changes the look and feel of application registration, so these directions might not always be up to date and the configuration steps might be slightly different.

## Add a New App



The screenshot shows the GitHub 'Personal settings' page. On the left, there's a sidebar with options like Profile, Account, Emails, Notifications, Billing, SSH and GPG keys, Security, and OAuth applications. The 'OAuth applications' option is currently selected. The main content area has tabs for 'Authorized applications' and 'Developer applications', with 'Developer applications' being active. A large box displays the message 'No OAuth applications yet' and a note that OAuth applications are used to access the GitHub API. It features a prominent green 'Register a new application' button.

Click the `Register a new application` button.

## Register App

The screenshot shows the GitHub 'Personal settings' page with the 'OAuth applications' section highlighted. On the left, there's a sidebar with links like 'Profile', 'Account', 'Emails', 'Notifications', 'Billing', 'SSH and GPG keys', 'Security', and 'OAuth applications'. Under 'OAuth applications', there are two registered apps: 'keycloak' and 'resteasy'. The main content area is titled 'Register a new OAuth application'. It has fields for 'Application name' (set to 'Keycloak'), 'Homepage URL' (empty), 'Application description' (empty), and 'Authorization callback URL' (set to 'http://localhost:8080/auth/realms/master/broker/github/endpoint'). At the bottom are 'Register application' and 'Cancel' buttons.

You'll have to copy the `Redirect URI` from the KeyCloak `Add Identity Provider` page and enter it into the `Authorization callback URL` field on the GitHub `Register a new OAuth application` page. Once you've completed this page you will be brought to the application's management page.

### *GitHub App Page*

The screenshot shows the Keycloak application configuration page. On the left, there is a sidebar with the following menu items:

- Profile
- Account
- Emails
- Notifications
- Billing
- SSH and GPG keys
- Security
- OAuth applications**
- Personal access tokens
- Repositories
- Organizations
- Saved replies

Below the sidebar, under "Organization settings", there are two entries:

- keycloak
- resteasy

The main content area is titled "Keycloak" and shows the following information:

- 0 users**
- Client ID**: d99e14593e09b145bbb2
- Client Secret**: 7d7681ed97a4a596bb6d257a785b5f80110c5506
- Revoke all user tokens** | **Reset client secret**
- A placeholder for an application logo with the text "Drag & drop" and "or choose an image".
- Application name**: Keycloak
- Description**: Something users will recognize and trust
- Homepage URL**: `http://localhost:8080/auth/realms/master/`
- Application description**: Application description is optional
- This is displayed to all potential users of your application

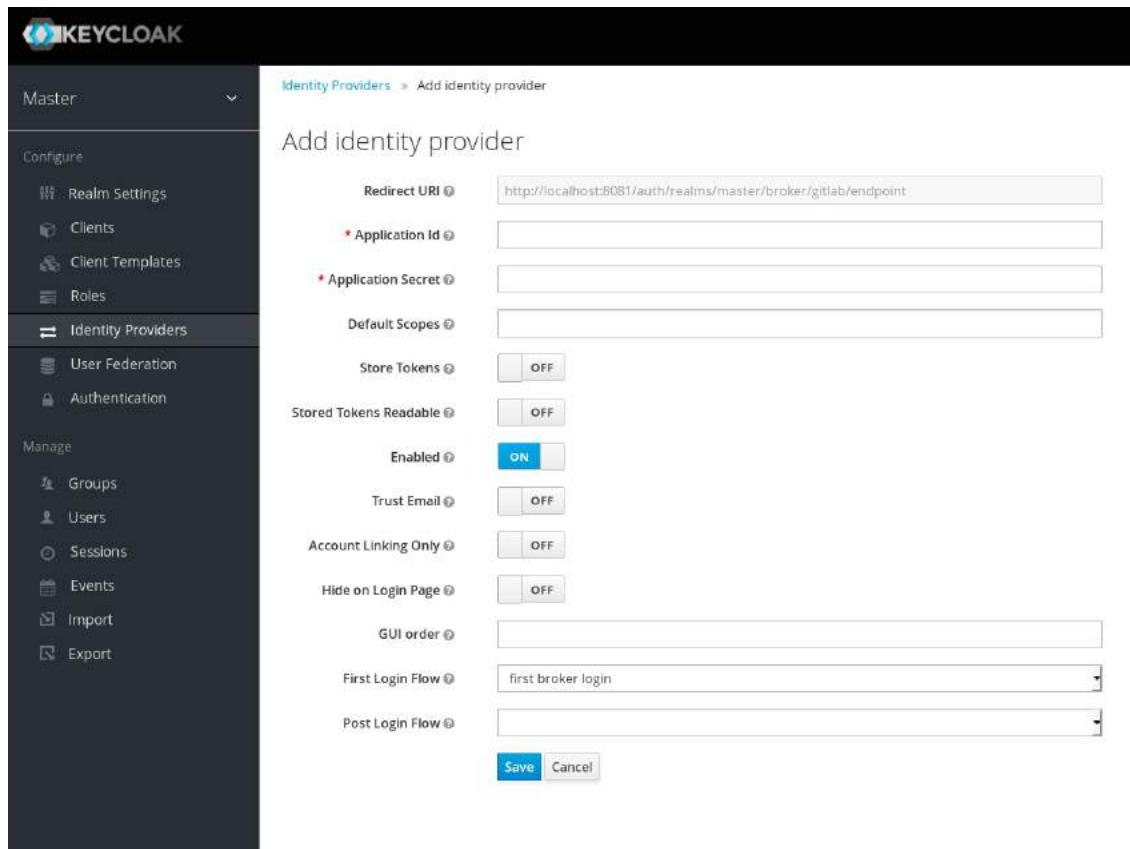
You will need to obtain the client ID and secret from this page so you can enter them into the KeyCloak Add identity provider page. Go back to KeyCloak and specify those items.

#### 12.4.4. GitLab

There are a number of steps you have to complete to be able to enable login with GitLab.

First, go to the `Identity Providers` left menu item and select `GitLab` from the `Add provider` drop down list. This will bring you to the `Add identity provider` page.

#### *Add Identity Provider*



Before you can click `Save`, you must obtain a `Client ID` and `client Secret` from GitLab.

You will use the `Redirect URI` from this page in a later step, which you will provide to GitLab when you register KeyCloak as a client there.

To enable login with GitLab you first have to register an application in [GitLab as OAuth2 authentication service provider](#).

GitLab often changes the look and feel of application registration, so what you see on the GitLab site may differ. If in doubt, see the GitLab documentation.

## Add a New App

The screenshot shows the 'User Settings' sidebar with 'Applications' selected. The main area displays the 'Applications' section, which manages OAuth providers and authorized applications. A form for 'Add new application' is shown, with 'Name' set to 'Keycloak' and 'Redirect URI' set to 'http://localhost:8081/auth/realm/master/broker/gitlab/endpoint'. Below the form, a note says 'Use one line per URI'. Under 'Scopes', several options are listed: 'api' (selected), 'read\_user', 'sudo', 'read\_registry', and 'openid'. A note for 'openid' states: 'The ability to authenticate using GitLab, and read-only access to the user's profile information and group memberships'. At the bottom right is a green 'Save application' button.

Copy the **Redirect URI** from the **Keycloak Add Identity Provider** page and enter it into the **Redirect URI** field on the **GitLab Add new application** page.

## GitLab App Page

The screenshot shows the 'User Settings' sidebar with 'Applications' selected. The main area displays the 'Edit application' form for 'Keycloak'. The 'Name' field is filled with 'Keycloak', and the 'Redirect URI' field contains 'http://localhost:8081/auth/realm/master/broker/gitlab/endpoint'. Below the form, a note says 'Use one line per URI'. Under 'Scopes', several options are listed: 'api' (selected), 'read\_user', 'sudo', 'read\_registry', and 'openid' (selected). A note for 'openid' states: 'The ability to authenticate using GitLab, and read-only access to the user's profile information and group memberships'. At the bottom right is a green 'Save application' button.

When you are done registering, click **Save application**. This will

open the application management page in GitLab. Find the client ID and secret from this page so you can enter them into the Keycloak `Add identity provider` page.

To finish, return to Keycloak and enter them. Click `Save`.

#### 12.4.5. Google

There are a number of steps you have to complete to be able to enable login with Google. First, go to the `Identity Providers` left menu item and select `Google` from the `Add provider` drop down list. This will bring you to the `Add identity provider` page.

*Add Identity Provider*

The screenshot shows the Keycloak Admin Console interface. The left sidebar is dark-themed with white text and icons. It has two main sections: 'Configure' and 'Manage'. Under 'Configure', the 'Identity Providers' option is selected and highlighted with a blue border. Other options include 'Realm Settings', 'Clients', 'Client Templates', 'Roles', 'User Federation', and 'Authentication'. Under 'Manage', there are links for 'Groups', 'Users', 'Sessions', 'Events', 'Import', and 'Export'. The main content area is titled 'Add identity provider' and is specifically for adding a Google identity provider. It contains several input fields and toggle switches. The 'Redirect URI' field is pre-filled with 'http://localhost:8081/auth/realms/master/broker/google/endpoint'. The 'Client ID' and 'Client Secret' fields are marked with red asterisks, indicating they are required. Other fields include 'Hosted Domain', 'Use userip Param', 'Default Scopes', 'Store Tokens' (set to OFF), 'Stored Tokens' (set to OFF and 'Readable' is checked), 'Enabled' (set to ON), 'Disable User Info' (set to OFF), 'Trust Email' (set to OFF), 'Account Linking Only' (set to OFF), 'Hide on Login Page' (set to OFF), 'GUI order' (empty input field), 'First Login Flow' (set to 'first broker login'), and 'Post Login Flow' (empty input field). At the bottom right are 'Save' and 'Cancel' buttons.

You can't click save yet, as you'll need to obtain a `Client ID` and `Client Secret` from Google. One piece of data you'll need from this page is the `Redirect URI`. You'll have to provide that to Google when you register KeyCloak as a client there, so copy this URI to your clipboard.

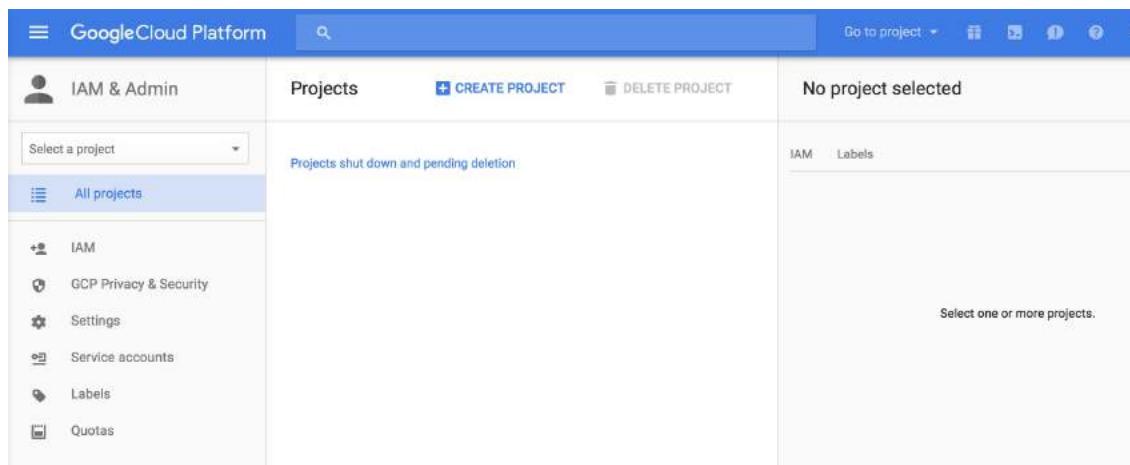
To enable login with Google you first have to create a project and a client in the [Google Developer Console](#). Then you need to copy the client ID and secret into the KeyCloak Admin Console.

Google often changes the look and feel of the Google Developer Console, so these directions might not always be up to date and the configuration steps might be slightly different.

Let's see first how to create a project with Google.

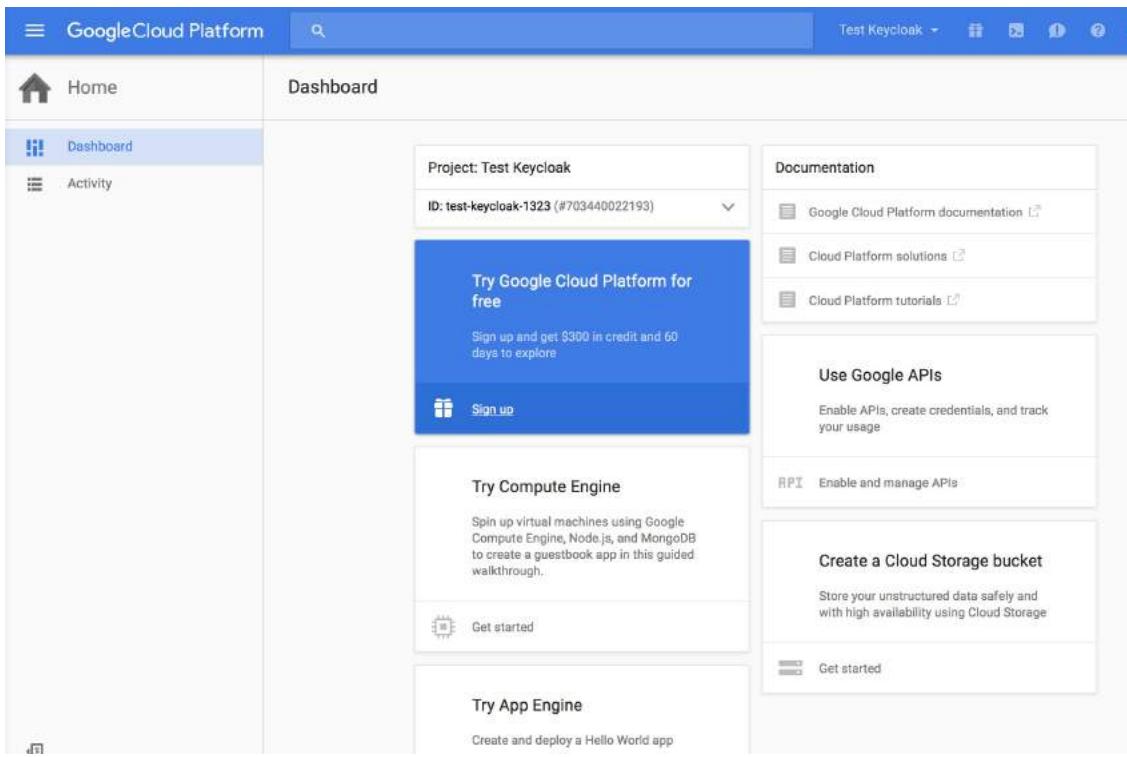
Log in to the [Google Developer Console](#).

### *Google Developer Console*



Click the `Create Project` button. Use any value for `Project name` and `Project ID` you want, then click the `Create` button. Wait for the project to be created (this may take a while). Once created you will be brought to the project's dashboard.

### *Dashboard*



Then navigate to the `APIs & Services` section in the Google Developer Console. On that screen, navigate to `Credentials` administration.

When users log into Google from KeyCloak they will see a consent screen from Google which will ask the user if KeyCloak is allowed to view information about their user profile. Thus Google requires some basic information about the product before creating any secrets for it. For a new project, you have first to configure `OAuth consent screen`.

For the very basic setup, filling in the Application name is sufficient. You can also set additional details like scopes for Google APIs in this page.

*Fill in OAuth consent screen details*

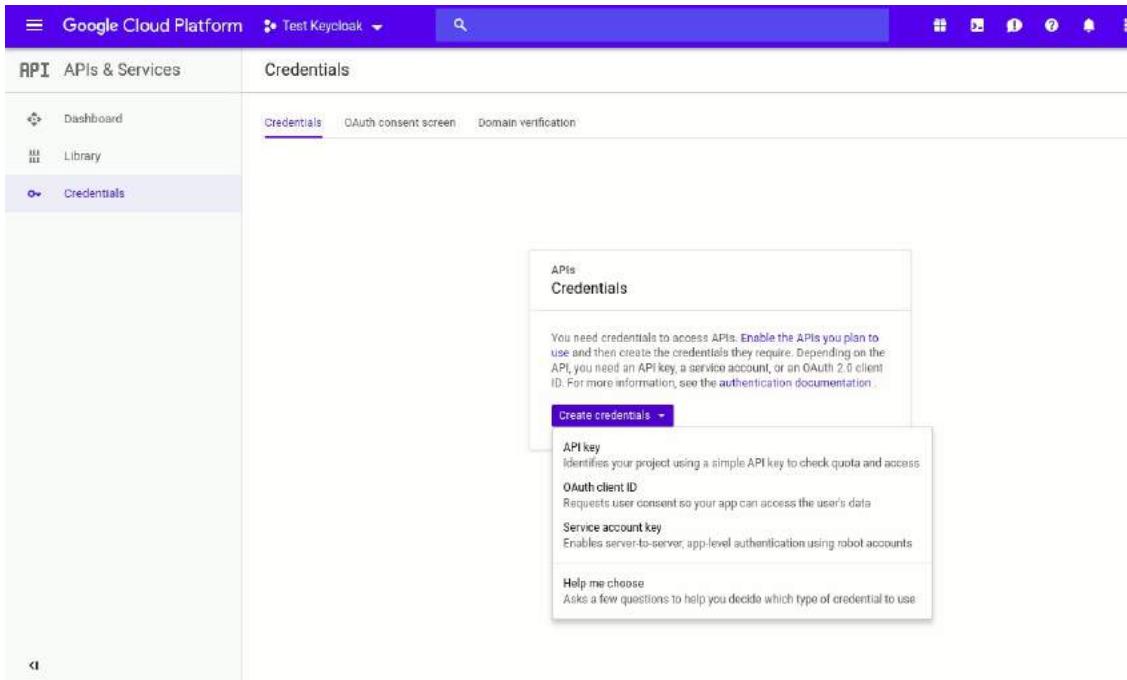
The screenshot shows the Google Cloud Platform API & Services Credentials page for a project named "Test Keycloak". The "OAuth consent screen" tab is selected. The page displays configuration options for an OAuth application, including:

- Application type:** Public (Any Google Account can grant access to the scopes required by this app. Learn more about scopes)
- Internal:** Only users with a Google Account in your organization can grant access to the scopes requested by this app.
- Verification status:** Not published
- Application name:** Test Keycloak
- Application logo:** A placeholder image showing a red "X" inside a circle.
- Support email:** Shown on the consent screen for user support.
- Scopes for Google APIs:** email, profile, openid
- Authorized domains:** example.com
- Application Homepage link:** https:// or http://
- Application Privacy Policy link:** https:// or http://
- Application Terms of Service link (Optional):** https:// or http://

At the bottom, there are three buttons: Save, Submit for verification, and Cancel.

The next step is to create OAuth client ID and client secret. Back in **Credentials** administration, navigate to **Credentials** tab and select **OAuth client ID** under the **Create credentials** button.

### *Create credentials*



You will then be brought to the `Create OAuth client ID` page. Select `Web application` as the application type. Specify the name you want for your client. You'll also need to copy and paste the `Redirect URI` from the KeyCloak `Add Identity Provider` page into the `Authorized redirect URIs` field. After you do this, click the `Create` button.

### *Create OAuth client ID*

Google Cloud Platform Test Keycloak

Create OAuth client ID

For applications that use the OAuth 2.0 protocol to call Google APIs, you can use an OAuth 2.0 client ID to generate an access token. The token contains a unique identifier. See Setting up OAuth 2.0 for more information.

**Application type**

- Web application
- Android Learn more
- Chrome App Learn more
- iOS Learn more
- Other

**Name**

**Restrictions**  
Enter JavaScript origins, redirect URLs, or both [Learn More](#).  
Origins and redirect domains must be added to the list of Authorized Domains in the [OAuth consent settings](#).

**Authorized JavaScript origins**  
For use with requests from a browser. This is the origin URI of the client application. It can't contain a wildcard (`http://example.com`) or a path (`https://example.com/subdir`). If you're using a nonstandard port, you must include it in the origin URL.

**Authorized redirect URIs**  
For use with requests from a web server. This is the path in your application that users are redirected to after they have authenticated with Google. The path will be appended with the authorization code for access. Must have a protocol. Cannot contain URL fragments or relative paths. Cannot be a public IP address.

**Create** **Cancel**

After you click **Create** you will be brought to the **Credentials** page. Click on your new OAuth 2.0 Client ID to view the settings of your new Google Client.

## Google Client Credentials

Google Cloud Platform Test Keycloak

**API Manager** **Credentials**

**Overview** **Credentials**

**Client ID for Web application**

Client ID	703440022193-bf3hdk60fcfg7pjg9kq9oduq1p28aufa.apps.googleusercontent.com
Client secret	ja-tbOqsKYLoVi5PluTxI4r2
Creation date	May 26, 2016, 11:50:48 AM

**Name**

**Restrictions**  
Enter JavaScript origins, redirect URLs, or both [Learn More](#).

**Authorized JavaScript origins**  
For use with requests from a browser. This is the origin URI of the client application. It can't contain a wildcard (`http://example.com`) or a path (`http://example.com/subdir`). If you're using a nonstandard port, you must include it in the origin URL.

**Authorized redirect URIs**  
For use with requests from a web server. This is the path in your application that users are redirected to after they have authenticated with Google. The path will be appended with the authorization code for access. Must have a protocol. Cannot contain URL fragments or relative paths. Cannot be a public IP address.

**Save** **Cancel**

You will need to obtain the client ID and secret from this page so you can enter them into the KeyCloak `Add identity provider` page. Go back to KeyCloak and specify those items.

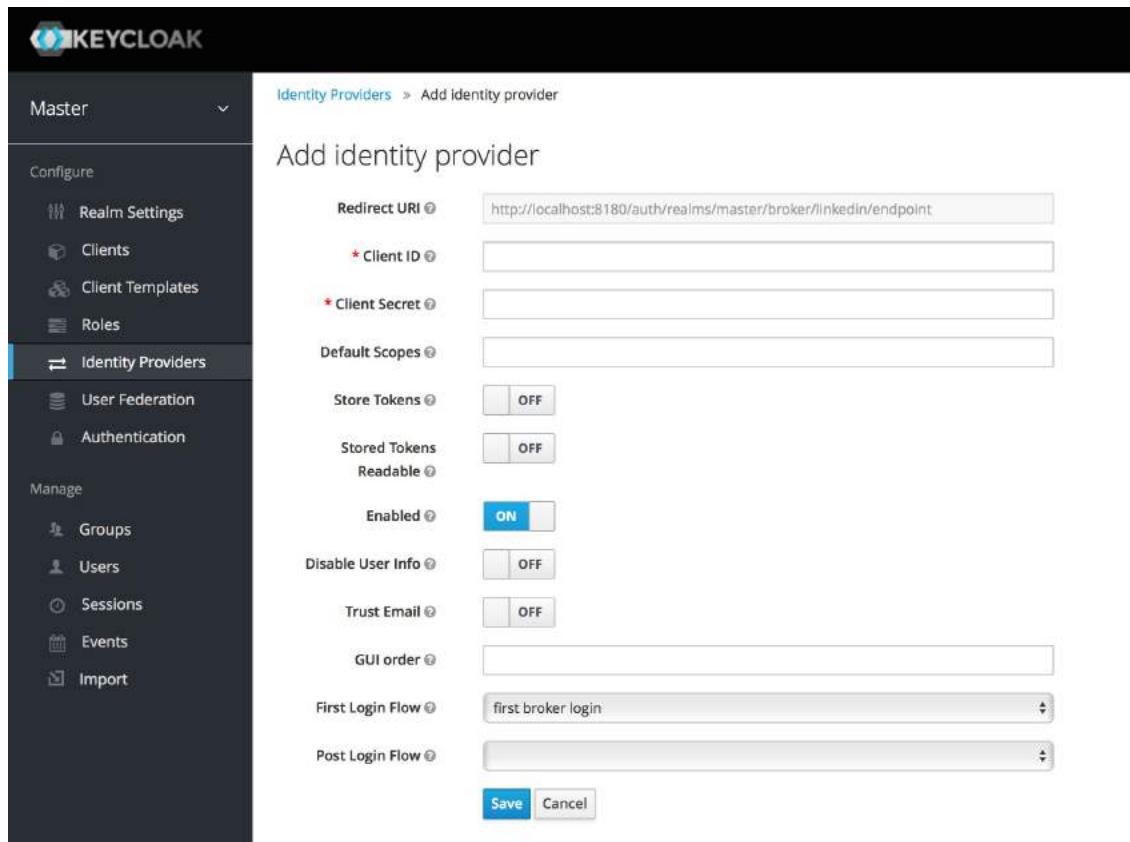
One config option to note on the `Add identity provider` page for Google is the `Default Scopes` field. This field allows you to manually specify the scopes that users must authorize when authenticating with this provider. For a complete list of scopes, please take a look at <https://developers.google.com/oauthplayground/>. By default, KeyCloak uses the following scopes: `openid profile email`.

If your organization uses the G Suite and you want to restrict access to only members of your organization, you must enter the domain that is used for the G Suite into the `Hosted Domain` field to enable it.

#### 12.4.6. LinkedIn

There are a number of steps you have to complete to be able to enable login with LinkedIn. First, go to the `Identity Providers` left menu item and select `LinkedIn` from the `Add provider` drop down list. This will bring you to the `Add identity provider` page.

#### *Add Identity Provider*

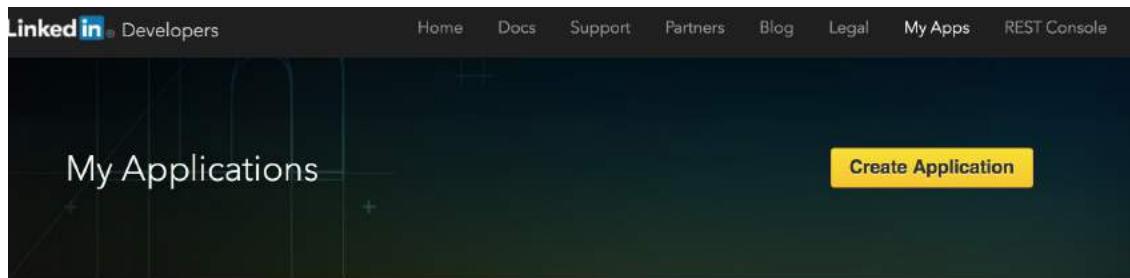


You can't click save yet, as you'll need to obtain a `Client ID` and `Client Secret` from LinkedIn. One piece of data you'll need from this page is the `Redirect URI`. You'll have to provide that to LinkedIn when you register KeyCloak as a client there, so copy this URI to your clipboard.

To enable login with LinkedIn you first have to create an application in [LinkedIn Developer Network](#).

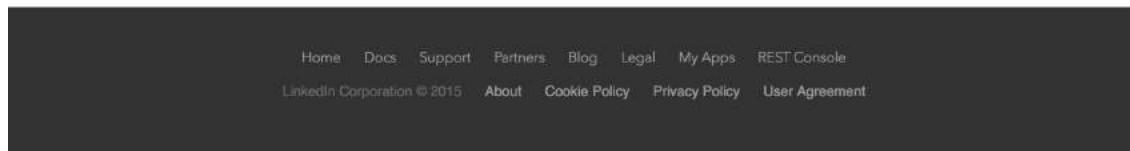
LinkedIn may change the look and feel of application registration, so these directions may not always be up to date.

*Developer Network*



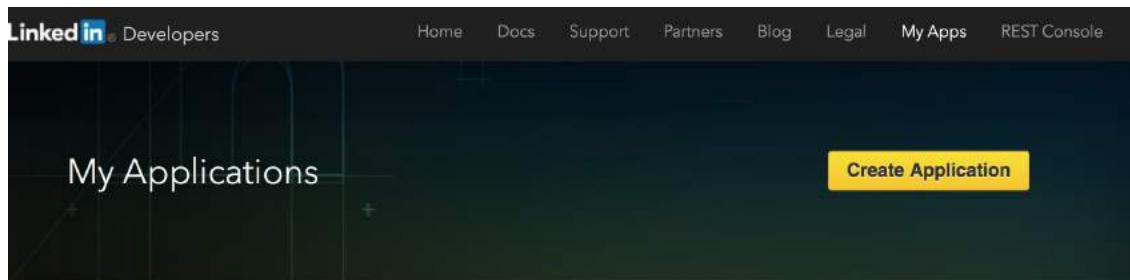
Manage your desktop and mobile applications that leverage LinkedIn APIs.

You do not have any applications.



Click on the **Create Application** button. This will bring you to the **Create a New Application Page**.

*Create App*



### Create a New Application

**Company Name:**\*

Red Hat

**Name:**\*

Keycloak

**Description:**\*

Keycloak

**Application Logo:**\*



Select File to Upload

**Application Use:**\*

Fill in the form with the appropriate values, then click the **Submit** button. This will bring you to the new application's settings page.

### *App Settings*

Authentication

Settings  
Roles  
Mobile  
JavaScript  
OS  
Usage & Limits  
My Applications

Authentication Keys

Client ID: 77omcv5r0oip1u

Client Secret: Zvf487WHpNL5ZwJb

Default Application Permissions

r\_basicprofile     r\_emailaddress     w\_share     rw\_company\_admin

OAuth 2.0

Authorized Redirect URLs:

http://localhost:8180/auth/realm/master/broker/linkedin/endpoint Add

OAuth 1.0a

Default "Accept" Redirect URL:

Default "Cancel" Redirect URL:

Select `r_basicprofile` and `r_emailaddress` in the `Default Application Permissions` section. You'll have to copy the `Redirect URI` from the KeyCloak `Add Identity Provider` page and enter it into the `OAuth 2.0 Authorized Redirect URLs` field on the LinkedIn app settings page. Don't forget to click the `Update` button after you do this!

You will then need to obtain the client ID and secret from this page so you can enter them into the KeyCloak `Add identity provider` page. Go back to KeyCloak and specify those items.

#### 12.4.7. Microsoft

There are a number of steps you have to complete to be able to enable login with Microsoft. First, go to the `Identity Providers` left menu item and select `Microsoft` from the `Add provider` drop down list.

This will bring you to the `Add identity provider` page.

## Add Identity Provider

The screenshot shows the Keycloak administration interface. The left sidebar is titled 'Master' and contains sections for 'Configure' (Realm Settings, Clients, Client Templates, Roles, Identity Providers), 'Manage' (Groups, Users, Sessions, Events, Import), and a 'Logout' button. The 'Identity Providers' section is currently selected. The main content area is titled 'Add identity provider'. It includes fields for 'Redirect URI' (set to `http://localhost:8180/auth/realms/master/broker/microsoft/endpoint`), 'Client ID' (empty), 'Client Secret' (empty), 'Default Scopes' (empty), 'Store Tokens' (OFF), 'Stored Tokens Readable' (OFF), 'Enabled' (ON), 'Disable User Info' (OFF), 'Trust Email' (OFF), 'GUI order' (empty), 'First Login Flow' (set to 'first broker login'), and 'Post Login Flow' (empty). At the bottom are 'Save' and 'Cancel' buttons.

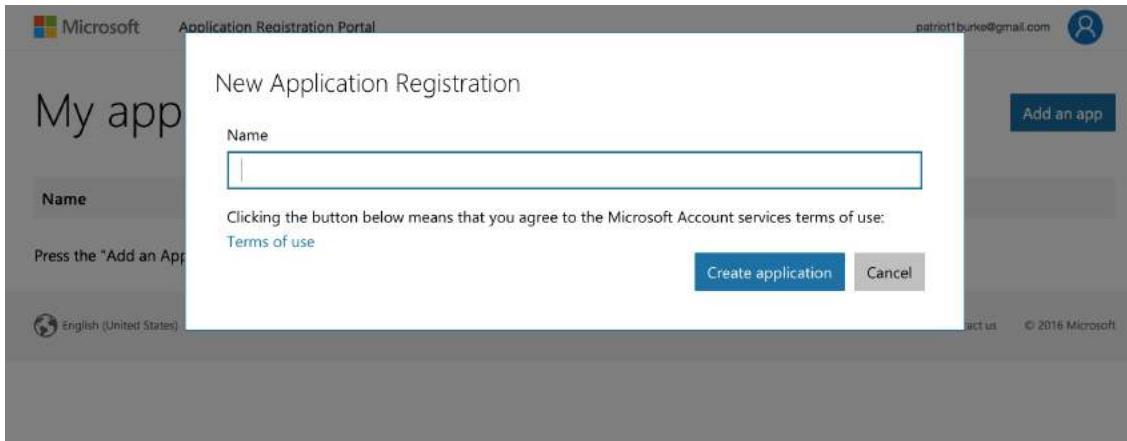
You can't click `Save` yet, as you'll need to obtain a `Client ID` and `Client Secret` from Microsoft. One piece of data you'll need from this page is the `Redirect URI`. You'll have to provide that to Microsoft when you register KeyCloak as a client there, so copy this URI to your clipboard.

To enable login with Microsoft account you first have to register an OAuth application at Microsoft. Go to the [Microsoft Application Registration url](#).

Microsoft often changes the look and feel of applicati-

on registration, so these directions might not always be up to date and the configuration steps might be slightly different.

## Register Application



Enter in the application name and click **Create application**. This will bring you to the application settings page of your new application.

## Settings

Application ID  
000000004019746A

### Application Secrets Learn More

[Generate New Password](#)

ALXZqddrh8U5aVS2sSOGFA

Version 0

Current

### Platforms

[Add Platform](#)

Web

[Delete](#)

- Allow Implicit Flow
- Restrict token issuing to this app

Limits the issuing of JSON Web Tokens (JWT) for your domain to exclusively this application.

#### Target Domain

This is the domain that other apps will use when they request a JWT for your app on Windows (such as www.contoso.com).

Target Domain

[Redirect URLs](#) [Add Url](#)

Enter a url

[Click here for help integrating your application with Microsoft.](#)

You'll have to copy the `Redirect URI` from the KeyCloak `Add Identity Provider` page and add it to the `Redirect URIs` field on the Microsoft application page. Be sure to click the `Add Url` button and `Save` your changes.

Finally, you will need to obtain the Application ID and secret from this page so you can enter them back on the KeyCloak `Add identity provider` page. Go back to KeyCloak and specify those items.

#### 12.4.8. OpenShift

OpenShift Online is currently in the developer preview mode. This documentation has been based on on-premise installations and local `minishift` development environment.

There are just a few steps you have to complete to be able to enable login with OpenShift. First, go to the `Identity Providers` left menu item and select `OpenShift` from the `Add provider` drop down list. This will bring you to the `Add identity provider` page.

##### *Add Identity Provider*

## Add identity provider

Redirect URI

\* Client ID

\* Client Secret

\* Base URL

Default Scopes

Store Tokens  OFF

Stored Tokens Readable  OFF

Enabled  ON

Disable User Info  OFF

Trust Email  OFF

GUI order

First Login Flow

Post Login Flow

### Registering OAuth client

You can register your client using `oc` command line tool.

```
$ oc create -f <(echo '  
kind: OAuthClient  
apiVersion: v1  
metadata:  
  name: kc-client (1)  
  secret: "..." (2)  
  redirectURIs:  
    - "http://www.example.com/" (3)  
  grantMethod: prompt (4)  
' )
```

The `name` of your OAuth client. Passed as `client_id`

- 1 request parameter when making requests to `<openshift_master>/oauth/authorize` and `<openshift_master>/oauth/token`.
- 2 `secret` is used as the `client_secret` request parameter.

The `redirect_uri` parameter specified in requests to `<openshift_master>/oauth/authorize` and `<openshift_master>/oauth/token` must be equal to (or prefixed by) one of the URIs in `redirectURIs`.
- 3 The `grantMethod` is used to determine what action to take when this client requests tokens and has not yet been granted access by the user.

Use client ID and secret defined by `oc create` command to enter them back on the KeyCloak `Add identity provider` page. Go back to KeyCloak and specify those items.

Please refer to [official OpenShift documentation](#) for more detailed guides.

#### 12.4.9. PayPal

There are a number of steps you have to complete to be able to enable login with PayPal. First, go to the `Identity Providers` left menu item and select `PayPal` from the `Add provider` drop down list. This will bring you to the `Add identity provider` page.

#### *Add Identity Provider*

## Add identity provider

Redirect URI ⓘ	http://localhost:8081/auth/realms/master/broker/paypal/endpoint
* Client ID ⓘ	<input type="text"/>
* Client Secret ⓘ	<input type="text"/>
Target Sandbox ⓘ	<input checked="" type="checkbox"/> OFF
Default Scopes ⓘ	<input type="text"/>
Store Tokens ⓘ	<input checked="" type="checkbox"/> OFF
Stored Tokens Readable ⓘ	<input checked="" type="checkbox"/> OFF
Enabled ⓘ	<input checked="" type="checkbox"/> ON
Disable User Info ⓘ	<input checked="" type="checkbox"/> OFF
Trust Email ⓘ	<input checked="" type="checkbox"/> OFF
Account Linking Only ⓘ	<input checked="" type="checkbox"/> OFF
Hide on Login Page ⓘ	<input checked="" type="checkbox"/> OFF
GUI order ⓘ	<input type="text"/>
First Login Flow ⓘ	first broker login
Post Login Flow ⓘ	<input type="text"/>
<input type="button" value="Save"/> <input type="button" value="Cancel"/>	

You can't click save yet, as you'll need to obtain a `Client ID` and `Client Secret` from PayPal. One piece of data you'll need from this page is the `Redirect URI`. You'll have to provide that to PayPal when you register KeyCloak as a client there, so copy this URI to your clipboard.

To enable login with PayPal you first have to register an application project in [PayPal Developer applications](#).

*Add a New App*

The screenshot shows the 'REST API apps' section of the PayPal Developer dashboard. On the left, there's a sidebar with 'Dashboard', 'My Apps & Credentials', 'Sandbox' (selected), 'Accounts', and 'Notifications'. The main area has a heading 'REST API apps' and a note: 'Create an app to receive REST API credentials for testing and live transactions.' Below this is a box containing the note: 'Note: Features available for live transactions are listed in your account eligibility.' A blue 'Create App' button is located at the bottom right of this box.

Click the `Create App` button.

## *Register App*

The screenshot shows the 'Create New App' page. The sidebar includes 'Dashboard', 'My Apps & Credentials', 'Sandbox' (selected), 'Accounts', 'Notifications', 'API Calls', 'IPN Simulator', 'Webhooks Events', 'Mock', and 'Live'. The main area has a heading 'Create New App' and a note: 'Create an app to receive REST API credentials for testing and live transactions.' It features an 'Application Details' section with 'App Name' set to 'Keycloak', 'Sandbox developer account' set to 'test.user-facilitator@test.com (NO)', and a reminder about business purpose. At the bottom is a blue 'Create App' button.

You will now be brought to the app settings page.

## *Do the following changes*

- Choose to configure either Sandbox or Live (choose Live if you haven't enabled the `Target Sandbox` switch on the `Add identity provider` page)
- Copy Client ID and Secret so you can paste them into the KeyCloak `Add identity provider` page.
- Scroll down to `App Settings`
- Copy the `Redirect URI` from the KeyCloak `Add Identity`

`Provider` page and enter it into the `Return URL` field.

- Check the `Log In with PayPal` checkbox.
- Check the `Full name` checkbox under the personal information section.
- Check the `Email address` checkbox under the address information section.
- Add both a privacy and a user agreement URL pointing to the respective pages on your domain.

#### 12.4.10. Stack Overflow

There are a number of steps you have to complete to be able to enable login with Stack Overflow. First, go to the `Identity Providers` left menu item and select `Stack Overflow` from the `Add provider` drop down list. This will bring you to the `Add identity provider` page.

#### *Add Identity Provider*

The screenshot shows the Keycloak administration interface. The left sidebar is titled 'Master' and contains the following navigation items under 'Configure': 'Realm Settings', 'Clients', 'Client Templates', 'Roles', 'Identity Providers' (which is highlighted in blue), 'User Federation', and 'Authentication'. Under 'Manage', there are links for 'Groups', 'Users', 'Sessions', 'Events', and 'Import'. The main content area is titled 'Add identity provider' and shows the configuration for a new provider. The fields are as follows:

- Redirect URI**: http://localhost:8180/auth/realm/master/broker/stackoverflow/endpoint
- \* Client ID**: (empty input field)
- \* Client Secret**: (empty input field)
- Key**: (empty input field)
- Default Scopes**: (empty input field)
- Store Tokens**: OFF (switch)
- Stored Tokens**: OFF (switch)  
  **Readable**: OFF (switch)
- Enabled**: ON (switch)
- Disable User Info**: OFF (switch)
- Trust Email**: OFF (switch)
- GUI order**: (empty input field)
- First Login Flow**: first broker login (dropdown menu)
- Post Login Flow**: (empty input field)

At the bottom right are 'Save' and 'Cancel' buttons.

To enable login with Stack Overflow you first have to register an OAuth application on [StackApps](#). Go to [registering your application on Stack Apps](#) URL and login.

Stack Overflow often changes the look and feel of application registration, so these directions might not always be up to date and the configuration steps might be slightly different.

## *Register Application*

The screenshot shows the 'Register Your V2.0 Application' form on the Stack Apps website. The top navigation bar includes links for 'Questions', 'Tags', 'Users', 'Badges', 'Unanswered', and 'Ask Question'. On the left, there's a sidebar with sections for 'Application Name', 'Description', 'OAuth Domain', 'Application Website', 'Application Icon (optional)', and a note about hosting the icon on a Stack Exchange Imgur account. On the right, there's a 'Why Register?' section with explanatory text and links for help and API promotion. At the bottom, a blue button labeled 'Register Your Application' is visible.

StackExchange ▾ 1 help ▾ Search Q&A

stackapps

Questions Tags Users Badges Unanswered Ask Question

Register Your V2.0 Application

Application Name  
Be Unique! Avoid implying an official Stack Exchange relationship.

Description

OAuth Domain  
example.com, subdomains will be automatically whitelisted

Application Website  
Where users can go to read about your application

Application Icon (optional)  
Must be hosted by the Stack Exchange Imgur account  Enable Client Side OAuth Flow

**Why Register?**

Because it's the neighborly thing to do. We like to know who is using our API, and how, so we can have the metrics we need to support your application and improve the API together.

Once it's ready for public consumption, we'll [help you promote your registered application](#) here on Stack Apps.

Upon registering, you'll be provided an API key which grants your app a **much** larger per-day request quota than using the API anonymously.

You'll also receive parameters for [authenticating users via OAuth 2.0](#).

**Register Your Application**

Enter in the application name and the OAuth Domain Name of your application and click **Register your Application**. Type in anything you want for the other items.

## *Settings*



Questions Tags Users Badges Unanswered Ask Question

Keycloak

**Client Id**

7209

This ID identifies your application to the Stack Exchange API. Your application client id is **not secret**, and may be safely embeded in distributed binaries.

Pass this as `client_id` in our OAuth 2.0 flow.

**Client Secret (reset)**

A8M5pezJvp9G)Nfx6aw9A((

Pass this as `client_secret` in our OAuth 2.0 flow if your app uses the explicit path.

This **must be** kept secret. Do not embed it in client side code or binaries you intend to distribute. If you need client side authentication, use the implicit OAuth 2.0 flow.

**Key**

sZA2ICUcqAr6ZkBkps4w((

Pass this as `key` when making requests against the Stack Exchange API to receive a [higher request quota](#).

This is not considered a secret, and may be safely embed in client side code or distributed binaries.

**Description**

Keycloak

This **text-only** blurb will be shown to users during authentication.

**OAuth Domain**

More Info

- [Authentication Statistics](#)
- [API Documentation](#)

Finally, you will need to obtain the client ID, secret, and key from this page so you can enter them back on the KeyCloak `Add identity provider` page. Go back to KeyCloak and specify those items.

### 12.4.11. Twitter

There are a number of steps you have to complete to be able to enable login with Twitter. First, go to the `Identity Providers` left menu item and select `Twitter` from the `Add provider` drop down list. This will bring you to the `Add identity provider` page.

#### *Add Identity Provider*

The screenshot shows the Keycloak administration interface. The left sidebar is titled 'Master' and contains sections for 'Configure' (Realm Settings, Clients, Client Templates, Roles), 'Identity Providers' (selected), 'User Federation', 'Authentication', and 'Manage' (Groups, Users, Sessions, Events, Import). The main content area is titled 'Add identity provider' and is specifically for 'Identity Providers'. It includes fields for 'Redirect URI' (set to 'http://localhost:8180/auth/realms/master/broker/twitter/endpoint'), 'Client ID' (marked with a red asterisk), 'Client Secret' (marked with a red asterisk), 'Default Scopes' (empty), and several toggle switches: 'Store Tokens' (OFF), 'Stored Tokens Readable' (OFF), 'Enabled' (ON), 'Disable User Info' (OFF), 'Trust Email' (OFF), and 'GUI order' (empty). Below these are dropdown menus for 'First Login Flow' (set to 'first broker login') and 'Post Login Flow' (empty). At the bottom are 'Save' and 'Cancel' buttons.

You can't click save yet, as you'll need to obtain a `Client ID` and `Client Secret` from Twitter. One piece of data you'll need from this page is the `Redirect URI`. You'll have to provide that to Twitter when you register KeyCloak as a client there, so copy this URI to your clipboard.

To enable login with Twitter you first have to create an application in the [Twitter Application Management](#).

### *Register Application*



## Status

Your application has been successfully deleted.

## Twitter Apps

You don't currently have any Twitter Apps.

[Create New App](#)

Click on the `Create New App` button. This will bring you to the `Create an Application` page.

### *Register Application*



## Create an application

### Application Details

**Name \***Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.**Description \***Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.**Website \***Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL is used in the source attribution for tweets created by your application and will be shown in user-facing authorization screens.  
(If you don't have a URL yet, just put a placeholder here but remember to change it later.)**Callback URL**Where should we return after successfully authenticating? OAuth 1.0a applications should explicitly specify their oauth\_callback URL on the request token step, regardless of the value given here. To restrict your application from using callbacks, leave this field blank.

Enter in a Name and Description. The Website can be anything, but cannot have a `localhost` address. For the `Callback URL` you must copy the `Redirect URI` from the KeyCloak Add Identity Provi-

der page.

You cannot use `localhost` in the `Callback URL`. Instead replace it with `127.0.0.1` if you are trying to test drive Twitter login on your laptop.

After clicking save you will be brought to the `Details` page.

## *App Details*

The screenshot shows the 'App Details' page for a Twitter application named 'keycloak2'. At the top, there's a green banner with the message: 'Your application has been created. Please take a moment to review and adjust your application's settings.' Below the banner, the application name 'keycloak2' is displayed with a blue Twitter icon. To the right of the name is a 'Test OAuth' button. A navigation bar below the name includes tabs for 'Details', 'Settings', 'Keys and Access Tokens', and 'Permissions', with 'Details' being the active tab. Under the application name, the details are listed: 'Key cloak test application' and 'http://twitter.com'. Below this, the 'Organization' section is shown with the note: 'Information about the organization or company associated with your application. This information is optional.' It lists 'Organization' as 'None' and 'Organization website' as 'None'. Finally, the 'Application Settings' section is shown with the note: 'Your application's Consumer Key and Secret are used to `authenticate` requests to the Twitter Platform.' It lists 'Access level' as 'Read and write (modify app permissions)', 'Consumer Key (API Key)' as 'Lf42VpLK40KnXp7Vi2FWBNZQe (manage keys and access tokens)', and 'Callback URL' as 'http://127.0.0.1:8080/auth/realm/master/broker/twitter/endpoint'.

Next go to the `Keys and Access Tokens` tab.

## *Keys and Access Tokens*



## keycloak2

Test OAuth

[Details](#) [Settings](#) [Keys and Access Tokens](#) [Permissions](#)

### Application Settings

Keep the "Consumer Secret" a secret. This key should never be human-readable in your application.

Consumer Key (API Key)	Lf42VpLK40KnXp7VI2FWBNZQe
Consumer Secret (API Secret)	SY2tAxCCcCjQcu1LKR8SJY8RpgPNCprty3qNe1SYLu5GwoKF9
Access Level	Read and write ( <a href="#">modify app permissions</a> )
Owner	patriot1burke
Owner ID	26763164

### Application Actions

[Regenerate Consumer Key and Secret](#) [Change App Permissions](#)

### Your Access Token

Finally, you will need to obtain the API Key and secret from this page and copy them back into the `Client ID` and `Client Secret` fields on the KeyCloak `Add identity provider` page.

## 12.5. OpenID Connect v1.0 Identity Providers

KeyCloak can broker identity providers based on the OpenID Connect protocol. These IDPs must support the [Authorization Code Flow](#) as defined by the specification in order to authenticate the user and authorize access.

To begin configuring an OIDC provider, go to the `Identity Providers` left menu item and select `OpenID Connect v1.0` from the `Add provider` drop down list. This will bring you to the `Add identity provider` page.

## Add Identity Provider

The screenshot shows the configuration interface for adding an identity provider. On the left, a sidebar lists 'Groups', 'Users', 'Sessions', 'Events', and 'Import'. The main panel is titled 'Add Identity Provider' and contains several configuration sections:

- General IDP Configuration:** Includes fields for 'Trust Email' (set to OFF), 'GUI order' (empty), 'First Login Flow' (set to 'first broker login'), and 'Post Login Flow' (empty).
- OpenID Connect Config:** This section is expanded and contains the following fields:
  - \* Authorization URL: Empty input field.
  - \* Token URL: Empty input field.
  - Logout URL: Empty input field.
  - Backchannel Logout: A switch set to OFF.
  - Disable User Info: A switch set to OFF.
  - User Info URL: Empty input field.
  - \* Client ID: Empty input field.
  - \* Client Secret: Empty input field.
  - Issuer: Empty input field.
  - Default Scopes: Empty input field.
  - Prompt: A dropdown menu set to 'unspecified'.
  - Validate Signatures: A switch set to OFF.
- Import External IDP Config:** Includes fields for 'Import from URL' (empty input field) and 'Import from file' (a 'Select file' button).

At the bottom are 'Save' and 'Cancel' buttons.

The initial configuration options on this page are described in [General IDP Configuration](#). You must define the OpenID Connect configuration options as well. They basically describe the OIDC IDP you are communicating with.

*Table 2. OpenID Connect Config*

Configuration	Description
Authorization URL	Authorization URL endpoint required by the OIDC protocol.

Token URL	Token URL endpoint required by the OIDC protocol.
Logout URL	Logout URL endpoint defined in the OIDC protocol. This value is optional.
Backchannel Logout	Backchannel logout is a background, out-of-band, REST invocation to the IDP to logout the user. Some IDPs can only perform logout through browser redirects as they may only be able to identify sessions via a browser cookie.
User Info URL	User Info URL endpoint defined by the OIDC protocol. This is an endpoint from which user profile information can be downloaded.
Client ID	This realm will act as an OIDC client to the external IDP. Your realm will need an OIDC client ID when using the Authorization Code Flow to interact with the external IDP.
Client Secret	This realm will need a client se-

	cret to use when using the Authorization Code Flow.
Issuer	Responses from the IDP may contain an issuer claim. This config value is optional. If specified, this claim will be validated against the value you provide.
Default Scopes	Space-separated list of OIDC scopes to send with the authentication request. The default is <code>openid</code> .
Prompt	Another optional switch. This is the prompt parameter defined by the OIDC specification. Through it you can force re-authentication and other options. See the specification for more details.
Accepts prompt=none forward from client	Specifies whether the IDP accepts forwarded authentication requests that contain the <code>prompt=none</code> query parameter or not. When a realm receives an auth request with <code>prompt=none</code> it checks if the user is currently authenticated

and normally returns a `login_required` error if the user is not logged in. However, when a default IDP can be determined for the auth request (either via `kc_idp_hint` query param or by setting up a default IDP for the realm) we should be able to forward the auth request with `prompt=none` to the default IDP so that it checks if the user is currently authenticated there. Because not all IDPs support requests with `prompt=none` this switch is used to indicate if the default IDP supports the param before redirecting the auth request.

It is important to note that if the user is not authenticated in the IDP, the client will still get a `login_required` error. Even if the user is currently authenticated in the IDP, the client might still get an `interaction_required` error if authentication or consent pages requiring user interaction would be otherwise

	<p>displayed. This includes required actions (e.g. change password), consent screens and any screens set to be displayed by the <code>first broker login</code> flow or <code>post broker login</code> flow.</p>
Validate Signatures	<p>Another optional switch. This is to specify if KeyCloak will verify the signatures on the external ID Token signed by this identity provider. If this is on, the KeyCloak will need to know the public key of the external OIDC identity provider. See below for how to set it up. <b>WARNING:</b> For the performance purposes, KeyCloak caches the public key of the external OIDC identity provider. If you think that private key of your identity provider was compromised, it is obviously good to update your keys, but it's also good to clear the keys cache. See <a href="#">Clearing the cache</a> section for more details.</p>
Use JWKS URL	<p>Applicable if <code>Validate Signatures</code> is on. If the switch is</p>

on, then identity provider public keys will be downloaded from given JWKS URL. This allows great flexibility because new keys will be always re-downloaded when the identity provider generates new keypair. If the switch is off, then public key (or certificate) from the Keycloak DB is used, so whenever the identity provider keypair changes, you will always need to import the new key to the Keycloak DB as well.

#### JWKS URL

URL where the identity provider JWK keys are stored. See the [JWK specification](#) for more details. If you use an external Keycloak as an identity provider, then you can use URL like <http://broker-keycloak:8180/auth/realms/test/protocol/openid-connect/certs> assuming your brokered Keycloak is running on <http://broker-keycloak:8180> and its realm is test .

#### Validating Public Key

Applicable if Use JWKS URL is

off. Here is the public key in PEM format that must be used to verify external IDP signatures.

### Validating Public Key Id

Applicable if `Use JWKS URL` is off. This field specifies ID of the public key in PEM format. This config value is optional. As there is no standard way for computing key ID from key, various external identity providers might use different algorithm from KeyCloak. If the value of this field is not specified, the validating public key specified above is used for all requests regardless of key ID sent by external IDP. When set, value of this field serves as key ID used by KeyCloak for validating signatures from such providers and must match the key ID specified by the IDP.

You can also import all this configuration data by providing a URL or file that points to OpenID Provider Metadata (see OIDC Discovery specification). If you are connecting to a KeyCloak external IDP, you can import the IDP settings from the url `<root>/auth/realms/{realm-`

`name}/.well-known/openid-configuration`. This link is a JSON document describing metadata about the IDP.

## 12.6. SAML v2.0 Identity Providers

Keycloak can broker identity providers based on the SAML v2.0 protocol.

To begin configuring an SAML v2.0 provider, go to the `Identity Providers` left menu item and select `SAML v2.0` from the `Add provider` drop down list. This will bring you to the `Add identity provider` page.

*Add Identity Provider*

The initial configuration options on this page are described in [General IDP Configuration](#). You must define the SAML configuration options as well. They basically describe the SAML IDP you are communicating with.

*Table 3. SAML Config*

Configuration	Description
Single Sign-On Service URL	This is a required field and specifies the SAML endpoint to start the authentication process.

	on process. If your SAML IDP publishes an IDP entity descriptor, the value of this field will be specified there.
Single Logout Service URL	This is an optional field that specifies the SAML logout endpoint. If your SAML IDP publishes an IDP entity descriptor, the value of this field will be specified there.
Backchannel Logout	Enable if your SAML IDP supports backchannel logout.
NameID Policy Format	Specifies the URI reference corresponding to a name identifier format. Defaults to <code>urn:oasis:names:tc:SAML:2.0:nameid-format:persistent</code> .
HTTP-POST Binding Response	When this realm responds to any SAML requests sent by the external IDP, which SAML binding should be used? If set to <code>off</code> , then the Redirect Binding will be used.
HTTP-POST Binding for AuthnRequest	When this realm requests authentication from the external SAML IDP, which SAML binding should be used? If set to <code>off</code> , then the Redirect Binding will be used.

Want AuthnRequests Signed	If true, it will use the realm's keypair to sign requests sent to the external SAML IDP.
Signature Algorithm	If <code>Want AuthnRequests Signed</code> is on, then you can also pick the signature algorithm to use.
SAML Signature Key Name	Signed SAML documents sent via POST binding contain identification of signing key in <code>KeyName</code> element. This by default contains KeyCloak key ID. However various external SAML IDPs might expect a different key name or no key name at all. This switch controls whether <code>KeyName</code> contains key ID (option <code>KEY_ID</code> ), subject from certificate corresponding to the realm key (option <code>CERT_SUBJECT</code> - expected for instance by Microsoft Active Directory Federation Services), or that the key name hint is completely omitted from the SAML message (option <code>NONE</code> ).
Force Authentication	Indicates that the user will be forced to enter their credentials at the external IDP even if they are already logged in.
Validate Signature	Whether or not the realm should expect that SAML requests and responses from

	the external IDP to be digitally signed. It is highly recommended you turn this on!
Validating X509 Certificate	The public certificate that will be used to validate the signatures of SAML requests and responses from the external IDP.

You can also import all this configuration data by providing a URL or file that points to the SAML IDP entity descriptor of the external IDP. If you are connecting to a Keycloak external IDP, you can import the IDP settings from the URL `<root>/auth/realms/{realm-name}/protocol/saml/descriptor`. This link is an XML document describing metadata about the IDP.

You can also import all this configuration data by providing a URL or XML file that points to the entity descriptor of the external SAML IDP you want to connect to.

### 12.6.1. SP Descriptor

Once you create a SAML provider, there is an `EXPORT` button that appears when viewing that provider. Clicking this button will export a SAML SP entity descriptor which you can use to import into the external SP.

This metadata is also available publicly by going to the URL.

```
http[s]://{host:port}/auth/realms/{realm-name}/broker/{broker-alias}/endpoint/descriptor
```

## 12.7. Client-suggested Identity Provider

OIDC applications can bypass the Keycloak login page by specifying a hint on which identity provider they want to use.

This is done by setting the `kc_idp_hint` query parameter in the Authorization Code Flow authorization endpoint.

Keycloak OIDC client adapters also allow you to specify this query parameter when you access a secured resource at the application.

For example:

```
GET /myapplication.com?kc_idp_hint=facebook HTTP/1.1
Host: localhost:8080
```

In this case, it is expected that your realm has an identity provider with an alias `facebook`. If this provider doesn't exist the login form will be displayed.

If you are using `keycloak.js` adapter, you can also achieve the same behavior:

```
var keycloak = new Keycloak('keycloak.json');

keycloak.createLoginUrl({
    idpHint: 'facebook'
});
```

The `kc_idp_hint` query parameter also allows the client to override

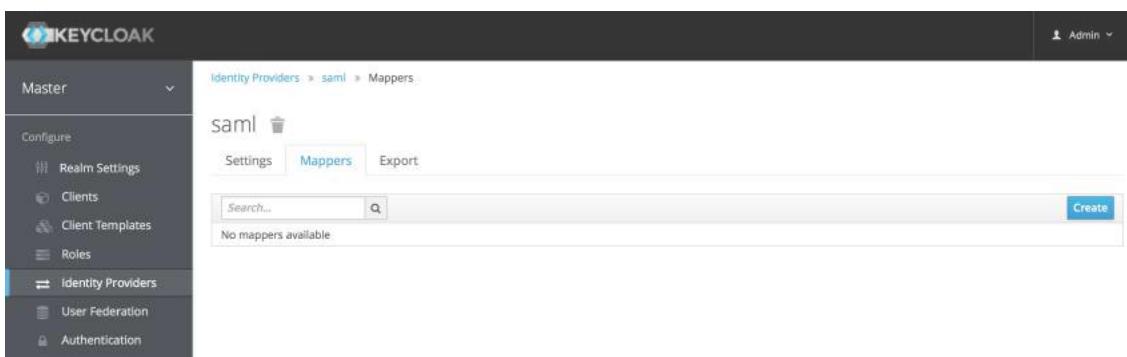
the default identity provider if one is configured for the `Identity Provider Redirector` authenticator. The client can also disable the automatic redirecting by setting the `kc_idp_hint` query parameter to an empty value.

## 12.8. Mapping Claims and Assertions

You can import the SAML and OpenID Connect metadata provided by the external IDP you are authenticating with into the environment of the realm. This allows you to extract user profile metadata and other information so that you can make it available to your applications.

Each new user that logs into your realm via an external identity provider will have an entry for them created in the local Keycloak database, based on the metadata from the SAML or OIDC assertions and claims.

If you click on an identity provider listed in the `Identity Providers` page for your realm, you will be brought to the IDPs `Settings` tab. On this page there is also a `Mappers` tab. Click on that tab to start mapping your incoming IDP metadata.

A screenshot of the Keycloak administration interface. The top navigation bar shows 'KEYCLOAK' and 'Admin'. The left sidebar is titled 'Master' and contains 'Configure' (with 'Realm Settings', 'Clients', 'Client Templates', 'Roles'), 'Identity Providers' (which is expanded), 'User Federation', and 'Authentication'. The main content area shows the path 'Identity Providers > saml > Mappers'. The 'Mappers' tab is active, indicated by a blue border. Below the tabs is a search bar with placeholder text 'Search...'. At the bottom right of the main area is a blue 'Create' button.

There is a `Create` button on this page. Clicking on this create button allows you to create a broker mapper. Broker mappers can import SAML attributes or OIDC ID/Access token claims into user attributes

and user role mappings.

The screenshot shows the Keycloak administrative interface. The left sidebar is titled 'Master' and contains the following navigation items under 'Configure': 'Realm Settings', 'Clients', 'Client Templates', 'Roles', 'Identity Providers' (which is highlighted in blue), 'User Federation', and 'Authentication'. Under 'Manage', there are 'Groups' and other options. The main content area has a title 'Add Identity Provider Mapper'. It includes fields for 'Name' (with a red asterisk indicating it's required), 'Mapper Type' (set to 'Attribute Importer'), 'Attribute Name', 'Friendly Name', and 'User Attribute Name'. At the bottom right are 'Save' and 'Cancel' buttons.

Select a mapper from the `Mapper Type` list. Hover over the tooltip to see a description of what the mapper does. The tooltips also describe what configuration information you need to enter. Click `Save` and your new mapper will be added.

For JSON based claims, you can use dot notation for nesting and square brackets to access array fields by index. For example 'contact.address[0].country'.

To investigate the structure of user profile JSON data provided by social providers you can enable the `DEBUG` level logger `org.keycloak.social.user_profile_dump`. This is done in the server's app-server configuration file (`domain.xml` or `standalone.xml`).

## 12.9. Available User Session Data

After a user logs in from the external IDP, there is some additional user session note data that KeyCloak stores that you can access. This data can be propagated to the client requesting a login via the token or

SAML assertion being passed back to it by using an appropriate client mapper.

### **identity\_provider**

This is the IDP alias of the broker used to perform the login.

### **identity\_provider\_identity**

This is the IDP username of the currently authenticated user. This is often the same as the KeyCloak username, but doesn't necessarily needs to be. For example KeyCloak user `john` can be linked to the Facebook user `john123@gmail.com`, so in that case value of user session note will be `john123@gmail.com`.

You can use a [Protocol Mapper](#) of type `User Session Note` to propagate this information to your clients.

## **12.10. First Login Flow**

When a user logs in through identity brokering some aspects of the user are imported and linked within the realm's local database. When KeyCloak successfully authenticates users through an external identity provider there can be two situations:

- There is already a KeyCloak user account imported and linked with the authenticated identity provider account. In this case, KeyCloak will just authenticate as the existing user and redirect back to application.
- There is not yet an existing KeyCloak user account imported and linked for this external user. Usually you just want to register and import the new account into KeyCloak database, but what if there is

an existing Keycloak account with the same email? Automatically linking the existing local account to the external identity provider is a potential security hole as you can't always trust the information you get from the external identity provider.

Different organizations have different requirements when dealing with some of the conflicts and situations listed above. For this, there is a `First Login Flow` option in the IDP settings which allows you to choose a [workflow](#) that will be used after a user logs in from an external IDP the first time. By default it points to `first broker login` flow, but you can configure and use your own flow and use different flows for different identity providers.

The flow itself is configured in admin console under `Authentication` tab. When you choose `First Broker Login` flow, you will see what authenticators are used by default. You can re-configure the existing flow. (For example you can disable some authenticators, mark some of them as `required`, configure some authenticators, etc).

### 12.10.1. Default First Login Flow

Let's describe the default behavior provided by `First Broker Login` flow.

#### Review Profile

This authenticator might display the profile info page, where the user can review their profile retrieved from an identity provider. The authenticator is configurable. You can set the `Update Profile On First Login` option. When `On`, users will be always presented with the profile page asking for additional information in order to fe-

derate their identities. When `missing`, users will be presented with the profile page only if some mandatory information (email, first name, last name) is not provided by the identity provider. If `Off`, the profile page won't be displayed, unless user clicks in later phase on `Review profile info` link (page displayed in later phase by `Confirm Link Existing Account` authenticator).

## Create User If Unique

This authenticator checks if there is already an existing Keycloak account with the same email or username like the account from the identity provider. If it's not, then the authenticator just creates a new local Keycloak account and links it with the identity provider and the whole flow is finished. Otherwise it goes to the next `Handle Existing Account` subflow. If you always want to ensure that there is no duplicated account, you can mark this authenticator as `REQUIRED`. In this case, the user will see the error page if there is an existing Keycloak account and the user will need to link his identity provider account through Account management.

## Confirm Link Existing Account

On the info page, the user will see that there is an existing Keycloak account with the same email. They can review their profile again and use different email or username (flow is restarted and goes back to `Review Profile` authenticator). Or they can confirm that they want to link their identity provider account with their existing Keycloak account. Disable this authenticator if you don't want users to see this confirmation page, but go straight to linking identity provider account by email verification or re-authentication.

## Verify Existing Account By Email

This authenticator is `ALTERNATIVE` by default, so it's used only if the realm has SMTP setup configured. It will send email to the user, where they can confirm that they want to link the identity provider with their Keycloak account. Disable this if you don't want to confirm linking by email, but instead you always want users to reauthenticate with their password (and alternatively OTP).

## Verify Existing Account By Re-authentication

This authenticator is used if email authenticator is disabled or not available (SMTP not configured for realm). It will display a login screen where the user needs to authenticate with his password to link their Keycloak account with the Identity provider. User can also re-authenticate with some different identity provider, which is already linked to their Keycloak account. You can also force users to use OTP. Otherwise it's optional and used only if OTP is already set for the user account.

### 12.10.2. Automatically Link Existing First Login Flow

The AutoLink authenticator would be dangerous in a generic environment where users can register themselves using arbitrary usernames/email addresses. Do not use this authenticator unless registration of users is carefully curated and usernames/email addresses are assigned, not requested.

In order to configure a first login flow in which users are automatically linked without being prompted, create a new flow with the following

two authenticators:

### Create User If Unique

This authenticator ensures that unique users are handled. Set the authenticator requirement to "Alternative".

### Automatically Link Brokered Account

Automatically link brokered identities without any validation with this authenticator. This is useful in an intranet environment of multiple user databases each with overlapping usernames/email addresses, but different passwords, and you want to allow users to use any password without having to validate. This is only reasonable if you manage all internal databases, and usernames/email addresses from one database matching those in another database belong to the same person. Set the authenticator requirement to "Alternative".

The described setup uses two authenticators, and is the simplest one, but it is possible to use other authenticators according to your needs. For example, you can add the Review Profile authenticator to the beginning of the flow if you still want end users to confirm their profile information.

## 12.11. Retrieving External IDP Tokens

Keycloak allows you to store tokens and responses from the authentication process with the external IDP. For that, you can use the `Store Token` configuration option on the IDP's settings page.

Application code can retrieve these tokens and responses to pull in extra user information, or to securely invoke requests on the external IDP. For example, an application might want to use the Google token to invoke on other Google services and REST APIs. To retrieve a token for a particular identity provider you need to send a request as follows:

```
GET /auth/realms/{realm}/broker/{provider_alias}/token
HTTP/1.1
Host: localhost:8080
Authorization: Bearer <KEYCLOAK ACCESS TOKEN>
```

An application must have authenticated with Keycloak and have received an access token. This access token will need to have the `broker` client-level role `read-token` set. This means that the user must have a role mapping for this role and the client application must have that role within its scope. In this case, given that you are accessing a protected service in Keycloak, you need to send the access token issued by Keycloak during the user authentication. In the broker configuration page you can automatically assign this role to newly imported users by turning on the `Stored Tokens Readable` switch.

These external tokens can be re-established by either logging in again through the provider, or using the client-initiated account linking API.

## 12.12. Identity broker logout

When logout from Keycloak is triggered, Keycloak will send a request to the external identity provider that was used to login to Keycloak, and the user will be logged out from this identity provider as well. It is possible to skip this behavior and avoid logout at the external identity provider. See [adapter logout documentation](#) for more details.



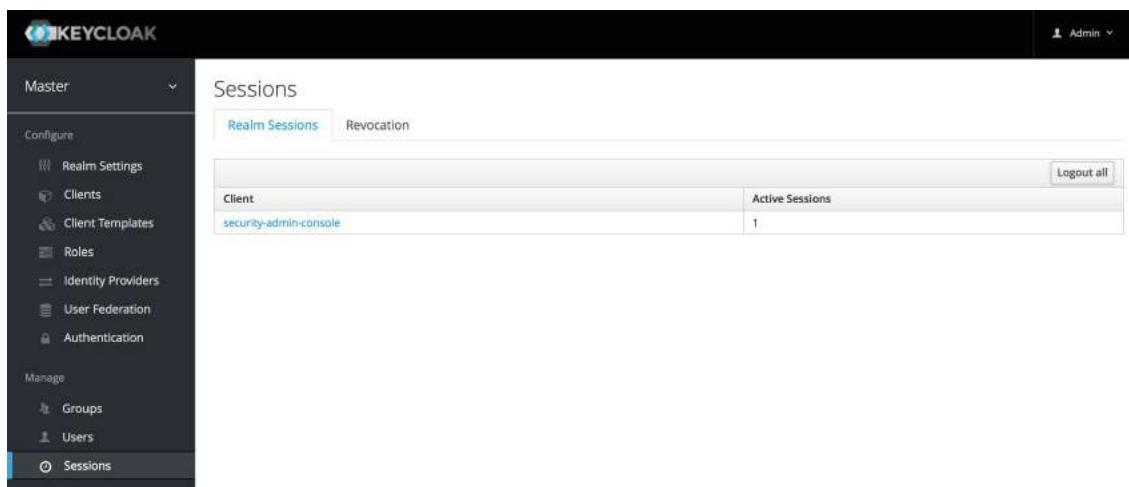
# 13. User Session Management

When a user logs into a realm, Keycloak maintains a user session for them and remembers each and every client they have visited within the session. There are a lot of administrative functions that realm admins can perform on these user sessions. They can view login stats for the entire realm and dive down into each client to see who is logged in and where. Admins can logout a user or set of users from the Admin Console. They can revoke tokens and set up all the token and session timeouts there too.

## 13.1. Administering Sessions

If you go to the `Sessions` left menu item you can see a top level view of the number of sessions that are currently active in the realm.

### *Sessions*



The screenshot shows the Keycloak Admin Console interface. The left sidebar has a dark theme with white text. It includes sections for `Configure` (with `Realm Settings`, `Clients`, `Client Templates`, `Roles`, `Identity Providers`, `User Federation`, and `Authentication`) and `Manage` (with `Groups` and `Users`). The `Sessions` item is highlighted with a blue border. The main content area has a header "Sessions" with tabs for `Realm Sessions` and `Revocation`. Below this is a table titled "Clients" with one row: "Client" (link to "security-admin-console") and "Active Sessions" (value 1). On the far right of the table is a button labeled "Logout all". The top right corner of the interface shows the user is "Admin" and has a dropdown menu.

A list of clients is given and how many active sessions there currently are for that client. You can also logout all users in the realm by clicking the `Logout all` button on the right side of this list.

### 13.1.1. Logout All Limitations

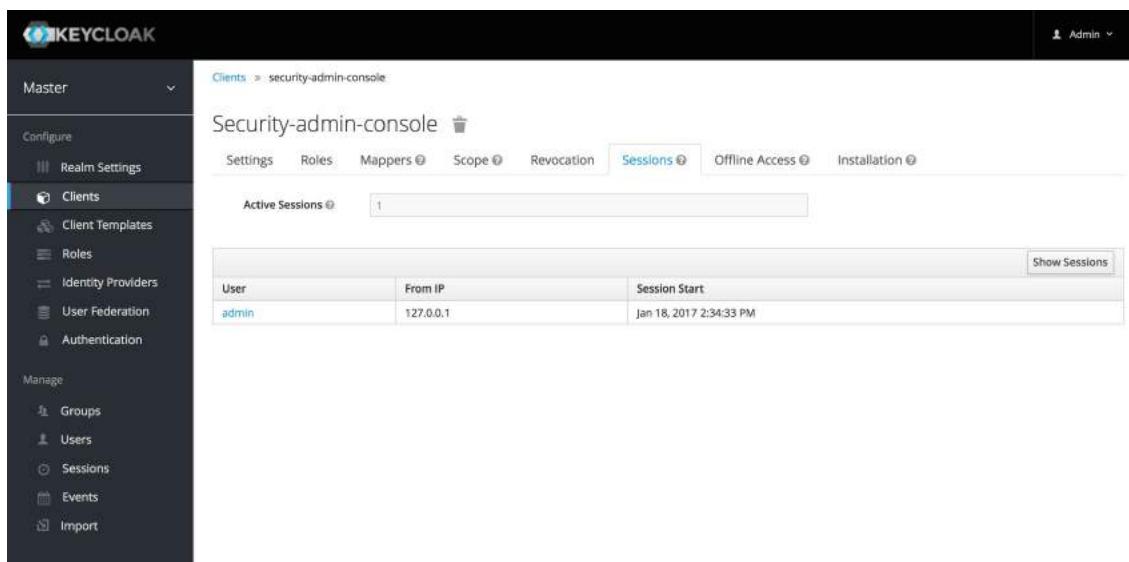
Any SSO cookies set will now be invalid and clients that request authentication in active browser sessions will now have to re-login. Only certain clients are notified of this logout event, specifically clients that are using the Keycloak OIDC client adapter. Other client types (i.e. SAML) will not receive a backchannel logout request.

It is important to note that any outstanding access tokens are not revoked by clicking `Logout all`. They have to expire naturally. You have to push a [revocation policy](#) out to clients, but that also only works with clients using the Keycloak OIDC client adapter.

### 13.1.2. Application Drilldown

On the `Sessions` page, you can also drill down to each client. This will bring you to the `Sessions` tab of that client. Clicking on the `Show Sessions` button there allows you to see which users are logged into that application.

#### *Application Sessions*

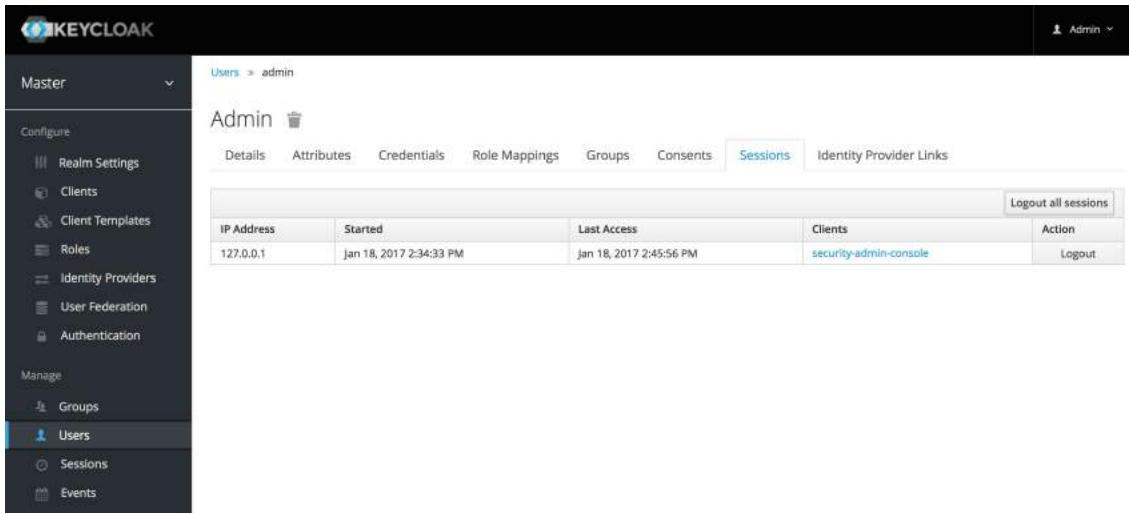


The screenshot shows the Keycloak security-admin-console interface. The left sidebar is dark-themed with white text and icons. It includes sections for **Configure** (Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication), **Manage** (Groups, Users, Sessions, Events, Import), and a search bar. The main content area has a light background. At the top, it says "Clients > security-admin-console". Below that is the client name "Security-admin-console" with a trash icon. A navigation bar with tabs: Settings, Roles, Mappers, Scope, Revocation, **Sessions**, Offline Access, Installation. Under "Sessions", it says "Active Sessions: 1". A table shows one session: "User: admin, From IP: 127.0.0.1, Session Start: Jan 18, 2017 2:34:33 PM". To the right of the table is a "Show Sessions" button. The top right corner shows the user "Admin" with a dropdown arrow.

### 13.1.3. User Drilldown

If you go to the `Sessions` tab of an individual user, you can also view the session information.

### *User Sessions*



The screenshot shows the Keycloak Admin UI. The left sidebar has a 'Master' dropdown and sections for 'Configure' (Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication) and 'Manage' (Groups, Users, Sessions, Events). The 'Users' section is currently selected. The main area shows the 'Admin' user details with tabs for Details, Attributes, Credentials, Role Mappings, Groups, Consents, Sessions (which is selected and highlighted in blue), and Identity Provider Links. Below the tabs is a table with one row of session information:

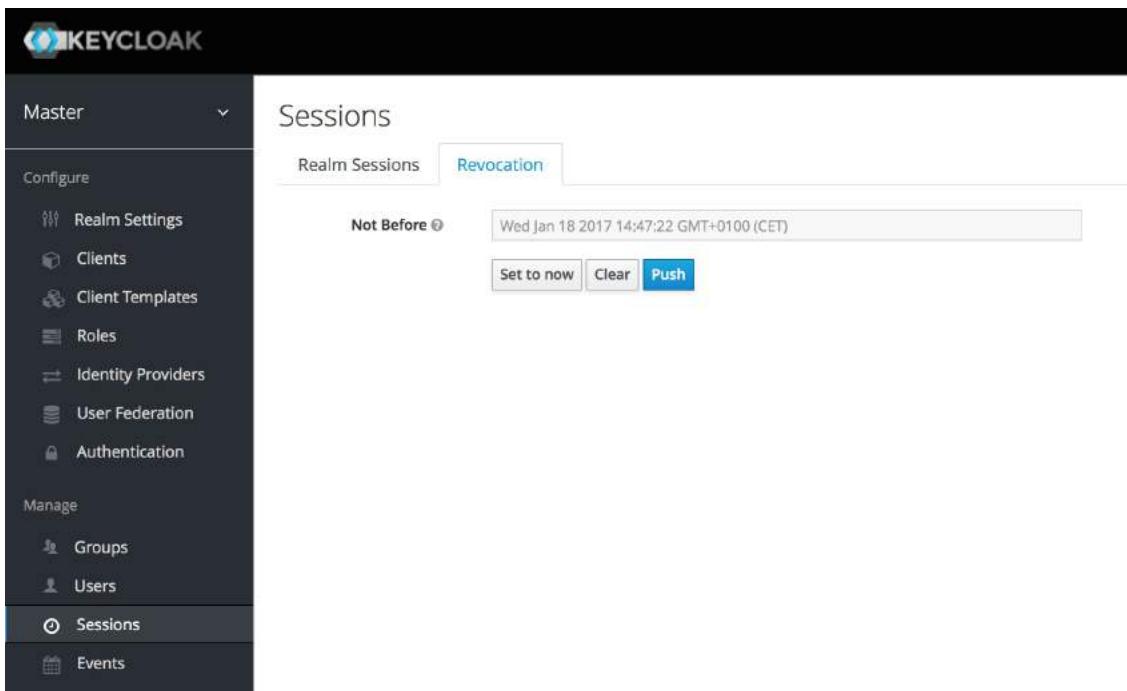
IP Address	Started	Last Access	Clients	Action
127.0.0.1	Jan 18, 2017 2:34:33 PM	Jan 18, 2017 2:45:56 PM	security-admin-console	Logout

A 'Logout all sessions' button is located at the top right of the table.

## 13.2. Revocation Policies

If your system is compromised you will want a way to revoke all sessions and access tokens that have been handed out. You can do this by going to the `Revocation` tab of the `Sessions` screen.

### *Revocation*



You can only set a time-based revocation policy. The console allows you to specify a time and date where any session or token issued before that time and date is invalid. The `Set to now` will set the policy to the current time and date. The `Push` button will push this revocation policy to any registered OIDC client that has the KeyCloak OIDC client adapter installed.

### 13.3. Session and Token Timeouts

KeyCloak gives you fine grain control of session, cookie, and token timeouts. This is all done on the `Tokens` tab in the `Realm Settings` left menu item.

#### *Tokens Tab*

The screenshot shows the Keycloak administration interface for the 'Master' realm. The left sidebar contains navigation links for 'Configure' (Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication), 'Manage' (Groups, Users, Sessions, Events, Import), and other realm settings. The main content area is titled 'Master' and has a 'Tokens' tab selected. The configuration includes:

- Revoke Refresh Token:** A switch set to 'OFF'.
- SSO Session Idle:** Set to 30 Minutes.
- SSO Session Max:** Set to 10 Hours.
- Offline Session Idle:** Set to 30 Days.
- Access Token Lifespan:** Set to 1 Minutes.
- Access Token Lifespan For Implicit Flow:** Set to 15 Minutes.
- Client login timeout:** Set to 1 Minutes.
- Login timeout:** Set to 30 Minutes.
- Login action timeout:** Set to 5 Minutes.

At the bottom are 'Save' and 'Cancel' buttons.

Let's walk through each of the items on this page.

Configuration	Description
Revoke Refresh Token	For OIDC clients that are doing the refresh token flow, this flag, if on, will revoke that refresh token and issue another with the request that the client has to use. This basically means that refresh tokens have a one time use.
SSO Session Idle	Also pertains to OIDC clients. If

	<p>the user is not active for longer than this timeout, the user session will be invalidated. How is idle time checked? A client requesting authentication will bump the idle timeout. Refresh token requests will also bump the idle timeout. There is a small window of time that is always added to the idle timeout before the session is actually invalidated (See note below).</p>
SSO Session Max	Maximum time before a user session is expired and invalidated. This is a hard number and time. It controls the maximum time a user session can remain active, regardless of activity.
SSO Session Idle Remember Me	Same as the standard SSO Session Idle configuration but specific to logins with remember me enabled. It allows for the specification of longer session idle timeouts when remember me is selected during the login process. It is an optional configuration and if not set to a value big-

	<p>ger than 0 it uses the same idle timeout set in the SSO Session Idle configuration.</p>
SSO Session Max Remember Me	<p>Same as the standard SSO Session Max but specific to logins with remember me enabled. It allows for the specification of longer lived sessions when remember me is selected during the login process. It is an optional configuration and if not set to a value bigger than 0 it uses the same session lifespan set in the SSO Session Max configuration.</p>
Offline Session Idle	<p>For <a href="#">offline access</a>, this is the time the session is allowed to remain idle before the offline token is revoked. There is a small window of time that is always added to the idle timeout before the session is actually invalidated (See note below).</p>
Offline Session Max Limited	<p>For <a href="#">offline access</a>, if this flag is on, Offline Session Max is enabled to control the maximum time</p>

	<p>the offline token can remain active, regardless of activity.</p>
Offline Session Max	<p>For <a href="#">offline access</a>, this is the maximum time before the corresponding offline token is revoked. This is a hard number and time. It controls the maximum time the offline token can remain active, regardless of activity.</p>
Access Token Lifespan	<p>When an OIDC access token is created, this value affects the expiration.</p>
Access Token Lifespan For Implicit Flow	<p>With the Implicit Flow no refresh token is provided. For this reason there's a separate timeout for access tokens created with the Implicit Flow.</p>
Client login timeout	<p>This is the maximum time that a client has to finish the Authorization Code Flow in OIDC.</p>
Login timeout	<p>Total time a login must take. If authentication takes longer than this time then the user will have to start the authentication pro-</p>

	cess over.
Login action timeout	Maximum time a user can spend on any one page in the authentication process.
User-Initiated Action Lifespan	Maximum time before an action permit sent by a user (e.g. forgot password e-mail) is expired. This value is recommended to be short because it is expected that the user would react to self-created action quickly.
Default Admin-Initiated Action Lifespan	Maximum time before an action permit sent to a user by an admin is expired. This value is recommended to be long to allow admins send e-mails for users that are currently offline. The default timeout can be overridden right before issuing the token.
Override User-Initiated Action Lifespan	Permits the possibility of having independent timeouts per operation (e.g. e-mail verification, forgot password, user actions and Identity Provider E-mail Verification). This field is non manda-

tory and if nothing is specified it defaults to the value configured at *User-Initiated Action Lifespan*.

For idle timeouts, there is a small window of time (2 minutes) during which the session is kept unexpired. For example, when you have timeout set to 30 minutes, it will be actually 32 minutes before the session is expired. This is needed for some corner-case scenarios in cluster and cross-datacenter environments, in cases where the token was refreshed on one cluster node for a very short time before the expiration and the other cluster nodes would in the meantime incorrectly consider the session as expired, because they had not yet received the message about successful refresh from the node which did the refresh.

## 13.4. Offline Access

Offline access is a feature described in [OpenID Connect specification](#). The idea is that during login, your client application will request an Offline token instead of a classic Refresh token. The application can save this offline token in a database or on disk and can use it later even if user is logged out. This is useful if your application needs to do some "offline" actions on behalf of user even when the user is not online. An example is a periodic backup of some data every night.

Your application is responsible for persisting the offline token in some

storage (usually a database) and then using it to manually retrieve new access token from Keycloak server.

The difference between a classic Refresh token and an Offline token is, that an offline token will never expire by default and is not subject of `SSO Session Idle timeout` and `SSO Session Max lifespan`. The offline token is valid even after a user logout or server restart. However by default you do need to use the offline token for a refresh token action at least once per 30 days (this value, `Offline Session Idle timeout`, can be changed in the administration console in the `Tokens` tab under `Realm Settings`). Moreover, if you enable the option `Offline Session Max Limited`, then the offline token expires after 60 days regardless of using the offline token for a refresh token action (this value, `Offline Session Max lifespan`, can also be changed in the administration console in the `Tokens` tab under `Realm Settings`). Also if you enable the option `Revoke refresh tokens`, then each offline token can be used just once. So after refresh, you always need to store the new offline token from refresh response into your DB instead of the previous one.

Users can view and revoke offline tokens that have been granted by them in the [User Account Service](#). The admin user can revoke offline tokens for individual users in admin console in the `Consents` tab of a particular user. The admin can also view all the offline tokens issued in the `Offline Access` tab of each client. Offline tokens can also be revoked by setting a [revocation policy](#).

To be able to issue an offline token, users need to have the role mapping for the realm-level role `offline_access`. Clients also need to have

that role in their scope. Finally, the client needs to have an `offline_access` client scope added as an `Optional client scope` to it, which is done by default.

The client can request an offline token by adding the parameter `scope=offline_access` when sending authorization request to KeyCloak. The KeyCloak OIDC client adapter automatically adds this parameter when you use it to access secured URL of your application (i.e. `http://localhost:8080/customer-portal/secured?scope=offline_access`). The Direct Access Grant and Service Accounts also support offline tokens if you include `scope=offline_access` in the body of the authentication request.

---

## 14. User Storage Federation

Many companies have existing user databases that hold information about users and their passwords or other credentials. In many cases, it is just not possible to migrate off of those existing stores to a pure KeyCloak deployment. KeyCloak can federate existing external user databases. Out of the box we have support for LDAP and Active Directory. You can also code your own extension for any custom user databases you might have using our User Storage SPI.

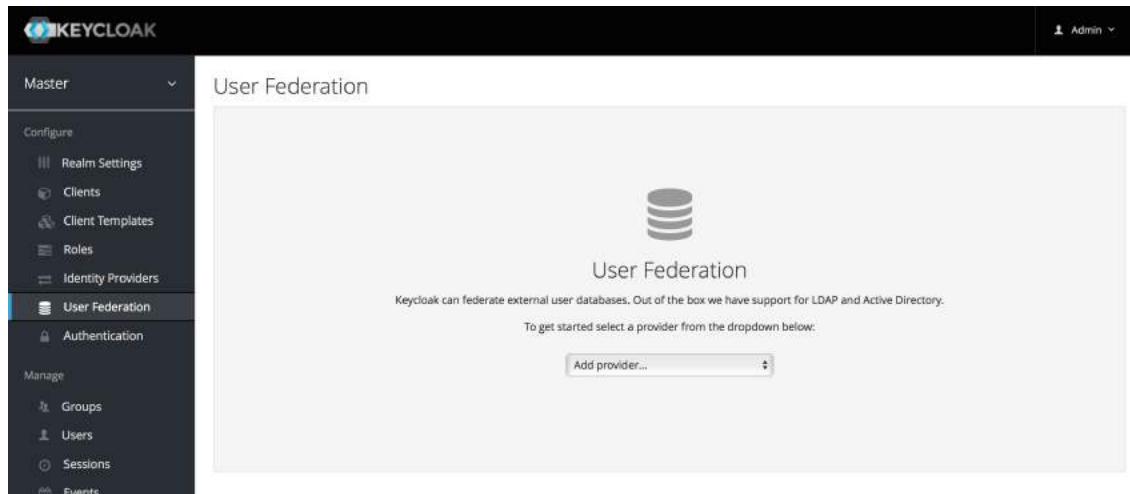
The way it works is that when a user logs in, KeyCloak will look into its own internal user store to find the user. If it can't find it there it will iterate over every User Storage provider you have configured for the realm until it finds a match. Data from the external store is mapped into a common user model that is consumed by the KeyCloak runtime. This common user model can then be mapped to OIDC token claims and SAML assertion attributes.

External user databases rarely have every piece of data needed to support all the features that KeyCloak has. In this case, the User Storage Provider can opt to store some things locally in the KeyCloak user store. Some providers even import the user locally and sync periodically with the external store. All this depends on the capabilities of the provider and how its configured. For example, your external user store may not support OTP. Depending on the provider, this OTP can be handled and stored by KeyCloak.

### 14.1. Adding a Provider

To add a storage provider go to the `User Federation` left menu item in the Admin Console.

## *User Federation*



On the center, there is an `Add Provider` list box. Choose the provider type you want to add and you will be brought to the configuration page of that provider.

## 14.2. Dealing with Provider Failures

If a User Storage Provider fails, that is, if your LDAP server is down, you may have trouble logging in and may not be able to view users in the admin console. KeyCloak does not catch failures when using a Storage Provider to lookup a user. It will abort the invocation. So, if you have a Storage Provider with a higher priority that fails during user lookup, the login or user query will fail entirely with an exception and abort. It will not fail over to the next configured provider.

The local KeyCloak user database is always searched first to resolve users before any LDAP or custom User Storage Provider. You may want to consider creating an admin account that is stored in the local KeyClo-

ak user database just in case any problems come up in connecting to your LDAP and custom back ends.

Each LDAP and custom User Storage Provider has an `enable` switch on its admin console page. Disabling the User Storage Provider will skip the provider when doing user queries so that you can view and login with users that might be stored in a different provider with lower priority. If your provider is using an `import` strategy and you disable it, imported users are still available for lookup, but only in read only mode. You will not be able to modify these users until you re-enable the provider.

The reason why KeyCloak does not fail over if a Storage Provider look-up fails is that user databases often have duplicate usernames or duplicate emails between them. This can cause security issues and unforeseen problems as the user may be loaded from one external store when the admin is expecting the user to be loaded from another.

### 14.3. LDAP and Active Directory

KeyCloak comes with a built-in LDAP/AD provider. It is possible to federate multiple different LDAP servers in the same KeyCloak realm. You can map LDAP user attributes into the KeyCloak common user model. By default, it maps username, email, first name, and last name, but you are free to configure additional [mappings](#). The LDAP provider also supports password validation via LDAP/AD protocols and different storage, edit, and synchronization modes.

To configure a federated LDAP store go to the Admin Console. Click on the `User Federation` left menu option. When you get to this page

there is an `Add Provider` select box. You should see `ldap` within this list. Selecting `ldap` will bring you to the LDAP configuration page.

### 14.3.1. Storage Mode

By default, KeyCloak will import users from LDAP into the local KeyCloak user database. This copy of the user is either synchronized on demand, or through a periodic background task. The one exception to this is passwords. Passwords are not imported and password validation is delegated to the LDAP server. The benefits to this approach is that all KeyCloak features will work as any extra per-user data that is needed can be stored locally. This approach also reduces load on the LDAP server as uncached users are loaded from the KeyCloak database the 2nd time they are accessed. The only load your LDAP server will have is password validation. The downside to this approach is that when a user is first queried, this will require a KeyCloak database insert. The import will also have to be synchronized with your LDAP server as needed.

Alternatively, you can choose not to import users into the KeyCloak user database. In this case, the common user model that the KeyCloak runtime uses is backed only by the LDAP server. This means that if LDAP doesn't support a piece of data that a KeyCloak feature needs that feature will not work. The benefit to this approach is that you do not have the overhead of importing and synchronizing a copy of the LDAP user into the KeyCloak user database.

This storage mode is controled by the `Import Users` switch. Set to `On` to import users.

### 14.3.2. Edit Mode

Users, through the [User Account Service](#), and admins through the Admin Console have the ability to modify user metadata. Depending on your setup you may or may not have LDAP update privileges. The `Edit Mode` configuration option defines the edit policy you have with your LDAP store.

## **READONLY**

Username, email, first name, last name, and other mapped attributes will be unchangeable. KeyCloak will show an error anytime anybody tries to update these fields. Also, password updates will not be supported.

## **WRITABLE**

Username, email, first name, last name, and other mapped attributes and passwords can all be updated and will be synchronized automatically with your LDAP store.

## **UNSYNCED**

Any changes to username, email, first name, last name, and passwords will be stored in KeyCloak local storage. It is up to you to figure out how to synchronize back to LDAP. This allows KeyCloak deployments to support updates of user metadata on a read-only LDAP server. This option only applies when you are importing users from LDAP into the local KeyCloak user database.

### **14.3.3. Other config options**

#### **Console Display Name**

Name used when this provider is referenced in the admin console

## Priority

The priority of this provider when looking up users or adding a user.

## Sync Registrations

Does your LDAP support adding new users? Click this switch if you want new users created by KeyCloak in the admin console or the registration page to be added to LDAP.

## Allow Kerberos authentication

Enable Kerberos/SPNEGO authentication in realm with users data provisioned from LDAP. More info in [Kerberos section](#).

## Other options

The rest of the configuration options should be self explanatory. You can mouseover the tooltips in Admin Console to see some more details about them.

### 14.3.4. Connect to LDAP over SSL

When you configure a secured connection URL to your LDAP store(for example `ldaps://myhost.com:636` ), KeyCloak will use SSL for the communication with LDAP server. The important thing is to properly configure a truststore on the KeyCloak server side, otherwise KeyCloak can't trust the SSL connection to LDAP.

The global truststore for the KeyCloak can be configured with the Truststore SPI. Please check out the [{installguide!}](#) for more detail. If you don't configure the truststore SPI, the truststore will fallback to the default mechanism provided by Java (either the file provided by system property `javax.net.ssl.trustStore` or the cacerts file from the

JDK if the system property is not set).

There is a configuration property `Use Truststore SPI` in the LDAP federation provider configuration, where you can choose whether the Truststore SPI is used. By default, the value is `Only for ldaps`, which is fine for most deployments. The Truststore SPI will only be used if the connection to LDAP starts with `ldaps`.

#### 14.3.5. Sync of LDAP users to KeyCloak

If you have import enabled, the LDAP Provider will automatically take care of synchronization (import) of needed LDAP users into the KeyCloak local database. As users log in, the LDAP provider will import the LDAP user into the KeyCloak database and then authenticate against the LDAP password. This is the only time users will be imported. If you go to the `Users` left menu item in the Admin Console and click the `View all users` button, you will only see those LDAP users that have been authenticated at least once by KeyCloak. It is implemented this way so that admins don't accidentally try to import a huge LDAP DB of users.

If you want to sync all LDAP users into the KeyCloak database, you may configure and enable the `Sync Settings` of the LDAP provider you configured. There are 2 types of synchronization:

##### Periodic Full sync

This will synchronize all LDAP users into KeyCloak DB. Those LDAP users, which already exist in KeyCloak and were changed in LDAP directly will be updated in KeyCloak DB (For example if user `Mary Kelly` was changed in LDAP to `Mary Smith` ).

## **Periodic Changed users sync**

When syncing occurs, only those users that were created or updated after the last sync will be updated and/or imported.

The best way to handle syncing is to click the `Synchronize all users` button when you first create the LDAP provider, then set up a periodic sync of changed users. The configuration page for your LDAP Provider has several options to support you.

### **14.3.6. LDAP Mappers**

LDAP mappers are `listeners`, which are triggered by the LDAP Provider at various points, provide another extension point to LDAP integration. They are triggered when a user logs in via LDAP and needs to be imported, during Keycloak initiated registration, or when a user is queried from the Admin Console. When you create an LDAP Federation provider, Keycloak will automatically provide set of built-in `mappers` for this provider. You are free to change this set and create a new mapper or update/delete existing ones.

#### **User Attribute Mapper**

This allows you to specify which LDAP attribute is mapped to which attribute of Keycloak user. So, for example, you can configure that LDAP attribute `mail` to the attribute `email` in the Keycloak database. For this mapper implementation, there is always a one-to-one mapping (one LDAP attribute is mapped to one Keycloak attribute)

#### **FullName Mapper**

This allows you to specify that the full name of the user, which is saved in some LDAP attribute (usually `cn`) will be mapped to

`firstName` and `lastname` attributes in the KeyCloak database. Having `cn` to contain full name of user is a common case for some LDAP deployments.

## Role Mapper

This allows you to configure role mappings from LDAP into KeyCloak role mappings. One Role mapper can be used to map LDAP roles (usually groups from a particular branch of LDAP tree) into roles corresponding to either realm roles or client roles of a specified client. It's not a problem to configure more Role mappers for the same LDAP provider. So for example you can specify that role mappings from groups under `ou=main,dc=example,dc=org` will be mapped to realm role mappings and role mappings from groups under `ou=finance,dc=example,dc=org` will be mapped to client role mappings of client `finance`.

## Hardcoded Role Mapper

This mapper will grant a specified KeyCloak role to each KeyCloak user linked with LDAP.

## Group Mapper

This allows you to configure group mappings from LDAP into KeyCloak group mappings. Group mapper can be used to map LDAP groups from a particular branch of an LDAP tree into groups in KeyCloak. It will also propagate user-group mappings from LDAP into user-group mappings in KeyCloak.

## MSAD User Account Mapper

This mapper is specific to Microsoft Active Directory (MSAD). It's

able to tightly integrate the MSAD user account state into the KeyCloak account state (account enabled, password is expired etc). It's using the `userAccountControl` and `pwdLastSet` LDAP attributes. (both are specific to MSAD and are not LDAP standard). For example if `pwdLastSet` is `0`, the KeyCloak user is required to update their password and there will be an `UPDATE_PASSWORD` required action added to the user. If `userAccountControl` is `514` (disabled account) the KeyCloak user is disabled as well.

By default, there are User Attribute mappers that map basic KeyCloak user attributes like `username`, `firstname`, `lastname`, and `email` to corresponding LDAP attributes. You are free to extend these and provide additional attribute mappings. Admin console provides tooltips, which should help with configuring the corresponding mappers.

#### 14.3.7. Password Hashing

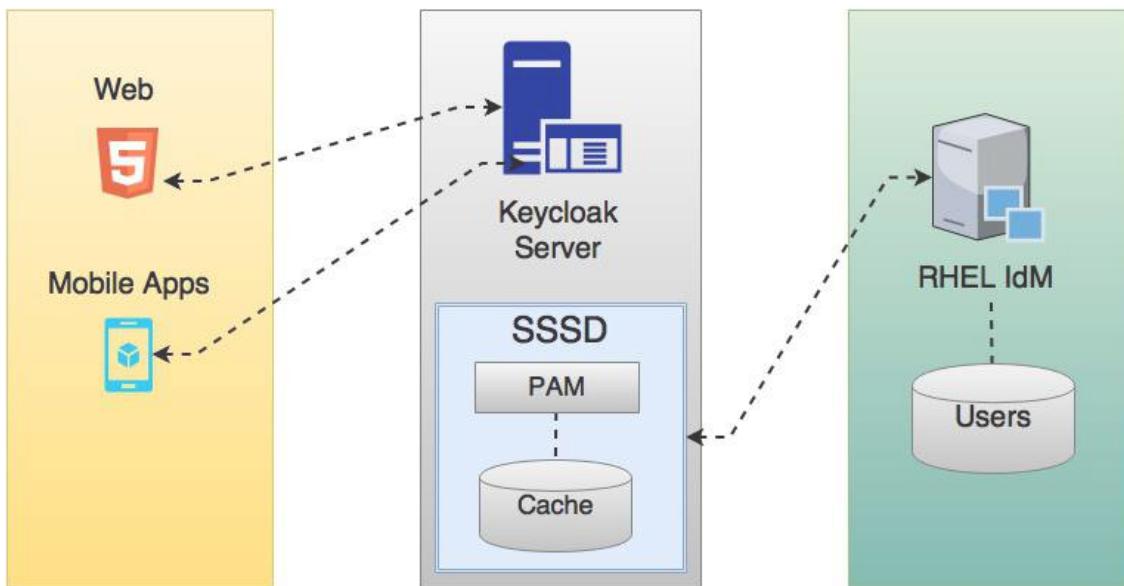
When the password of user is updated from KeyCloak and sent to LDAP, it is always sent in plain-text. This is different from updating the password to built-in KeyCloak database, when the hashing and salting is applied to the password before it is sent to DB. In the case of LDAP, the KeyCloak relies on the LDAP server to provide hashing and salting of passwords.

Most of LDAP servers (Microsoft Active Directory, RHDS, FreeIPA) provide this by default. Some others (OpenLDAP, ApacheDS) may store the passwords in plain-text by default and you may need to explicitly enable password hashing for them. See the documentation of your LDAP server more details.

## 14.4. SSSD and FreeIPA Identity Management Integration

KeyCloak also comes with a built-in [SSSD](#) (System Security Services Daemon) plugin. SSSD is part of the latest Fedora or Red Hat Enterprise Linux and provides access to multiple identity and authentication providers. It provides benefits such as failover and offline support. To see configuration options and for more information see [the Red Hat Enterprise Linux Identity Management documentation](#).

SSSD also integrates with the FreeIPA identity management (IdM) server, providing authentication and access control. For KeyCloak, we benefit from this integration authenticating against PAM services and retrieving user data from SSSD. For more information about using Red Hat Identity Management in Linux environments, see [the Red Hat Enterprise Linux Identity Management documentation](#).



Most of the communication between Keycloak and SSSD occurs through read-only D-Bus interfaces. For this reason, the only way to provision and update users is to use the FreeIPA/IdM administration in-

terface. By default, like the LDAP federation provider, it is set up only to import username, email, first name, and last name.

Groups and roles are automatically registered, but not synchronized, so any changes made by the KeyCloak administrator directly in KeyCloak is not synchronized with SSSD.

Information on how to configure the FreeIPA/IdM server follows.

#### 14.4.1. FreeIPA/IdM Server

As a matter of simplicity, a [FreeIPA Docker image](#) already available is used. To set up a server, see the [FreeIPA documentation](#).

Running a FreeIPA server with Docker requires this command:

```
docker run --name freeipa-server-container -it \
-h server.freeipa.local -e PASSWORD=YOUR_PASSWORD \
-v /sys/fs/cgroup:/sys/fs/cgroup:ro \
-v /var/lib/ipa-data:/data:Z freeipa/freeipa-server
```

The parameter `-h` with `server.freeipa.local` represents the FreeIPA/IdM server hostname. Be sure to change `YOUR_PASSWORD` to a password of your choosing.

After the container starts, change `/etc/hosts` to:

```
x.x.x.x      server.freeipa.local
```

If you do not make this change, you must set up a DNS server.

So that the SSSD federation provider is started and running on Keycloak you must enroll your Linux machine in the IPA domain:

```
ipa-client-install --mkhomedir -p admin -w password
```

To ensure that everything is working as expected, on the client machine, run:

```
kinit admin
```

You should be prompted for the password. After that, you can add users to the IPA server using this command:

```
$ ipa user-add john --first=John --last=Smith --email=john@smith.com --phone=042424242 --street="Testing street" \
--city="Testing city" --state="Testing State" --postal-code=0000000000
```

#### 14.4.2. SSSD and D-Bus

As mentioned previously, the federation provider obtains the data from SSSD using D-BUS and authentication occurs using PAM.

First, you have to install the `sssd-dbus` RPM, which allows information from SSSD to be transmitted over the system bus.

```
$ sudo yum install sssd-dbus
```

This script makes the necessary changes to `/etc/sssd/sssd.conf`:

```
[domain/your-hostname.local]
...
ldap_user_extra_attrs = mail:mail, sn:sn, givenname:given-
name, telephonenumber:telephonenumber
...
[sssd]
services = nss, sudo, pam, ssh, ifp
...
[ifp]
allowed_uids = root, yourOSUsername
user_attributes = +mail, +telephonenumber, +givenname, +sn
```

Also, a `keycloak` file is included under `/etc/pam.d/`:

```
auth    required    pam_sss.so
account required    pam_sss.so
```

Ensure everything is working as expected by running `dbus-send`:

```
sudo dbus-send --print-reply --system --dest=org.
freedesktop.sssd.infopipe /org/freedesktop/sssd/infopipe
org.freedesktop.sssd.infopipe.GetUserGroups string:john
```

You should be able to see the user's group. If this command returns a timeout or an error, it means that the federation provider will also not be able to retrieve anything on KeyCloak.

Most of the time this occurs because the machine was not enrolled in the FreeIPA IdM server or you do not have permission to access the SSSD service.

If you do not have permission, ensure that the user running KeyCloak is included in the `/etc/sssd/sssd.conf` file in the following section:

```
[ifp]
allowed_uids = root, your_username
```

#### 14.4.3. Enabling the SSSD Federation Provider

KeyCloak uses DBus-Java to communicate at a low level with D-Bus, which depends on the [Unix Sockets Library](#).

For authentication with PAM KeyCloak uses JNA. Be sure you have this package installed:

```
$ sudo yum install jna
```

Use `ssctl user-checks` command to validate your setup:

```
$ sudo sssctl user-checks admin -s keycloak
```

### 14.5. Configuring a Federated SSSD Store

After installation, you need to configure a federated SSSD store.

To configure a federated SSSD store, complete the following steps:

1. Navigate to the Administration Console.
2. From the left menu, select **User Federation**.
3. From the **Add Provider** dropdown list, select **sssd**. The `sssd` configuration page opens.
4. Click **Save**.

Now you can authenticate against Keycloak using FreeIPA/IdM credentials.

## 14.6. Custom Providers

Keycloak does have an SPI for User Storage Federation that you can use to write your own custom providers. You can find documentation for this in our [{developerguide!}](#).

---

# 15. Auditing and Events

Keycloak provides a rich set of auditing capabilities. Every single login action can be recorded and stored in the database and reviewed in the Admin Console. All admin actions can also be recorded and reviewed. There is also a Listener SPI with which plugins can listen for these events and perform some action. Built-in listeners include a simple log file and the ability to send an email if an event occurs.

## 15.1. Login Events

Login events occur for things like when a user logs in successfully, when somebody enters in a bad password, or when a user account is updated. Every single event that happens to a user can be recorded and viewed. By default, no events are stored or viewed in the Admin Console. Only error events are logged to the console and the server's log file. To start persisting you'll need to enable storage. Go to the `Events` left menu item and select the `Config` tab.

*Event Configuration*

The screenshot shows the Keycloak interface with the title 'Events Config' at the top. On the left, there's a sidebar with 'Master' selected. Under 'Configure', 'Events' is highlighted. Other options include Realm Settings, Clients, Client Templates, Roles, Identity Providers, User Federation, Authentication, Groups, Users, Sessions, and Import. The main content area has tabs for Login Events, Admin Events, and Config, with 'Config' currently active. It displays 'Event Listeners' with 'jboss-logging' selected. Below that are sections for 'Login Events Settings' and 'Admin Events Settings', each with a 'Save Events' switch set to 'OFF'.

To start storing events you'll need to turn the `Save Events` switch to `on` under the `Login Events Settings`.

### *Save Events*

The **Saved Types** field allows you to specify which event types you want to store in the event store. The **Clear events** button allows you to delete all the events in the database. The **Expiration** field allows you to specify how long you want to keep events stored. Once you've enabled storage of login events and decided on your settings, don't forget to click the **Save** button on the bottom of this page.

To view events, go to the **Login Events** tab.

## Login Events

The screenshot shows the Keycloak Admin UI with the 'Events' page selected. The left sidebar has 'Events' highlighted. The main area shows three events:

Time	Event Type	Details								
1/18/17 3:03:04 PM	CODE_TO_TOKEN	<table border="1"><tr><td>Client</td><td>security-admin-console</td></tr><tr><td>User</td><td>5e073544-a49b-45c5-bff2-a43679d4a4e9</td></tr><tr><td>IP Address</td><td>127.0.0.1</td></tr><tr><td>Details</td><td>[+]</td></tr></table>	Client	security-admin-console	User	5e073544-a49b-45c5-bff2-a43679d4a4e9	IP Address	127.0.0.1	Details	[+]
Client	security-admin-console									
User	5e073544-a49b-45c5-bff2-a43679d4a4e9									
IP Address	127.0.0.1									
Details	[+]									

| 1/18/17 3:03:04 PM | LOGIN | |            |                                      | |------------|--------------------------------------| | Client     | security-admin-console               | | User       | 5e073544-a49b-45c5-bff2-a43679d4a4e9 | | IP Address | 127.0.0.1                            | | Details    | [+]                                  | |
| 1/18/17 3:02:55 PM | LOGOUT | |            |                                      | |------------|--------------------------------------| | Client     |                                      | | User       | 5e073544-a49b-45c5-bff2-a43679d4a4e9 | | IP Address | 127.0.0.1                            | | Details    | [+]                                  | |

As you can see, there's a lot of information stored and, if you are storing every event, there are a lot of events stored for each login action. The **Filter** button on this page allows you to filter which events you are actually interested in.

## Login Event Filter

The screenshot shows the Keycloak Admin UI with the 'Events' page selected. The left sidebar has 'Events' highlighted. The main area shows a filtered list of events. The 'Event Type' filter is set to 'LOGIN'. Only one event is listed:

Time	Event Type	Details								
1/18/17 3:03:04 PM	LOGIN	<table border="1"><tr><td>Client</td><td>security-admin-console</td></tr><tr><td>User</td><td>5e073544-a49b-45c5-bff2-a43679d4a4e9</td></tr><tr><td>IP Address</td><td>127.0.0.1</td></tr><tr><td>Details</td><td>[+]</td></tr></table>	Client	security-admin-console	User	5e073544-a49b-45c5-bff2-a43679d4a4e9	IP Address	127.0.0.1	Details	[+]
Client	security-admin-console									
User	5e073544-a49b-45c5-bff2-a43679d4a4e9									
IP Address	127.0.0.1									
Details	[+]									

In this screenshot, we're filtering only **Login** events. Clicking the **Up-**

`date` button runs the filter.

### 15.1.1. Event Types

Login events:

- Login - A user has logged in.
- Register - A user has registered.
- Logout - A user has logged out.
- Code to Token - An application/client has exchanged a code for a token.
- Refresh Token - An application/client has refreshed a token.

Account events:

- Social Link - An account has been linked to a social provider.
- Remove Social Link - A social provider has been removed from an account.
- Update Email - The email address for an account has changed.
- Update Profile - The profile for an account has changed.
- Send Password Reset - A password reset email has been sent.
- Update Password - The password for an account has changed.
- Update TOTP - The TOTP settings for an account have changed.
- Remove TOTP - TOTP has been removed from an account.
- Send Verify Email - An email verification email has been sent.

- Verify Email - The email address for an account has been verified.

For all events there is a corresponding error event.

### 15.1.2. Event Listener

Event listeners listen for events and perform an action based on that event. There are two built-in listeners that come with Keycloak: Logging Event Listener and Email Event Listener.

The Logging Event Listener writes to a log file whenever an error event occurs and is enabled by default. Here's an example log message:

```
11:36:09,965 WARN [org.keycloak.events] (default task-51)
type=LOGIN_ERROR, realmId=master,
          clientId=myapp,
          userId=19aeb848-96fc-44f6-b0a3-
59a17570d374, ipAddress=127.0.0.1,
          error=invalid_user_credentials, auth_-
method=openid-connect, auth_type=code,
          redirect_uri=http://local-
host:8180/myapp,
          code_id=b669da14-cdbb-41d0-b055-
0810a0334607, username=admin
```

This logging is very useful if you want to use a tool like Fail2Ban to detect if there is a hacker bot somewhere that is trying to guess user passwords. You can parse the log file for `LOGIN_ERROR` and pull out the IP Address. Then feed this information into Fail2Ban so that it can help prevent attacks.

The Email Event Listener sends an email to the user's account when an event occurs. The Email Event Listener only supports the following events at the moment:

- Login Error
- Update Password
- Update TOTP
- Remove TOTP

To enable the Email Listener go to the `Config` tab and click on the `Event Listeners` field. This will show a drop down list box where you can select email.

You can exclude one or more events by editing the `standalone.xml`, `standalone-ha.xml`, or `domain.xml` that comes with your distribution and adding for example:

```
<spi name="eventsListener">
  <provider name="email" enabled="true">
    <properties>
      <property name="exclude-events" value="["UP-
DATE_TOTP","REMOVE_TOTP"]"/>
    </properties>
  </provider>
</spi>
```

See the [{installguide!}](#) for more details on where the `standalone.xml`, `standalone-ha.xml`, or `domain.xml` file lives.

## 15.2. Admin Events

Any action an admin performs within the admin console can be recorded for auditing purposes. The Admin Console performs administrative functions by invoking on the Keycloak REST interface. Keycloak audits these REST invocations. The resulting events can then be viewed in

the Admin Console.

To enable auditing of Admin actions, go to the `Events` left menu item and select the `Config` tab.

### *Event Configuration*

The screenshot shows the Keycloak Admin Console interface. The top navigation bar has the Keycloak logo and the text "KEYCLOAK". Below it is a dark sidebar with a dropdown menu set to "Master". The sidebar contains two main sections: "Configure" and "Manage". Under "Configure", the "Events" option is highlighted with a blue underline. Under "Manage", the "Events" option is also present. The main content area is titled "Events Config" with a help icon. It has three tabs: "Login Events", "Admin Events", and "Config", with "Config" being the active tab. The "Events Config" section contains a "Event Listeners" field containing "jboss-logging", a "Login Events Settings" section with a "Save Events" switch set to "OFF", and an "Admin Events Settings" section with a "Save Events" switch set to "OFF".

In the `Admin Events Settings` section, turn on the `Save Events` switch.

### *Admin Event Configuration*

The `Include Representation` switch will include any JSON document that is sent through the admin REST API. This allows you to view exactly what an admin has done, but can lead to a lot of information stored in the database. The `Clear admin events` button allows you to wipe out the current information stored.

To view the admin events go to the `Admin Events` tab.

## *Admin Events*

Time	Operation Type	Resource Type	Resource Path	Details
1/18/17 3:10:37 PM	CREATE	CLIENT	clients/0545b728-532b-4512-94be-d2347200ae76	<a href="#">Auth</a> <a href="#">Representation</a>
1/18/17 3:10:17 PM	ACTION	USER	users/4c1c44a8-e2f6-417c-9d52-624b7b9eb76a/reset-password	<a href="#">Auth</a>

If the `Details` column has a `Representation` box, you can click on that to view the JSON that was sent with that operation.

## Admin Representation

The screenshot shows the Keycloak Admin interface. On the left, there's a sidebar with 'Master' selected. Under 'Configure', 'Events' is highlighted. The main area shows 'Admin Events' with a 'Login Events' tab selected. A modal dialog is open, displaying a JSON object representing an event:

```
{ "clientId": "myapp2", "rootUrl": "http://localhost:8080/myapp2", "enabled": true, "redirectUris": [], "protocol": "openid-connect", "attributes": {} }
```

Below the modal, a table lists events:

Time	Operation Type	Resource Type	Details
1/18/17 3:10:37 PM	CREATE	CLIENT	client/05169728-5120-4512-9abe-d2347200ae76
1/18/17 3:10:17 PM	ACTION	USER	users/4c1c44a8-e2f6-417c-9d52-624b7b9eb76a/reset-password

Buttons at the bottom right of the modal include '+ Filter', 'Update', and 'Reset'.

You can also filter for the events you are interested in by clicking the **Filter** button.

## Admin Event Filter

The screenshot shows the Keycloak Admin interface with 'Events' selected in the sidebar. The main area displays the 'Admin Events' filter form:

**Operation Types:** ACTION

**Resource Types:** Select resource types...

**Resource Path:** (empty)

**Date (From):** yyyy-MM-dd

**Date (To):** yyyy-MM-dd

**Authentication Details:**

**Realm:** (empty)

**Client:** (empty)

**User:** (empty)

**IP Address:** (empty)

At the bottom, a table shows the filtered results:

Time	Operation Type	Resource Type	Resource Path	Details
1/18/17 3:10:17 PM	ACTION	USER	users/4c1c44a8-e2f6-417c-9d52-624b7b9eb76a/reset-password	Auth

---

## 16. Export and Import

Keycloak has the ability to export and import the entire database. This can be especially useful if you want to migrate your whole Keycloak database from one environment to another or migrate to a different database (for example from MySQL to Oracle). Export and import is triggered at server boot time and its parameters are passed in via Java system properties. It is important to note that because import and export happens at server startup, no other actions should be taken on the server or the database while this happens.

You can export/import your database either to:

- Directory on local filesystem
- Single JSON file on your filesystem

When importing using the directory strategy, note that the files need to follow the naming convention specified below. If you are importing files which were previously exported, the files already follow this convention.

- <REALM\_NAME>-realm.json, such as "acme-roadrunner-affairs-realm.json" for the realm named "acme-roadrunner-affairs"
- <REALM\_NAME>-users-<INDEX>.json, such as "acme-roadrunner-affairs-users-0.json" for the first users file of the realm named "acme-roadrunner-affairs"

If you export to a directory, you can also specify the number of users that will be stored in each JSON file.

If you have bigger amount of users in your database (500 or more), it's highly recommended to export into directory rather than to single file. Exporting into single file may lead to the very big file. Also the directory provider is using separate transaction for each "page" (file with users), which leads to much better performance. Default count of users per file (and transaction) is 50, which showed us best performance, but you have possibility to override (See below). Exporting to single file is using one transaction per whole export and one per whole import, which results in bad performance with large amount of users.

To export into unencrypted directory you can use:

```
bin/standalone.sh -Dkeycloak.migration.action=export  
-Dkeycloak.migration.provider=dir -Dkeycloak.migration.dir=  
<DIR TO EXPORT TO>
```

And similarly for import just use `-Dkeycloak.migration.action=import` instead of `export`. To export into single JSON file you can use:

```
bin/standalone.sh -Dkeycloak.migration.action=export  
-Dkeycloak.migration.provider=singleFile -Dkeycloak.  
migration.file=<FILE TO EXPORT TO>
```

Here's an example of importing:

```
bin/standalone.sh -Dkeycloak.migration.action=import  
-Dkeycloak.migration.provider=singleFile -Dkeycloak.  
migration.file=<FILE TO IMPORT>  
-Dkeycloak.migration.strategy=OVERWRITE_EXISTING
```

Other available options are:

#### **-Dkeycloak.migration.realmName**

This property is used if you want to export just one specified realm instead of all. If not specified, then all realms will be exported.

#### **-Dkeycloak.migration.usersExportStrategy**

This property is used to specify where users are exported. Possible values are:

- DIFFERENT\_FILES - Users will be exported into different files according to the maximum number of users per file. This is default value.
- SKIP - Exporting of users will be skipped completely.
- REALM\_FILE - All users will be exported to same file with the realm settings. (The result will be a file like "foo-realm.json" with both realm data and users.)
- SAME\_FILE - All users will be exported to same file but different from the realm file. (The result will be a file like "foo-realm.json" with realm data and "foo-users.json" with users.)

#### **-Dkeycloak.migration.usersPerFile**

This property is used to specify the number of users per file (and also per DB transaction). It's 50 by default. It's used only if usersExportStrategy is DIFFERENT\_FILES

### **-Dkeycloak.migration.strategy**

This property is used during import. It can be used to specify how to proceed if a realm with same name already exists in the database where you are going to import data. Possible values are:

- IGNORE\_EXISTING - Ignore importing if a realm of this name already exists.
- OVERWRITE\_EXISTING - Remove existing realm and import it again with new data from the JSON file. If you want to fully migrate one environment to another and ensure that the new environment will contain the same data as the old one, you can specify this.

When importing realm files that weren't exported before, the option `keycloak.import` can be used. If more than one realm file needs to be imported, a comma separated list of file names can be specified. This is more appropriate than the cases before, as this will happen only after the master realm has been initialized. Examples:

- `-Dkeycloak.import=/tmp/realm1.json`
- `-Dkeycloak.import=/tmp/realm1.json,/tmp/realm2.json`

## **16.1. Admin console export/import**

Import of most resources can be performed from the admin console as

well as export of most resources. Export of users is not supported.

Note: Attributes containing secrets or private information will be masked in export file. Export files obtained via Admin Console are thus not appropriate for backups or data transfer between servers. Only boot-time exports are appropriate for that.

The files created during a "startup" export can also be used to import from the admin UI. This way, you can export from one realm and import to another realm. Or, you can export from one server and import to another. Note: The admin console export/import allows just one realm per file.

The admin console import allows you to "overwrite" resources if you choose. Use this feature with caution, especially on a production system. Export .json files from Admin Console Export operation are generally not appropriate for data import since they contain invalid values for secrets.

The admin console export allows you to export clients, groups, and roles. If there is a great number of any of these assets in your realm, the operation may take some time to complete. During that time server may not be responsive to user requests. Use this feature with caution, especially on a production system.

# 17. User Account Service

Keycloak has a built-in User Account Service which every user has access to. This service allows users to manage their account, change their credentials, update their profile, and view their login sessions. The URL to this service is `<server-root>/auth/realms/{realm-name}/account`.

## *Account Service*

The screenshot shows the 'Edit Account' page in the Keycloak interface. The left sidebar has a 'Account' item selected. The main form contains the following fields:

Edit Account		* Required fields
Username	admin	
Email *	john@example.com	
First name *	John	
Last name *	Doe	<input type="button" value="User"/>

At the bottom right are 'Cancel' and 'Save' buttons.

The initial page is the user's profile, which is the `Account` left menu item. This is where they specify basic data about themselves. This screen can be extended to allow the user to manage additional attributes. See the [{developerguide!}](#) for more details.

The `Password` left menu item allows the user to change their password.

## *Password Update*

The screenshot shows the Keycloak Security Admin Console interface. The left sidebar has a 'Account' section with 'Password' selected, and other options like 'Authenticator', 'Sessions', and 'Applications'. The main area is titled 'Change Password' with a note 'All fields required'. It contains three input fields: 'Password', 'New Password', and 'Confirmation', each with a length indicator. A 'Save' button is at the bottom right.

The **Authenticator** menu item allows the user to set up OTP if they desire. This will only show up if OTP is a valid authentication mechanism for your realm. Users are given directions to install [FreeOTP](#) or [Google Authenticator](#) on their mobile device to be their OTP generator. The QR code you see in the screen shot can be scanned into the FreeOTP or Google Authenticator mobile application for nice and easy setup.

### *OTP Authenticator*

The screenshot shows the Keycloak Security Admin Console interface. The left sidebar has a navigation menu with items: Account, Password, **Authenticator**, Sessions, and Applications. The **Authenticator** item is selected and highlighted with a blue background. The main content area is titled "Authenticator". It contains instructions: "1. Install FreeOTP or Google Authenticator on your device. Both applications are available in [Google Play](#) and Apple App Store." and "2. Open the application and scan the barcode or enter the key." Below these instructions is a QR code. Underneath the QR code is a string of characters: "NQ4T OUDH KEZG YSZQ PI2F AZ3D I4YU 42DL". Further down, it says "3. Enter the one-time code provided by the application and click Save to finish the setup." There is a text input field labeled "One-time code" and two buttons at the bottom right: "Cancel" and "Save".

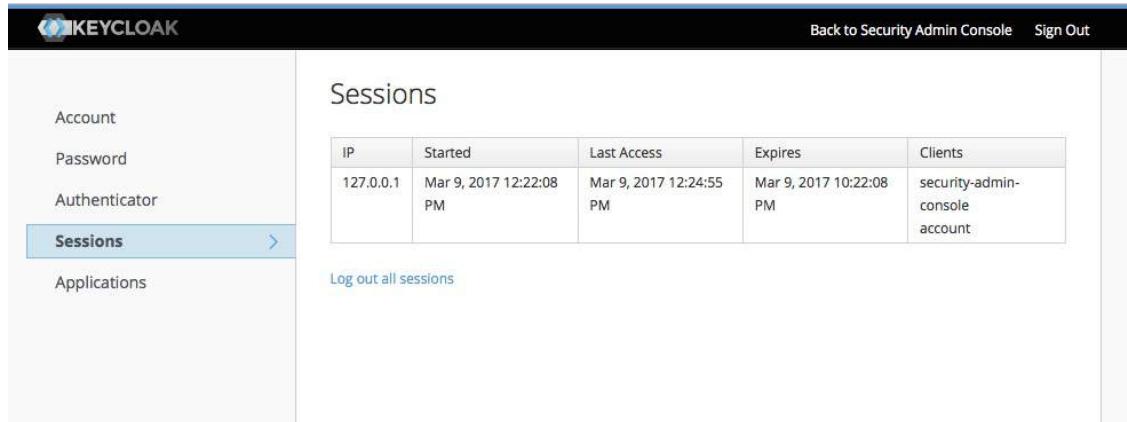
The **Federated Identity** menu item allows the user to link their account with an [identity broker](#) (this is usually used to link social provider accounts together). This will show the list of external identity providers you have configured for your realm.

### *Federated Identity*

The screenshot shows the Keycloak Security Admin Console interface. The left sidebar has a navigation menu with items: Account, Password, Authenticator, **Federated Identity**, Sessions, and Applications. The **Federated Identity** item is selected and highlighted with a blue background. The main content area is titled "Federated Identities". It lists several external identity providers with input fields and "Add" buttons: facebook, linkedin, stackoverflow, and saml. Each provider has a corresponding input field and an "Add" button to its right.

The `Sessions` menu item allows the user to view and manage which devices are logged in and from where. They can perform logout of these sessions from this screen too.

## *Sessions*



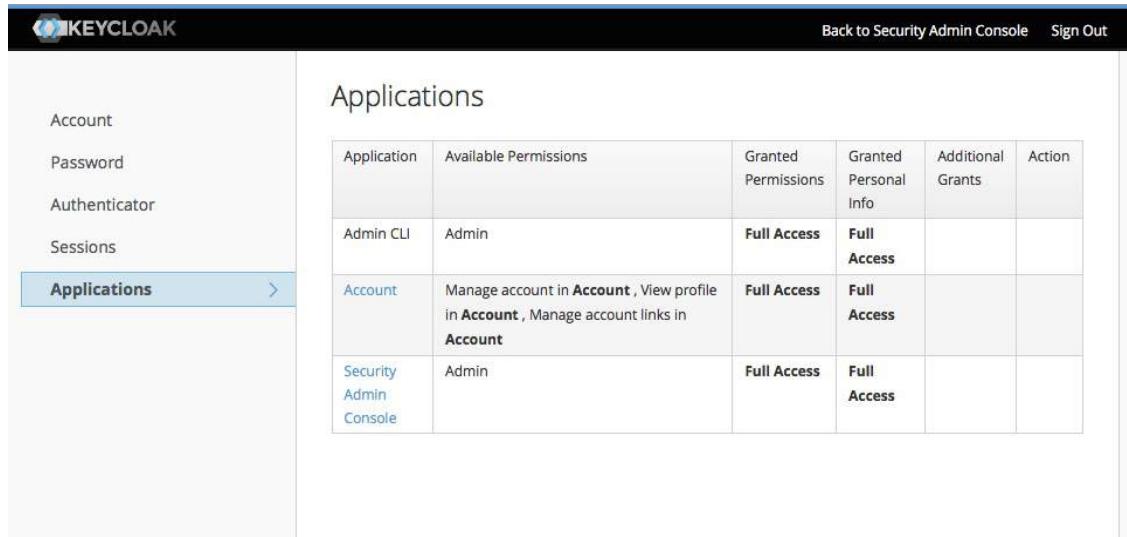
The screenshot shows the 'Sessions' page of the Keycloak User Account Service. The left sidebar has a 'Sessions' item selected. The main content area is titled 'Sessions' and contains a table with one row of data:

IP	Started	Last Access	Expires	Clients
127.0.0.1	Mar 9, 2017 12:22:08 PM	Mar 9, 2017 12:24:55 PM	Mar 9, 2017 10:22:08 PM	security-admin-console account

A link 'Log out all sessions' is visible below the table.

The `Applications` menu item shows users which applications they have access to.

## *Applications*



The screenshot shows the 'Applications' page of the Keycloak User Account Service. The left sidebar has an 'Applications' item selected. The main content area is titled 'Applications' and contains a table with four rows of data:

Application	Available Permissions	Granted Permissions	Granted Personal Info	Additional Grants	Action
Admin CLI	Admin	Full Access	Full Access		
Account	Manage account in <b>Account</b> , View profile in <b>Account</b> , Manage account links in <b>Account</b>	Full Access	Full Access		
Security Admin Console	Admin	Full Access	Full Access		

## 17.1. Themeable

Like all UIs in KeyCloak, the User Account Service is completely themeable and internationalizable. See the [{developerguide!}](#) for more de-

tails.

---

# 18. Threat Model Mitigation

This chapter discusses possible security vulnerabilities any authentication server could have and how Keycloak mitigates those vulnerabilities. A good list of potential vulnerabilities and what security implementations should do to mitigate them can be found in the [OAuth 2.0 Threat Model](#) document put out by the IETF. Many of those vulnerabilities are discussed here.

## 18.1. Host

Keycloak uses the public hostname for a number of things. For example, in the token issuer fields and URLs sent in password reset emails.

By default, the hostname is based on the request headers and there is no check to make sure this hostname is valid.

If you are not using a load balancer or proxy in front of Keycloak that prevents invalid host headers, you must explicitly configure what hostnames should be accepted.

The Hostname SPI provides a way to configure the hostname for a request. Out of the box there are two providers. These are request and fixed. It is also possible to develop your own provider in the case the built-in providers do not provide the functionality needed.

### 18.1.1. Request provider

This is the default hostname provider and uses request headers to determine the hostname. As it uses the headers from the request it is im-

portant to use this in combination with a proxy or a filter that rejects invalid hostnames.

It is beyond the scope of this documentation to provide instructions on how to configure valid hostnames for a proxy. To configure it in a filter you need to edit standalone.xml to set permitted aliases for the server. The following example will only permit requests to `auth.example.com`:

```
<subsystem xmlns="{subsystem_undertow_xml_urn}">
    <server name="default-server" default-host="ignore">
        ...
        <host name="default-host" alias="auth.example.com">
            <location name="/" handler="welcome-content"/>
            <http-invoker security-realm="ApplicationRe-
alrn"/>
        </host>
    </server>
</subsystem>
```

The changes that have been made from the default config is to add the attribute `default-host="ignore"` and update the attribute `alias`. `default-host="ignore"` prevents unknown hosts from being handled, while `alias` is used to list the accepted hosts.

Here is the equivalent configuration using CLI commands:

```
/subsystem=undertow/server=default-server:write-attribu-
te(name=default-host,value=ignore)
/subsystem=undertow/server=default-server/host=default-
host:write-attribute(name=alias,value=[auth.example.com])

:reload
```

### 18.1.2. Fixed provider

The fixed provider makes it possible to configure a fixed hostname. Unlike the request provider the fixed provider allows internal applications to invoke Keycloak on an alternative URL (for example an internal IP address). It is also possible to override the hostname for a specific realm through the configuration of the realm in the admin console.

This is the recommended provider to use in production.

To change to the fixed provider and configure the hostname edit `standalone.xml`. The following example shows the fixed provider with the hostname set to `auth.example.com`:

```
<spi name="hostname">
    <default-provider>fixed</default-provider>
    <provider name="fixed" enabled="true">
        <properties>
            <property name="hostname" value="auth.example.
com"/>
            <property name="httpPort" value="-1"/>
            <property name="httpsPort" value="-1"/>
        </properties>
    </provider>
</spi>
```

Here is the equivalent configuration using CLI commands:

```
/subsystem=keycloak-server/spi=hostname:write-attri-
    bu-
te(name=default-provider, value="fixed")
/subsystem=keycloak-server/spi=hostname/provider=fixed:wri-
    te-attribute(name=properties.hostname,value="auth.example.
com")
```

By default the `httpPort` and `httpsPort` are received from the request. As long as any proxies are configured correctly it should not be necessary to change this. It is possible to configure fixed ports if necessary by setting the `httpPort` and `httpsPort` properties on the fixed provider.

In most cases the scheme should be set correctly. This may not be true if the reverse proxy is unable to set the `X-Forwarded-For` header correctly, or if there is an internal application using non-https to invoke KeyCloak. In such cases it is possible to set the `alwaysHttps` to `true`.

### 18.1.3. Custom provider

To develop a custom hostname provider you need to implement `org.keycloak.urls.HostnameProviderFactory` and `org.keycloak.urls.HostnameProvider`.

Follow the instructions in the Service Provider Interfaces section in [{developerguide!}](#) for more information on how to develop a custom provider.

## 18.2. Admin Endpoints and Console

The KeyCloak administrative REST API and the web console are exposed by default on the same port as non-admin usage. If you are exposing KeyCloak on the Internet, we recommend not also exposing the admin endpoints on the Internet.

This can be achieved either directly in KeyCloak or with a proxy such as Apache or nginx.

For the proxy option please follow the documentation for the proxy.

You need to control access to any requests to `/auth/admin`.

To achieve this directly in Keycloak there are a few options. This document covers two options, IP restriction and separate ports.

### 18.2.1. IP Restriction

It is possible to restrict access to `/auth/admin` to only specific IP addresses.

The following example restricts access to `/auth/admin` to IP addresses in the range `10.0.0.1` to `10.0.0.255`.

```
<subsystem xmlns="{subsystem_undertow_xml_urn}">
  ...
    <server name="default-server">
      ...
        <host name="default-host" alias="localhost">
          ...
            <filter-ref name="ipAccess"/>
          </host>
        </server>
        <filters>
          <expression-filter name="ipAccess" expression="path-prefix('/auth/admin') -> ip-access-control(acl=['10.0.0.0/24 allow'])"/>
        </filters>
      ...
    </subsystem>
```

Equivalent configuration using CLI commands:

```
/subsystem=undertow/configuration=filter/expression-filter=ipAccess:add(expression="path-prefix[/auth/admin] -> ip-access-control(acl=['10.0.0.0/24 allow']))
```

```
/subsystem=undertow/server=default-server/host=default-host/filter-ref=ipAccess:add()
```

For IP restriction if you are using a proxy it is important to configure it correctly to make sure Keycloak receives the client IP address and not the proxy IP address

### 18.2.2. Port Restriction

It is possible to expose `/auth/admin` to a different port that is not exposed on the Internet.

The following example exposes `/auth/admin` on port `8444` while not permitting access with the default port `8443`.

```
<subsystem xmlns="{subsystem_undertow_xml_urn}">
  ...
  <server name="default-server">
    ...
      <https-listener name="https" socket-binding="https"
security-realm="ApplicationRealm" enable-http2="true"/>
      <https-listener name="https-admin" socket-binding="https-admin"
security-realm="ApplicationRealm"
enable-http2="true"/>
      <host name="default-host" alias="localhost">
        ...
          <filter-ref name="portAccess"/>
        </host>
      </server>
      <filters>
        <expression-filter name="portAccess" expression="path-prefix('/auth/admin') and not equals(%p, 8444) ->
response-code(403)"/>
      </filters>
    ...
  </subsystem>
```

```
...
<socket-binding-group name="standard-sockets" default-interface="public" port-offset="${jboss.socket.binding.port-offset:0}">
    ...
        <socket-binding name="https" port="${jboss.https.port:8443}" />
        <socket-binding name="https-admin" port="${jboss.https.port:8444}" />
    ...
</socket-binding-group>
```

Equivalent configuration using CLI commands:

```
/socket-binding-group=standard-sockets/socket-binding=https-admin/:add(port=8444)

/subsystem=undertow/server=default-server/https-listener=https-admin:add(socket-binding=https-admin, security-realm=ApplicationRealm, enable-http2=true)

/subsystem=undertow/configuration=filter/expression-filter=portAccess:add(, expression="path-prefix('/auth/admin') and not equals(%p, 8444) -> response-code(403)") 
/subsystem=undertow/server=default-server/host=default-host/filter-ref=portAccess:add()
```

### 18.3. Password guess: brute force attacks

A brute force attack happens when an attacker is trying to guess a user's password. Keycloak has some limited brute force detection capabilities. If turned on, a user account will be temporarily disabled if a threshold of login failures is reached. To enable this feature go to the `Realm Settings` left menu item, click on the `Security Defenses` tab, then additional go to the `Brute Force Detection` sub-tab.

## Brute Force Detection

The screenshot shows the Keycloak Admin UI with the 'Master' realm selected. The left sidebar has 'Realm Settings' selected under 'Configure'. The main tab 'Security Defenses' is active, and the sub-tab 'Brute Force Detection' is selected. The configuration includes:

- Enabled: ON
- Max Login Failures: 30
- Wait Increment: 1 Minutes
- Quick Login Check Milli Seconds: 1000
- Minimum Quick Login Wait: 1 Minutes
- Max Wait: 15 Minutes
- Failure Reset Time: 12 Hours

Buttons at the bottom are 'Save' and 'Cancel'.

There are 2 different configurations for brute force detection; permanent lockout and temporary lockout. Permanent lockout will disable a user's account after an attack is detected; the account will be disabled until an administrator renables it. Temporary lockout will disable a user's account for a time period after an attack is detected; the time period for which the account is disabled increases the longer the attack continues.

### Common Parameters

#### Max Login Failures

Maximum number of login failures permitted. Default value is 30.

#### Quick Login Check Milli Seconds

Minimum time required between login attempts. Default is 1000.

#### Minimum Quick Login Wait

Minimum amount of time the user will be temporarily disabled if logins attempts are quicker than *Quick Login Check Milli Seconds*. Default is 1 minute.

## Temporary Lockout Parameters

### Wait Increment

Amount of time added to the time a user is temporarily disabled after each time *Max Login Failures* is reached. Default is 1 minute.

### Max Wait

The maximum amount of time for which a user will be temporarily disabled. Default is 15 minutes.

### Failure Reset Time

Time after which the failure count will be reset; timer runs from the last failed login. Default is 12 hours.

## Permanent Lockout Algorithm

1. On successful login
  - a. Reset count
2. On failed login
  - a. Increment count

- b. If `count` greater than *Max Login Failures*
  - i. Permanently disable user
- c. Else if time between this failure and the last failure is less than *Quick Login Check Milli Seconds*
  - i. Temporarily disable user for *Minimum Quick Login Wait*

When a user is disabled they can not login until an administrator enables the user; enabling an account resets `count`.

## **Temporary Lockout Algorithm**

- 1. On successful login
  - a. Reset `count`
- 2. On failed login
  - a. If time between this failure and the last failure is greater than *Failure Reset Time*
    - i. Reset `count`
  - b. Increment `count`
  - c. Calculate `wait` using  $\text{Wait Increment} * (\text{count} / \text{Max Login Failures})$ . The division is an integer division so will always be rounded down to a whole number
  - d. If `wait` equals 0 and time between this failure and the last failure is less than *Quick Login Check Milli Seconds*

then set `wait` to *Minimum Quick Login Wait* instead

- i. Temporarily disable the user for the smaller of `wait` and *Max Wait* seconds

Login failures when a user is temporarily disabled do not increment `count`.

The downside of KeyCloak brute force detection is that the server becomes vulnerable to denial of service attacks. An attacker can simply try to guess passwords for any accounts it knows and these account will be disabled. Eventually we will expand this functionality to take client IP address into account when deciding whether to block a user.

A better option might be a tool like [Fail2Ban](#). You can point this service at the KeyCloak server's log file. KeyCloak logs every login failure and client IP address that had the failure. Fail2Ban can be used to modify firewalls after it detects an attack to block connections from specific IP addresses.

### 18.3.1. Password Policies

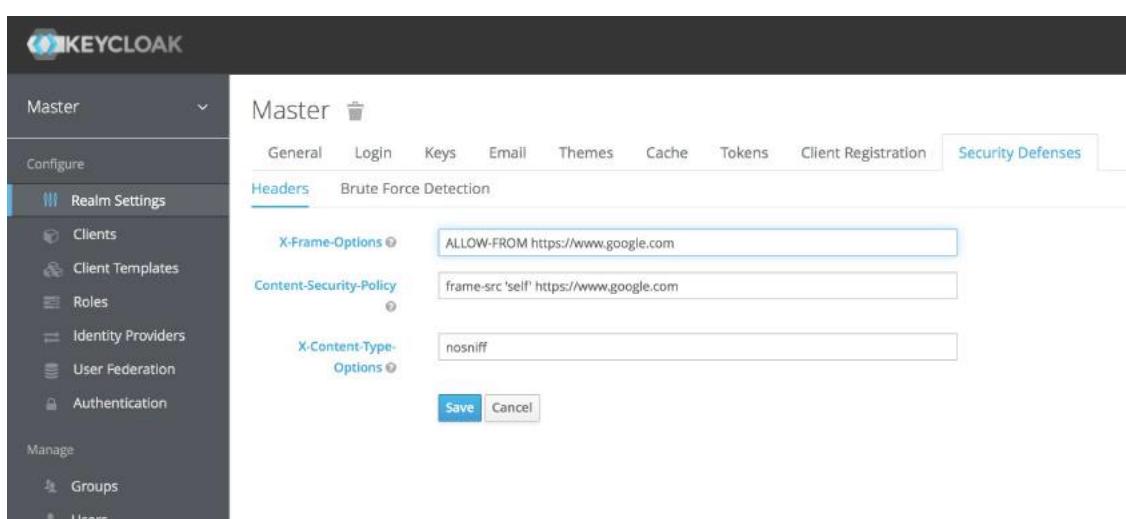
Another thing you should do to prevent password guess is to have a complex enough password policy to ensure that users pick hard to guess passwords. See the [Password Policies](#) chapter for more details.

The best way to prevent password guessing though is to set up the server to use a one-time-password (OTP).

## 18.4. Clickjacking

With clickjacking, a malicious site loads the target site in a transparent iFrame overlaid on top of a set of dummy buttons that are carefully constructed to be placed directly under important buttons on the target site. When a user clicks a visible button, they are actually clicking a button (such as a "login" button) on the hidden page. An attacker can steal a user's authentication credentials and access their resources.

By default, every response by KeyCloak sets some specific browser headers that can prevent this from happening. Specifically, it sets [X-FRAME-OPTIONS](#) and [Content-Security-Policy](#). You should take a look at the definition of both of these headers as there is a lot of fine-grain browser access you can control. In the admin console you can specify the values these headers will have. Go to the `Realm Settings` left menu item and click the `Security Defenses` tab and make sure you are on the `Headers` sub-tab.



The screenshot shows the Keycloak Admin Console interface. The left sidebar has a 'Master' dropdown and sections for 'Configure' (selected), 'Realm Settings' (selected), 'Clients', 'Client Templates', 'Roles', 'Identity Providers', 'User Federation', 'Authentication', 'Manage' (Groups, Users), and 'Logout'. The main content area has a 'Master' title and tabs for General, Login, Keys, Email, Themes, Cache, Tokens, Client Registration, and Security Defenses (selected). Under 'Headers', there are three fields: 'X-Frame-Options' with value 'ALLOW-FROM https://www.google.com', 'Content-Security-Policy' with value 'frame-src \'self\' https://www.google.com', and 'X-Content-Type-Options' with value 'nosniff'. At the bottom are 'Save' and 'Cancel' buttons.

By default, KeyCloak only sets up a *same-origin* policy for iframes.

## 18.5. SSL/HTTPS Requirement

If you do not use SSL/HTTPS for all communication between the Key-

Cloak auth server and the clients it secures, you will be very vulnerable to man in the middle attacks. OAuth 2.0/OpenID Connect uses access tokens for security. Without SSL/HTTPS, attackers can sniff your network and obtain an access token. Once they have an access token they can do any operation that the token has been given permission for.

KeyCloak has [three modes for SSL/HTTPS](#). SSL can be hard to set up, so out of the box, KeyCloak allows non-HTTPS communication over private IP addresses like localhost, 192.168.x.x, and other private IP addresses. In production, you should make sure SSL is enabled and required across the board.

On the adapter/client side, KeyCloak allows you to turn off the SSL trust manager. The trust manager ensures identity the client is talking to. It checks the DNS domain name against the server's certificate. In production you should make sure that each of your client adapters is configured to use a truststore. Otherwise you are vulnerable to DNS man in the middle attacks.

## 18.6. CSRF Attacks

Cross-site request forgery (CSRF) is a web-based attack whereby HTTP requests are transmitted from a user that the web site trusts or has authenticated with(e.g. via HTTP redirects or HTML forms). Any site that uses cookie based authentication is vulnerable to these types of attacks. These attacks are mitigated by matching a state cookie against a posted form or query parameter.

The OAuth 2.0 login specification requires that a state cookie be used and matched against a transmitted state parameter. KeyCloak fully im-

plements this part of the specification so all logins are protected.

The KeyCloak Admin Console is a pure JavaScript/HTML5 application that makes REST calls to the backend KeyCloak admin REST API. These calls all require bearer token authentication and are made via JavaScript Ajax calls. CSRF does not apply here. The admin REST API can also be configured to validate the CORS origins as well.

The only part of KeyCloak that really falls into CSRF is the user account management pages. To mitigate this KeyCloak sets a state cookie and also embeds the value of this state cookie within hidden form fields or query parameters in action links. This query or form parameter is checked against the state cookie to verify that the call was made by the user.

## 18.7. Unspecific Redirect URIs

For the [Authorization Code Flow](#), if you register redirect URIs that are too general, then it would be possible for a rogue client to impersonate a different client that has a broader scope of access. This could happen for instance if two clients live under the same domain. So, it's a good idea to make your registered redirect URIs as specific as feasible.

## 18.8. Compromised Access and Refresh Tokens

There are a few things you can do to mitigate access tokens and refresh tokens from being stolen. The most important thing is to enforce SSL/HTTPS communication between KeyCloak and its clients and applications. It might seem obvious, but since KeyCloak does not have SSL enabled by default, an administrator might not realize that it is necessary.

Another thing you can do to mitigate leaked access tokens is to shorten their lifespans. You can specify this within the [timeouts page](#). Short lifespans (minutes) for access tokens for clients and applications to refresh their access tokens after a short amount of time. If an admin detects a leak, they can logout all user sessions to invalidate these refresh tokens or set up a revocation policy. Making sure refresh tokens always stay private to the client and are never transmitted ever is very important as well.

You can also mitigate against leaked access tokens and refresh tokens by issuing these tokens as holder-of-key tokens. See [OAuth 2.0 Mutual TLS Client Certificate Bound Access Token](#) to learn how.

If an access token or refresh token is compromised, the first thing you should do is go to the admin console and push a not-before revocation policy to all applications. This will enforce that any tokens issued prior to that date are now invalid. Pushing new not-before policy will also ensure that application will be forced to download new public keys from KeyCloak, hence it is also useful for the case, when you think that realm signing key was compromised. More info in the [keys chapter](#).

You can also disable specific applications, clients, and users if you feel that any one of those entities is completely compromised.

## 18.9. Compromised Authorization Code

For the [OIDC Auth Code Flow](#), it would be very hard for an attacker to compromise KeyCloak authorization codes. KeyCloak generates a cryptographically strong random value for its authorization codes so it would be very hard to guess an access token. An authorization code can

only be used once to obtain an access token. In the admin console you can specify how long an authorization code is valid for on the [timeouts page](#). This value should be really short, as short as a few seconds and just long enough for the client to make the request to obtain a token from the code.

## 18.10. Open redirectors

An attacker could use the end-user authorization endpoint and the redirect URI parameter to abuse the authorization server as an open redirector. An open redirector is an endpoint using a parameter to automatically redirect a user agent to the location specified by the parameter value without any validation. An attacker could utilize a user's trust in an authorization server to launch a phishing attack.

KeyCloak requires that all registered applications and clients register at least one redirection URI pattern. Any time a client asks KeyCloak to perform a redirect (on login or logout for example), KeyCloak will check the redirect URI vs. the list of valid registered URI patterns. It is important that clients and applications register as specific a URI pattern as possible to mitigate open redirector attacks.

## 18.11. Password database compromised

KeyCloak does not store passwords in raw text. It stores a hash of them using the PBKDF2 algorithm. It actually uses a default of 20,000 hashing iterations! This is the security community's recommended number of iterations. This can be a rather large performance hit on your system as PBKDF2, by design, gobbles up a significant amount of CPU. It is up to you to decide how serious you want to be to protect your pass-

word database.

## 18.12. Limiting Scope

By default, each new client application has an unlimited `role scope mappings`. This means that every access token that is created for that client will contain all the permissions the user has. If the client gets compromised and the access token is leaked, then each system that the user has permission to access is now also compromised. It is highly suggested that you limit the roles an access token is assigned by using the [Scope menu](#) for each client. Or alternatively, you can set role scope mappings at the Client Scope level and assign Client Scopes to your client by using the [Client Scope menu](#).

## 18.13. Limit Token Audience

In environments where the level of trust among services is low, it is a good practice to limit the audiences on the token. The motivation behind this is described in the [OAuth2 Threat Model](#) document and more details are in the [Audience Support section](#).

## 18.14. SQL Injection Attacks

At this point in time, there is no knowledge of any SQL injection vulnerabilities in KeyCloak.

---

# 19. The Admin CLI

In previous chapters, we described how to use the Keycloak Admin Console to perform administrative tasks. You can also perform those tasks from the command-line interface (CLI) by using the Admin CLI command-line tool.

## 19.1. Installing the Admin CLI

The Admin CLI is packaged inside Keycloak Server distribution. You can find execution scripts inside the `bin` directory.

The Linux script is called `kcadm.sh`, and the script for Windows is called `kcadm.bat`.

You can add the Keycloak server directory to your `PATH` to use the client from any location on your file system.

For example, on:

- Linux:

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

- Windows:

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

We assume the `KEYCLOAK_HOME` environment (env) variable is set to the path where you extracted the Keycloak Server distribution.

To avoid repetition, the rest of this document only gives Windows examples in places where the difference in the CLI is more than just in the `kcadm` command name.

## 19.2. Using the Admin CLI

The Admin CLI works by making HTTP requests to Admin REST endpoints. Access to them is protected and requires authentication.

Consult the Admin REST API documentation for details about JSON attributes for specific endpoints.

1. Start an authenticated session by providing credentials, that is, logging in. You are ready to perform create, read, update, and delete (CRUD) operations.

For example, on

- Linux:

```
$ kcadm.sh config credentials --server http://localhost:8080  
$ kcadm.sh create realms -s realm=demorealm -s enabled=true  
$ CID=$(kcadm.sh create clients -r demorealm -s clientName=client1)  
$ kcadm.sh get clients/$CID/installation/providers/keycloak
```

- Windows:

```
c:\> kcadm config credentials --server http://localhost:8080/auth  
c:\> kcadm create realms -s realm=demorealm -s enabled=true  
c:\> kcadm create clients -r demorealm -s clientId=my_client  
c:\> set /p CID=<clientid.txt  
c:\> kcadm get clients/%CID%/installation/providers/keycloak
```

2. In a production environment, you must access KeyCloak with `https`: to avoid exposing tokens to network sniffers. If a server's certificate is not issued by one of the trusted certificate authorities (CAs) that are included in Java's default certificate truststore, prepare a `truststore.jks` file and instruct the Admin CLI to use it.

For example, on:

- Linux:

```
$ kcadm.sh config truststore --trustpass $PASSWORD ~/keystore.jks
```

- Windows:

```
c:\> kcadm config truststore --trustpass %PASSWORD% %HOMEPATH%\keystore.jks
```

## 19.3. Authenticating

When you log in with the Admin CLI, you specify a server endpoint URL and a realm, and then you specify a user name. Another option is to specify only a clientId, which results in using a special "service account". When you log in using a user name, you must use a password for the specified user. When you log in using a clientId, you only need the client secret, not the user password. You could also use `Signed`

`JWT` instead of the client secret.

Make sure the account used for the session has the proper permissions to invoke Admin REST API operations. For example, the `realm-admin` role of the `realm-management` client allows the user to administer the realm within which the user is defined.

There are two primary mechanisms for authentication. One mechanism uses `kcadm config credentials` to start an authenticated session.

```
$ kcadm.sh config credentials --server http://localhost:8080/
```

This approach maintains an authenticated session between the `kcadm` command invocations by saving the obtained access token and the associated refresh token. It may also maintain other secrets in a private configuration file. See [next chapter](#) for more information on the configuration file.

The second approach only authenticates each command invocation for the duration of that invocation. This approach increases the load on the server and the time spent with roundtrips obtaining tokens. The benefit of this approach is not needing to save any tokens between invocations, which means nothing is saved to disk. This mode is used when the `--no-config` argument is specified.

For example, when performing an operation, we specify all the information required for authentication.

```
$ kcadm.sh get realms --no-config --server http://localhost:8
```



Run the `kcadm.sh help` command for more information on using the Admin CLI.

Run the `kcadm.sh config credentials --help` command for more information about starting an authenticated session.

## 19.4. Working with alternative configurations

By default, the Admin CLI automatically maintains a configuration file called `kcadm.config` located under the user's home directory. In Linux-based systems, the full path name is `$HOME/.keycloak/kadm.config`. On Windows, the full path name is `%HOMEPATH%\keycloak\kadm.config`. You can use the `--config` option to point to a different file or location so you can maintain multiple authenticated sessions in parallel.

It is best to perform operations tied to a single configuration file from a single thread.

Make sure you do not make the configuration file visible to other users on the system. It contains access tokens and secrets that should be kept private. By default, the `~/keycloak` directory and its content are created automatically with proper access limits. If the directory already exists, its permissions are not updated.

If your unique circumstances require you to avoid storing secrets inside a configuration file, you can do so. It will be less convenient and you will have to make more token requests. To not store secrets, use the `--`

`no-config` option with all your commands and specify all the authentication information needed by the `config credentials` command with each `kcadm` invocation.

## 19.5. Basic operations and resource URIs

The Admin CLI allows you to generically perform CRUD operations against Admin REST API endpoints with additional commands that simplify performing certain tasks.

The main usage pattern is listed below, where the `create`, `get`, `update`, and `delete` commands are mapped to the HTTP verbs `POST`, `GET`, `PUT`, and `DELETE`, respectively.

```
$ kcadm.sh create ENDPOINT [ARGUMENTS]
$ kcadm.sh get ENDPOINT [ARGUMENTS]
$ kcadm.sh update ENDPOINT [ARGUMENTS]
$ kcadm.sh delete ENDPOINT [ARGUMENTS]
```

`ENDPOINT` is a target resource URI and can either be absolute (starting with `http:` or `https:`) or relative, used to compose an absolute URL of the following format:

```
SERVER_URI/admin/realm/REALM/ENDPOINT
```

For example, if you authenticate against the server <http://localhost:8080/auth> and realm is `master`, then using `users` as `ENDPOINT` results in the resource URL <http://localhost:8080/auth/admin/realm/master/users>.

If you set `ENDPOINT` to `clients`, the effective resource URI would

be <http://localhost:8080/auth/admin/realms/master/clients>.

There is a `realms` endpoint that is treated slightly differently because it is the container for realms. It resolves to:

```
 SERVER_URI/admin/realms
```

There is also a `serverinfo` endpoint, which is treated the same way because it is independent of realms.

When you authenticate as a user with realm-admin powers, you might need to perform commands on multiple realms. In that case, specify the `-r` option to tell explicitly which realm the command should be executed against. Instead of using `REALM` as specified via the `--realm` option of `kcadm.sh config credentials`, the `TARGET_REALM` is used.

```
 SERVER_URI/admin/realms/TARGET_REALM/ENDPOINT
```

For example,

```
$ kcadm.sh config credentials --server http://localhost:8080/  
$ kcadm.sh create users -s username=testuser -s enabled=true
```

In this example, you start a session authenticated as the `admin` user in the `master` realm. You then perform a POST call against the resource URL <http://localhost:8080/auth/admin/realms/demorealm/users>.

The `create` and `update` commands send a JSON body to the server

by default. You can use `-f FILENAME` to read a premade document from a file. When you can use `-f -` option, the message body is read from standard input. You can also specify individual attributes and their values as seen in the previous `create users` example. They are composed into a JSON body and sent to the server.

There are several ways to update a resource using the `update` command. You can first determine the current state of a resource and save it to a file, and then edit that file and send it to the server for updating.

For example:

```
$ kcadm.sh get realms/demorealm > demorealm.json  
$ vi demorealm.json  
$ kcadm.sh update realms/demorealm -f demorealm.json
```

This method updates the resource on the server with all the attributes in the sent JSON document.

Another option is to perform an on-the-fly update using the `-s`, `--set` options to set new values.

For example:

```
$ kcadm.sh update realms/demorealm -s enabled=false
```

That method only updates the `enabled` attribute to `false`.

By default, the `update` command first performs a `get` and then merges the new attribute values with existing values. This is the preferred

behavior. In some cases, the endpoint may support the `PUT` command but not the `GET` command. You can use the `-n` option to perform a "no-merge" update, which performs a `PUT` command without first running a `GET` command.

## 19.6. Realm operations

### Creating a new realm

Use the `create` command on the `realms` endpoint to create a new enabled realm, and set the attributes to `realm` and `enabled`.

```
$ kcadm.sh create realms -s realm=demorealm -s enabled=true
```

A realm is not enabled by default. By enabling it, you can use a realm immediately for authentication.

A description for a new object can also be in a JSON format.

```
$ kcadm.sh create realms -f demorealm.json
```

You can send a JSON document with realm attributes directly from a file or piped to a standard input.

For example, on:

- Linux:

```
$ kcadm.sh create realms -f - << EOF
{ "realm": "demorealm", "enabled": true }
EOF
```

- Windows:

```
c:\> echo { "realm": "demorealm", "enabled": true } | kcadm  
create realms -f -
```

## Listing existing realms

The following command returns a list of all realms.

```
$ kcadm.sh get realms
```

A list of realms is additionally filtered on the server to return only realms a user can see.

Returning the entire realm description often provides too much information. Most users are interested only in a subset of attributes, such as realm name and whether the realm is enabled. You can specify which attributes to return by using the `--fields` option.

```
$ kcadm.sh get realms --fields realm(enabled)
```

You can also display the result as comma separated values.

```
$ kcadm.sh get realms --fields realm --format csv --noquotes
```



## Getting a specific realm

You append a realm name to a collection URI to get an individual realm.

```
$ kcadm.sh get realms/master
```

## Updating a realm

1. Use the `-s` option to set new values for the attributes when you want to change only some of the realm's attributes.

For example:

```
$ kcadm.sh update realms/demorealm -s enabled=false
```

2. If you want to set all writable attributes with new values, run a `get` command, edit the current values in the JSON file, and resubmit.

For example:

```
$ kcadm.sh get realms/demorealm > demorealm.json
$ vi demorealm.json
$ kcadm.sh update realms/demorealm -f demorealm.json
```

## Deleting a realm

Run the following command to delete a realm.

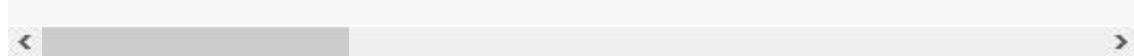
```
$ kcadm.sh delete realms/demorealm
```

## Turning on all login page options for the realm

Set the attributes controlling specific capabilities to `true`.

For example:

```
$ kcadm.sh update realms/demorealm -s registrationAllowed=true
```



## Listing the realm keys

Use the `get` operation on the `keys` endpoint of the target realm.

```
$ kcadm.sh get keys -r demorealm
```

## Generating new realm keys

1. Get the ID of the target realm before adding a new RSA-generated key pair.

For example:

```
$ kcadm.sh get realms/domorealm --fields id --format csv
```

2. Add a new key provider with a higher priority than the existing providers as revealed by `kcadm.sh get keys -r demorealm`.

For example, on:

- Linux:

```
$ kcadm.sh create components -r demorealm -s name=rsa-(
```

- Windows:

```
c:\> kcadm create components -r demorealm -s name=rsa-(
```

3. Set the `parentId` attribute to the value of the target realm's ID.

The newly added key should now become the active key as revealed by `kcadm.sh get keys -r demorealm`.

## Adding new realm keys from a Java Key Store file

1. Add a new key provider to add a new key pair already prepared as a JKS file on the server.

For example, on:

- Linux:

```
$ kcadm.sh create components -r demorealm -s name=java
```

- Windows:

```
c:\> kcadm create components -r demorealm -s name=java
```

2. Make sure to change the attribute values for `keystore`, `keystorePassword`, `keyPassword`, and `alias` to match your specific keystore.
3. Set the `parentId` attribute to the value of the target realm's ID.

## Making the key passive or disabling the key

1. Identify the key you want to make passive

```
$ kcadm.sh get keys -r demorealm
```

2. Use the key's `providerId` attribute to construct an endpoint URI, such as `components/PROVIDER_ID`.

### 3. Perform an `update`.

For example, on:

- Linux:

```
$ kcadm.sh update components/PROVIDER_ID -r demorealm .
```

- Windows:

```
c:\> kcadm update components/PROVIDER_ID -r demorealm .
```

You can update other key attributes.

4. Set a new `enabled` value to disable the key, for example, `config.enabled=["false"]`.
5. Set a new `priority` value to change the key's priority, for example, `config.priority=["110"]`.

## Deleting an old key

1. Make sure the key you are deleting has been passive and disabled to prevent any existing tokens held by applications and users from abruptly failing to work.
2. Identify the key you want to make passive.

```
$ kcadm.sh get keys -r demorealm
```

3. Use the `providerId` of that key to perform a delete.

```
$ kcadm.sh delete components/PROVIDER_ID -r demorealm
```

## Configuring event logging for a realm

Use the `update` command on the `events/config` endpoint.

The `eventsListeners` attribute contains a list of `EventListenerProviderFactory` IDs that specify all event listeners receiving events. Separately, there are attributes that control a built-in event storage, which allows querying past events via the Admin REST API. There is separate control over logging of service calls (`eventsEnabled`) and auditing events triggered during Admin Console or Admin REST API (`adminEventsEnabled`). You may want to set up expiry of old events so that your database does not fill up; `eventsExpiration` is set to time-to-live expressed in seconds.

Here is an example of setting up a built-in event listener that receives all the events and logs them through jboss-logging. (Using a logger called `org.keycloak.events`, error events are logged as `WARN`, and others are logged as `DEBUG`.)

For example, on:

- Linux:

```
$ kcadm.sh update events/config -r demorealm -s 'eventsListeners=["jboss-logging"]'
```

- Windows:

```
c:\> kcadm update events/config -r demorealm -s "eventsListeners=[\"jboss-logging\"]"
```

Here is an example of turning on storage of all available ERROR events—not including auditing events—for 2 days so they can be retrieved via Admin REST.

For example, on:

- Linux:

```
$ kcadm.sh update events/config -r demorealm -s eventsEnabled=true -s 'enabledEventTypes=[ "LOGIN_ERROR", "REGISTER_ERROR", "LOGOUT_ERROR", "CODE_TO_TOKEN_ERROR", "CLIENT_LOGIN_ERROR", "FEDERATED_IDENTITY_LINK_ERROR", "REMOVE_FEDERATED_IDENTITY_ERROR", "UPDATE_EMAIL_ERROR", "UPDATE_PROFILE_ERROR", "UPDATE_PASSWORD_ERROR", "UPDATE_TOTP_ERROR", "VERIFY_EMAIL_ERROR", "REMOVE_TOTP_ERROR", "SEND_VERIFY_EMAIL_ERROR", "SEND_RESET_PASSWORD_ERROR", "SEND_IDENTITY_PROVIDER_LINK_ERROR", "RESET_PASSWORD_ERROR", "IDENTITY_PROVIDER_FIRST_LOGIN_ERROR", "IDENTITY_PROVIDER_POST_LOGIN_ERROR", "CUSTOM_REQUIRED_ACTION_ERROR", "EXECUTE_ACTIONS_ERROR", "CLIENT_REGISTER_ERROR", "CLIENT_UPDATE_ERROR", "CLIENT_DELETE_ERROR"]' -s eventsExpiration=172800
```

- Windows:

```
c:\> kcadm update events/config -r demorealm -s eventsEnabled=true -s "enabledEventTypes=[ \"LOGIN_ERROR\", \"REGISTER_ERROR\", \"LOGOUT_ERROR\", \"CODE_TO_TOKEN_ERROR\", \"CLIENT_LOGIN_ERROR\", \"FEDERATED_IDENTITY_LINK_ERROR\", \"REMOVE_FEDERATED_IDENTITY_ERROR\", \"UPDATE_EMAIL_ERROR\", \"UPDATE_PROFILE_ERROR\", \"UPDATE_PASSWORD_ERROR\", \"UPDATE_TOTP_ERROR\", \"VERIFY_EMAIL_ERROR\", \"REMOVE_TOTP_ERROR\", \"SEND_VERIFY_EMAIL_ERROR\", \"SEND_RESET_PASSWORD_ERROR\", \"SEND_IDENTITY_PROVIDER_LINK_ERROR\", \"RESET_PASSWORD_ERROR\", \"IDENTITY_PROVIDER_FIRST_LOGIN_ERROR\", \"IDENTITY_PROVIDER_POST_LOGIN_ERROR\", \"CUSTOM_REQUIRED_ACTION_ERROR\", \"EXECUTE_ACTIONS_ERROR\", \"CLIENT_REGISTER_ERROR\", \"CLIENT_UPDATE_ERROR\", \"CLIENT_DELETE_ERROR\"]" -s eventsExpiration=172800
```

Here is an example of how to reset stored event types to **all available event types**; setting to empty list is the same as enumerating all.

```
$ kcadm.sh update events/config -r demorealm -s enabledEventT
```

Here is an example of how to enable storage of auditing events.

```
$ kcadm.sh update events/config -r demorealm -s adminEventsEn
```

Here is an example of how to get the last 100 events; they are ordered from newest to oldest.

```
$ kcadm.sh get events --offset 0 --limit 100
```

Here is an example of how to delete all saved events.

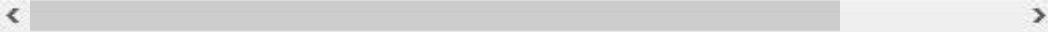
```
$ kcadm delete events
```

## Flushing the caches

1. Use the `create` command and one of the following endpoints:  
`clear-realm-cache`, `clear-user-cache`, or `clear-keys-cache`.
2. Set `realm` to the same value as the target realm.

For example:

```
$ kcadm.sh create clear-realm-cache -r demorealm -s realm  
$ kcadm.sh create clear-user-cache -r demorealm -s realm=  
$ kcadm.sh create clear-keys-cache -r demorealm -s realm=
```



## Importing a realm from exported .json file

1. Use the `create` command on the `partialImport` endpoint.
2. Set `ifResourceExists` to one of `FAIL`, `SKIP`, `OVERWRITE`.
3. Use `-f` to submit the exported realm `.json` file

For example:

```
$ kcadm.sh create partialImport -r demorealm2 -s ifResour
```



If realm does not yet exist, you first have to create it.

For example:

```
$ kcadm.sh create realms -s realm=demorealm2 -s enabled=t
```



## 19.7. Role operations

### Creating a realm role

Use the `roles` endpoint to create a realm role.

```
$ kcadm.sh create roles -r demorealm -s name=user -s 'descrip
```

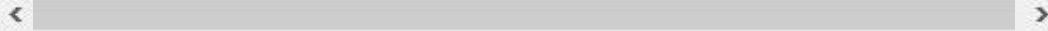


### Creating a client role

1. Identify the client first and then use the `get` command to list avail-

able clients when creating a client role.

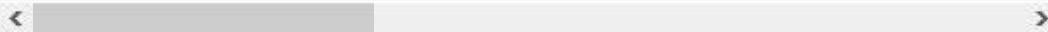
```
$ kcadm.sh get clients -r demorealm --fields id,clientId
```



2. Create a new role by using the `clientId` attribute to construct an endpoint URI, such as `clients/ID/roles`.

For example:

```
$ kcadm.sh create clients/a95b6af3-0bdc-4878-ae2e-6d61a4e
```



## Listing realm roles

Use the `get` command on the `roles` endpoint to list existing realm roles.

```
$ kcadm.sh get roles -r demorealm
```

You can also use the `get-roles` command.

```
$ kcadm.sh get-roles -r demorealm
```

## Listing client roles

There is a dedicated `get-roles` command to simplify listing realm and client roles. It is an extension of the `get` command and behaves the same with additional semantics for listing roles.

Use the `get-roles` command, passing it either the `clientId` attribute (via the `--cclientid` option) or `id` (via the `--cid` option) to iden-

tify the client to list client roles.

For example:

```
$ kcadm.sh get-roles -r demorealm --cclientid realm-managemen
```

### Getting a specific realm role

Use the `get` command and the role `name` to construct an endpoint URI for a specific realm role: `roles/ROLE_NAME`, where `user` is the name of the existing role.

For example:

```
$ kcadm.sh get roles/user -r demorealm
```

You can also use the special `get-roles` command, passing it a role name (via the `--rolename` option) or ID (via the `--roleid` option).

For example:

```
$ kcadm.sh get-roles -r demorealm --rolename user
```

### Getting a specific client role

Use a dedicated `get-roles` command, passing it either the clientId attribute (via the `--cclientid` option) or ID (via the `--cid` option) to identify the client, and passing it either the role name (via the `--rolename` option) or ID (via the `--roleid`) to identify a specific client role.

For example:

```
$ kcadm.sh get-roles -r demorealm --clientid realm-management
```

### Updating a realm role

Use the `update` command with the same endpoint URI that you used to get a specific realm role.

For example:

```
$ kcadm.sh update roles/user -r demorealm -s 'description=Role
```

### Updating a client role

Use the `update` command with the same endpoint URI that you used to get a specific client role.

For example:

```
$ kcadm.sh update clients/a95b6af3-0bdc-4878-ae2e-6d61a4eca9a
```

### Deleting a realm role

Use the `delete` command with the same endpoint URI that you used to get a specific realm role.

For example:

```
$ kcadm.sh delete roles/user -r demorealm
```

## Deleting a client role

Use the `delete` command with the same endpoint URI that you used to get a specific client role.

For example:

```
$ kcadm.sh delete clients/a95b6af3-0bdc-4878-ae2e-6d61a4eca9a
```

## Listing assigned, available, and effective realm roles for a composite role

Use a dedicated `get-roles` command to list assigned, available, and effective realm roles for a composite role.

1. To list **assigned** realm roles for the composite role, you can specify the target composite role by either name (via the `--rname` option) or ID (via the `--rid` option).

For example:

```
$ kcadm.sh get-roles -r demorealm --rname testrole
```

2. Use the additional `--effective` option to list **effective** realm roles.

For example:

```
$ kcadm.sh get-roles -r demorealm --rname testrole --effe
```

3. Use the `--available` option to list realm roles that can still be added to the composite role.

For example:

```
$ kcadm.sh get-roles -r demorealm --rname testrole --avai
```

### Listing assigned, available, and effective client roles for a composite role

Use a dedicated `get-roles` command to list assigned, available, and effective client roles for a composite role.

1. To list **assigned** client roles for the composite role, you can specify the target composite role by either name (via the `--rname` option) or ID (via the `--rid` option) and client by either the `clientId` attribute (via the `--cclientid` option) or ID (via the `--cid` option).

For example:

```
$ kcadm.sh get-roles -r demorealm --rname testrole --ccli
```

2. Use the additional `--effective` option to list **effective** realm roles.

For example:

```
$ kcadm.sh get-roles -r demorealm --rname testrole --ccli
```

3. Use the `--available` option to list realm roles that can still be added to the target composite role.

For example:

```
$ kcadm.sh get-roles -r demorealm --rname testrole --ccli
```

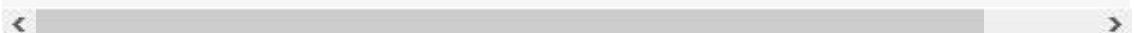


## Adding realm roles to a composite role

There is a dedicated `add-roles` command that can be used for adding realm roles and client roles.

The following example adds the `user` role to the composite role `testrole`.

```
$ kcadm.sh add-roles --rname testrole --rolename user -r demo
```

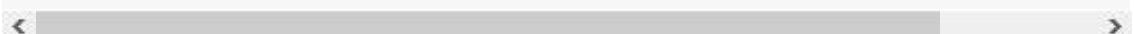


## Removing realm roles from a composite role

There is a dedicated `remove-roles` command that can be used to remove realm roles and client roles.

The following example removes the `user` role from the target composite role `testrole`.

```
$ kcadm.sh remove-roles --rname testrole --rolename user -r d
```



## Adding client roles to a realm role

Use a dedicated `add-roles` command that can be used for adding realm roles and client roles.

The following example adds the roles defined on the client `realm-management - create-client` role and the `view-users` role to the `testrole` composite role.

```
$ kcadm.sh add-roles -r demorealm --rname testrole --cclienti
```

## Adding client roles to a client role

1. Determine the ID of the composite client role by using the `get-roles` command.

For example:

```
$ kcadm.sh get-roles -r demorealm --cclientid test-client
```

2. Assume that there is a client with a clientId attribute of `test-client`, a client role called `support`, and another client role called `operations`, which becomes a composite role, that has an ID of "fc400897-ef6a-4e8c-872b-1581b7fa8a71".
3. Use the following example to add another role to the composite role.

```
$ kcadm.sh add-roles -r demorealm --cclientid test-client
```

4. List the roles of a composite role by using the `get-roles --all` command.

For example:

```
$ kcadm.sh get-roles --rid fc400897-ef6a-4e8c-872b-1581b7
```

## Removing client roles from a composite role

Use a dedicated `remove-roles` command to remove client roles from

a composite role.

Use the following example to remove two roles defined on the client `realm management - create-client` role and the `view-users` role from the `testrole` composite role.

```
$ kcadm.sh remove-roles -r demorealm --rname testrole --cclie
```

### Adding client roles to a group

Use a dedicated `add-roles` command that can be used for adding realm roles and client roles.

The following example adds the roles defined on the client `realm management - create-client` role and the `view-users` role to the `Group` group (via the `--gname` option). The group can alternatively be specified by ID (via the `--gid` option).

See [Group operations](#) for more operations that can be performed to groups.

```
$ kcadm.sh add-roles -r demorealm --gname Group --cclientid r
```

### Removing client roles from a group

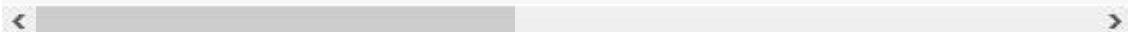
Use a dedicated `remove-roles` command to remove client roles from a group.

Use the following example to remove two roles defined on the client `realm management - create-client` role and the `view-users`

role from the `Group` group.

See [Group operations](#) for more operations that can be performed to groups.

```
$ kcadm.sh remove-roles -r demorealm --gname Group --cclienti
```

A horizontal scroll bar with a grey track and a white slider, indicating the command continues on the next line.

## 19.8. Client operations

### Creating a client

1. Run the `create` command on a `clients` endpoint to create a new client.

For example:

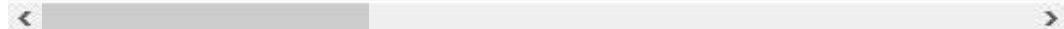
```
$ kcadm.sh create clients -r demorealm -s clientId=myapp
```

A horizontal scroll bar with a grey track and a white slider, indicating the command continues on the previous line.

2. Specify a secret if you want to set a secret for adapters to authenticate.

For example:

```
$ kcadm.sh create clients -r demorealm -s clientId=myapp
```

A horizontal scroll bar with a grey track and a white slider, indicating the command continues on the previous line.

### Listing clients

Use the `get` command on the `clients` endpoint to list clients.

For example:

```
$ kcadm.sh get clients -r demorealm --fields id,clientId
```

This example filters the output to list only the `id` and `clientId` attributes.

### Getting a specific client

Use a client's ID to construct an endpoint URI that targets a specific client, such as `clients/ID`.

For example:

```
$ kcadm.sh get clients/c7b8547f-e748-4333-95d0-410b76b3f4a3 -
```



### Getting the current secret for a specific client

Use a client's ID to construct an endpoint URI, such as `clients/ID/client-secret`.

For example:

```
$ kcadm.sh get clients/$CID/client-secret
```

### Getting an adapter configuration file (`keycloak.json`) for a specific client

Use a client's ID to construct an endpoint URI that targets a specific client, such as `clients/ID/installation/providers/keycloak-oidc-keycloak-json`.

For example:

```
$ kcadm.sh get clients/c7b8547f-e748-4333-95d0-410b76b3f4a3/i
```

## Getting a WildFly subsystem adapter configuration for a specific client

Use a client's ID to construct an endpoint URI that targets a specific client, such as `clients/ID/installation/providers/keycloak-oidc-jboss-subsystem`.

For example:

```
$ kcadm.sh get clients/c7b8547f-e748-4333-95d0-410b76b3f4a3/i
```

## Getting a Docker-v2 example configuration for a specific client

Use a client's ID to construct an endpoint URI that targets a specific client, such as `clients/ID/installation/providers/docker-v2-compose-yaml`.

Note that response will be in `.zip` format.

For example:

```
$ kcadm.sh get http://localhost:8080/auth/admin/realms/demore
```

## Updating a client

Use the `update` command with the same endpoint URI that you used to get a specific client.

For example, on:

- Linux:

```
$ kcadm.sh update clients/c7b8547f-e748-4333-95d0-  
410b76b3f4a3 -r demorealm -s enabled=false -s publicCli-  
ent=true -s 'redirectUris=["http://local-  
host:8080/myapp/*"]' -s baseUrl=http://localhost:8080/myapp  
-s adminUrl=http://localhost:8080/myapp
```

- Windows:

```
c:\> kcadm update clients/c7b8547f-e748-4333-95d0-  
410b76b3f4a3 -r demorealm -s enabled=false -s publicCli-  
ent=true -s "redirectUris=[\"http://local-  
host:8080/myapp/*\"]" -s baseUrl=http://local-  
host:8080/myapp -s adminUrl=http://localhost:8080/myapp
```

## Deleting a client

Use the `delete` command with the same endpoint URI that you used to get a specific client.

For example:

```
$ kcadm.sh delete clients/c7b8547f-e748-4333-95d0-410b76b3f4a
```

## Adding or removing roles for client's service account

Service account for the client is just a special kind of user account with username `service-account-CLIENT_ID`. You can perform user operations on this account as if it was a regular user.

## 19.9. User operations

### Creating a user

Run the `create` command on the `users` endpoint to create a new user.

For example:

```
$ kcadm.sh create users -r demorealm -s username=testuser -s
```

### Listing users

Use the `users` endpoint to list users. The target user will have to change the password the next time they log in.

For example:

```
$ kcadm.sh get users -r demorealm --offset 0 --limit 1000
```

You can filter users by `username`, `firstName`, `lastName`, or `email`.

For example:

```
$ kcadm.sh get users -r demorealm -q email=google.com  
$ kcadm.sh get users -r demorealm -q username=testuser
```

Filtering does not use exact matching. For example, the above example would match the value of the `username` attribute against the `*testuser*` pattern.

You can also filter across multiple attributes by specifying multiple `-q` options, which return only users that match the condition for all the attributes.

## Getting a specific user

Use a user's ID to compose an endpoint URI, such as

`users/USER_ID`.

For example:

```
$ kcadm.sh get users/0ba7a3fd-6fd8-48cd-a60b-2e8fd82d56e2 -r
```



## Updating a user

Use the `update` command with the same endpoint URI that you used to get a specific user.

For example, on:

- Linux:

```
$ kcadm.sh update users/0ba7a3fd-6fd8-48cd-a60b-2e8fd82d56e2 -r demorealm -s 'requiredActions=["VERIFY_E-MAIL", "UPDATE_PROFILE", "CONFIGURE_TOTP", "UPDATE_PASSWORD"]'
```

- Windows:

```
c:\> kcadm update users/0ba7a3fd-6fd8-48cd-a60b-2e8fd82d56e2 -r demorealm -s "requiredActions=[\"VERIFY_E-MAIL\", \"UPDATE_PROFILE\", \"CONFIGURE_TOTP\", \"UPDATE_PASSWORD\"]"
```

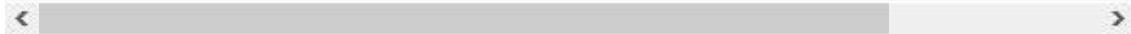
```
WORD\"]"
```

## Deleting a user

Use the `delete` command with the same endpoint URI that you used to get a specific user.

For example:

```
$ kcadm.sh delete users/0ba7a3fd-6fd8-48cd-a60b-2e8fd82d56e2
```

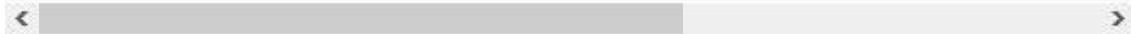


## Resetting a user's password

Use the dedicated `set-password` command to reset a user's password.

For example:

```
$ kcadm.sh set-password -r demorealm --username testuser --ne
```



That command sets a temporary password for the user. The target user will have to change the password the next time they log in.

You can use `--userid` if you want to specify the user by using the `id` attribute.

You can achieve the same result using the `update` command on an endpoint constructed from the one you used to get a specific user, such as `users/USER_ID/reset-password`.

For example:

```
$ kcadm.sh update users/0ba7a3fd-6fd8-48cd-a60b-2e8fd82d56e2/
```

The last parameter ( `-n` ) ensures that only the `PUT` command is performed without a prior `GET` command. It is necessary in this instance because the `reset-password` endpoint does not support `GET`.

### **Listing assigned, available, and effective realm roles for a user**

You can use a dedicated `get-roles` command to list assigned, available, and effective realm roles for a user.

1. Specify the target user by either user name or ID to list **assigned** realm roles for the user.

For example:

```
$ kcadm.sh get-roles -r demorealm --username testuser
```

1. Use the additional `--effective` option to list **effective** realm roles.

For example:

```
$ kcadm.sh get-roles -r demorealm --username testuser --
```

2. Use the `--available` option to list realm roles that can still be added to the user.

For example:

```
$ kcadm.sh get-roles -r demorealm --username testuser --
```



## Listing assigned, available, and effective client roles for a user

Use a dedicated `get-roles` command to list assigned, available, and effective client roles for a user.

1. Specify the target user by either a user name (via the `--username` option) or an ID (via the `--uid` option) and client by either a `clientId` attribute (via the `--clientid` option) or an ID (via the `--cid` option) to list **assigned** client roles for the user.

For example:

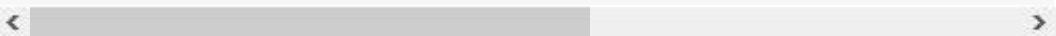
```
$ kcadm.sh get-roles -r demorealm --username testuser --
```



2. Use the additional `--effective` option to list **effective** realm roles.

For example:

```
$ kcadm.sh get-roles -r demorealm --username testuser --
```



3. Use the `--available` option to list realm roles that can still be added to the user.

For example:

```
$ kcadm.sh get-roles -r demorealm --username testuser --
```



## Adding realm roles to a user

Use a dedicated `add-roles` command to add realm roles to a user.

Use the following example to add the `user` role to user `testuser`.

```
$ kcadm.sh add-roles --username testuser --rolename user -r
```

## Removing realm roles from a user

Use a dedicated `remove-roles` command to remove realm roles from a user.

Use the following example to remove the `user` role from the user `testuser`.

```
$ kcadm.sh remove-roles --username testuser --rolename user
```

## Adding client roles to a user

Use a dedicated `add-roles` command to add client roles to a user.

Use the following example to add two roles defined on the client `realm management` - `create-client` role and the `view-users` role to the user `testuser`.

```
$ kcadm.sh add-roles -r demorealm --username testuser --ccli
```

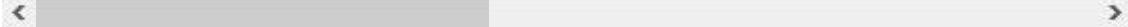
## Removing client roles from a user

Use a dedicated `remove-roles` command to remove client roles from

a user.

Use the following example to remove two roles defined on the realm management client.

```
$ kcadm.sh remove-roles -r demorealm --username testuser --c
```

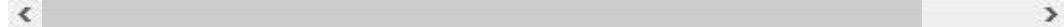
A horizontal scroll bar with a grey track and a white slider, indicating the command continues on the next line.

## Listing a user's sessions

1. Identify the user's ID, and then use it to compose an endpoint URI, such as `users/ID/sessions`.
2. Use the `get` command to retrieve a list of the user's sessions.

For example:

```
$kcadm get users/6da5ab89-3397-4205-afaa-e201ff638f9e/ses
```

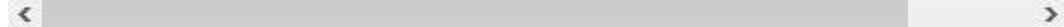
A horizontal scroll bar with a grey track and a white slider, indicating the command continues on the next line.

## Logging out a user from a specific session

1. Determine the session's ID as described above.
2. Use the session's ID to compose an endpoint URI, such as `sessions/ID`.
3. Use the `delete` command to invalidate the session.

For example:

```
$ kcadm.sh delete sessions/d0eaa7cc-8c5d-489d-811a-69d3c4
```

A horizontal scroll bar with a grey track and a white slider, indicating the command continues on the next line.

## Logging out a user from all sessions

You need a user's ID to construct an endpoint URI, such as `users/ID/logout`.

Use the `create` command to perform `POST` on that endpoint URI.

For example:

```
$ kcadm.sh create users/6da5ab89-3397-4205-afaa-e201ff638f9e/
```

## 19.10. Group operations

### Creating a group

Use the `create` command on the `groups` endpoint to create a new group.

For example:

```
$ kcadm.sh create groups -r demorealm -s name=Group
```

### Listing groups

Use the `get` command on the `groups` endpoint to list groups.

For example:

```
$ kcadm.sh get groups -r demorealm
```

### Getting a specific group

Use the group's ID to construct an endpoint URI, such as `groups/GROUP_ID`.

For example:

```
$ kcadm.sh get groups/51204821-0580-46db-8f2d-27106c6b5ded -r
```

## Updating a group

Use the `update` command with the same endpoint URI that you used to get a specific group.

For example:

```
$ kcadm.sh update groups/51204821-0580-46db-8f2d-27106c6b5ded
```

## Deleting a group

Use the `delete` command with the same endpoint URI that you used to get a specific group.

For example:

```
$ kcadm.sh delete groups/51204821-0580-46db-8f2d-27106c6b5ded
```

## Creating a subgroup

Find the ID of the parent group by listing groups, and then use that ID to construct an endpoint URI, such as `groups/GROUP_ID/children`.

For example:

```
$ kcadm.sh create groups/51204821-0580-46db-8f2d-27106c6b5ded
```



## Moving a group under another group

1. Find the ID of an existing parent group and of an existing child group.
2. Use the parent group's ID to construct an endpoint URI, such as `groups/PARENT_GROUP_ID/children`.
3. Run the `create` command on this endpoint and pass the child group's ID as a JSON body.

For example:

```
$ kcadm.sh create groups/51204821-0580-46db-8f2d-27106c6b5ded
```

## Get groups for a specific user

Use a user's ID to determine a user's membership in groups to compose an endpoint URI, such as `users/USER_ID/groups`.

For example:

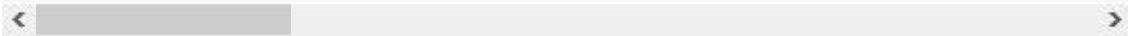
```
$ kcadm.sh get users/b544f379-5fc4-49e5-8a8d-5cfb71f46f53/gro
```

## Adding a user to a group

Use the `update` command with an endpoint URI composed from user's ID and a group's ID, such as `users/USER_ID/groups/GROUP_ID`, to add a user to a group.

For example:

```
$ kcadm.sh update users/b544f379-5fc4-49e5-8a8d-5cfb71f46f53/
```



## Removing a user from a group

Use the `delete` command on the same endpoint URI as used for adding a user to a group, such as `users/USER_ID/groups/GROUP_ID`, to remove a user from a group.

For example:

```
$ kcadm.sh delete users/b544f379-5fc4-49e5-8a8d-5cfb71f46f53/
```



## Listing assigned, available, and effective realm roles for a group

Use a dedicated `get-roles` command to list assigned, available, and effective realm roles for a group.

1. Specify the target group by name (via the `--gname` option), path (via the [command] `--gpath` option), or ID (via the `--gid` option) to list **assigned** realm roles for the group.

For example:

```
$ kcadm.sh get-roles -r demorealm --gname Group
```

2. Use the additional `--effective` option to list **effective** realm roles.

For example:

```
$ kcadm.sh get-roles -r demorealm --gname Group --effecti
```



- 
3. Use the `--available` option to list realm roles that can still be added to the group.

For example:

```
$ kcadm.sh get-roles -r demorealm --gname Group --availab
```



### Listing assigned, available, and effective client roles for a group

Use a dedicated `get-roles` command to list assigned, available, and effective client roles for a group.

1. Specify the target group by either name (via the `--gname` option) or ID (via the `--gid` option), and client by either the `clientId` attribute (via the [command] `--cclientid` option) or ID (via the `--id` option) to list **assigned** client roles for the user.

For example:

```
$ kcadm.sh get-roles -r demorealm --gname Group --cclient
```



2. Use the additional `--effective` option to list **effective** realm roles.

For example:

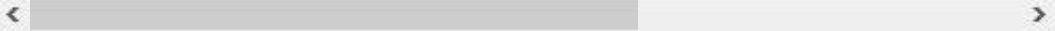
```
$ kcadm.sh get-roles -r demorealm --gname Group --cclient
```



3. Use the `--available` option to list realm roles that can still be added to the group.

For example:

```
$ kcadm.sh get-roles -r demorealm --gname Group --cclient
```



## 19.11. Identity provider operations

### Listing available identity providers

Use the `serverinfo` endpoint to list available identity providers.

For example:

```
$ kcadm.sh get serverinfo -r demorealm --fields 'identityProv
```



The `serverinfo` endpoint is handled similarly to the `realms` endpoint in that it is not resolved relative to a target realm because it exists outside any specific realm.

### Listing configured identity providers

Use the `identity-provider/instances` endpoint.

For example:

```
$ kcadm.sh get identity-provider/instances -r demorealm --fie
```



### Getting a specific configured identity provider

Use the `alias` attribute of the identity provider to construct an end-

point URI, such as `identity-provider/instances/ALIAS`, to get a specific identity provider.

For example:

```
$ kcadm.sh get identity-provider/instances/facebook -r demoreal
```

### Removing a specific configured identity provider

Use the `delete` command with the same endpoint URI that you used to get a specific configured identity provider to remove a specific configured identity provider.

For example:

```
$ kcadm.sh delete identity-provider/instances/facebook -r demoreal
```

### Configuring a Keycloak OpenID Connect identity provider

1. Use `keycloak-oidc` as the `providerId` when creating a new identity provider instance.
2. Provide the `config` attributes: `authorizationUrl`, `tokenUrl`, `clientId`, and `clientSecret`.

For example:

```
$ kcadm.sh create identity-provider/instances -r demoreal
```

### Configuring an OpenID Connect identity provider

Configure the generic OpenID Connect provider the same way you configure the Keycloak OpenID Connect provider, except that you set the `providerId` attribute value to `oidc`.

## Configuring a SAML 2 identity provider

1. Use `saml` as the `providerId`.
2. Provide the `config` attributes: `singleSignOnServiceUrl`, `nameIDPolicyFormat`, and `signatureAlgorithm`.

For example:

```
$ kcadm.sh create identity-provider/instances -r demorealm -s
```

## Configuring a Facebook identity provider

1. Use `facebook` as the `providerId`.
2. Provide the `config` attributes: `clientId` and `clientSecret`.  
You can find these attributes in the Facebook Developers application configuration page for your application.

For example:

```
$ kcadm.sh create identity-provider/instances -r demorealm
```

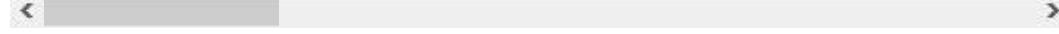
## Configuring a Google identity provider

1. Use `google` as the `providerId`.
2. Provide the `config` attributes: `clientId` and `clientSecret`.  
You can find these attributes in the Google Developers application

configuration page for your application.

For example:

```
$ kcadm.sh create identity-provider/instances -r demoreal
```

A horizontal scroll bar with a grey track and a white slider, indicating the command continues on the next line.

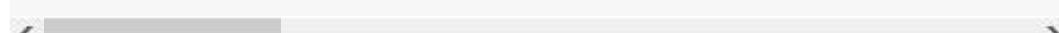
## Configuring a Twitter identity provider

1. Use `twitter` as the `providerId`.
2. Provide the `config` attributes `clientId` and `clientSecret`.

You can find these attributes in the Twitter Application Management application configuration page for your application.

For example:

```
$ kcadm.sh create identity-provider/instances -r demoreal
```

A horizontal scroll bar with a grey track and a white slider, indicating the command continues on the next line.

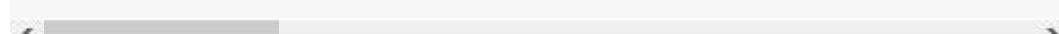
## Configuring a GitHub identity provider

1. Use `github` as the `providerId`.
2. Provide the `config` attributes `clientId` and `clientSecret`.

You can find these attributes in the GitHub Developer Application Settings page for your application.

For example:

```
$ kcadm.sh create identity-provider/instances -r demoreal
```

A horizontal scroll bar with a grey track and a white slider, indicating the command continues on the next line.

## Configuring a LinkedIn identity provider

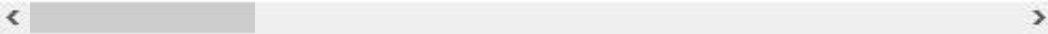
1. Use `linkedin` as the `providerId`.

2. Provide the `config` attributes `clientId` and `clientSecret`.

You can find these attributes in the LinkedIn Developer Console application page for your application.

For example:

```
$ kcadm.sh create identity-provider/instances -r demoreal...
```



## Configuring a Microsoft Live identity provider

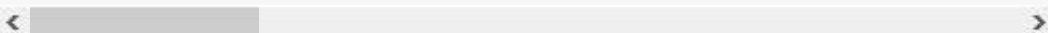
1. Use `microsoft` as the `providerId`.

2. Provide the `config` attributes `clientId` and `clientSecret`.

You can find these attributes in the Microsoft Application Registration Portal page for your application.

For example:

```
$ kcadm.sh create identity-provider/instances -r demoreal...
```



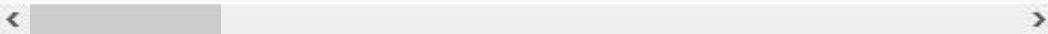
## Configuring a Stack Overflow identity provider

1. Use `stackoverflow` command as the `providerId`.

2. Provide the `config` attributes `clientId`, `clientSecret`, and `key`. You can find these attributes in the Stack Apps OAuth page for your application.

For example:

```
$ kcadm.sh create identity-provider/instances -r demoreal...
```



## 19.12. Storage provider operations

## Configuring a Kerberos storage provider

1. Use the `create` command against the `components` endpoint.
2. Specify realm id as a value of the `parentId` attribute.
3. Specify `kerberos` as the value of the `providerId` attribute, and `org.keycloak.storage.UserStorageProvider` as the value of the `providerType` attribute.
4. For example:

```
$ kcadm.sh create components -r demorealm -s parentId=dem
```

## Configuring an LDAP user storage provider

1. Use the `create` command against the `components` endpoint.
2. Specify `ldap` as a value of the `providerId` attribute, and `org.keycloak.storage.UserStorageProvider` as the value of the `providerType` attribute.
3. Provide the realm ID as the value of the `parentId` attribute.
4. Use the following example to create a Kerberos-integrated LDAP provider.

```
$ kcadm.sh create components -r demorealm -s name=kerbero
```

## Removing a user storage provider instance

1. Use the storage provider instance's `id` attribute to compose an endpoint URI, such as `components/ID`.
2. Run the `delete` command against this endpoint.

For example:

```
$ kcadm.sh delete components/3d9c572b-8f33-483f-98a6-8bb4
```

A horizontal bar with a left arrow, a central gray area, and a right arrow, used for copying text.

### Triggering synchronization of all users for a specific user storage provider

1. Use the storage provider's `id` attribute to compose an endpoint URI, such as `user-storage/ID_OF_USER_STORAGE_INSTANCE/sync`.
2. Add the `action=triggerFullSync` query parameter and run the `create` command.

For example:

```
$ kcadm.sh create user-storage/b7c63d02-b62a-4fc1-977c-94
```

A horizontal bar with a left arrow, a central gray area, and a right arrow, used for copying text.

### Triggering synchronization of changed users for a specific user storage provider

1. Use the storage provider's `id` attribute to compose an endpoint URI, such as `user-storage/ID_OF_USER_STORAGE_INSTANCE/sync`.
2. Add the `action=triggerChangedUsersSync` query parameter and run the `create` command.

For example:

```
$ kcadm.sh create user-storage/b7c63d02-b62a-4fc1-977c-94
```

A horizontal bar with a left arrow, a central gray area, and a right arrow, used for copying text.

### Test LDAP user storage connectivity

1. Run the `get` command on the `testLDAPConnection` endpoint.
2. Provide query parameters `bindCredential`, `bindDn`, `connectionUrl`, and `useTruststoreSpi`, and then set the `action` query parameter to `testConnection`.

For example:

```
$ kcadm.sh get testLDAPConnection -q action=testConnectio
```

### Test LDAP user storage authentication

1. Run the `get` command on the `testLDAPConnection` endpoint.
2. Provide the query parameters `bindCredential`, `bindDn`, `connectionUrl`, and `useTruststoreSpi`, and then set the `action` query parameter to `testAuthentication`.

For example:

```
$ kcadm.sh get testLDAPConnection -q action=testAuthentic
```

## 19.13. Adding mappers

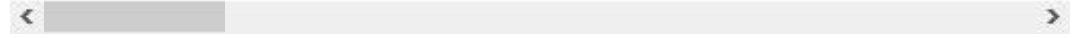
### Adding a hardcoded role LDAP mapper

1. Run the `create` command on the `components` endpoint.
2. Set the `providerType` attribute to `org.keycloak.storage.ldap.mappers.LDAPStorageMapper`.
3. Set the `parentId` attribute to the ID of the LDAP provider instance.
4. Set the `providerId` attribute to `hardcoded-ldap-role-map-`

per . Make sure to provide a value of role configuration parameter.

For example:

```
$ kcadm.sh create components -r demorealm -s name=hardcod
```

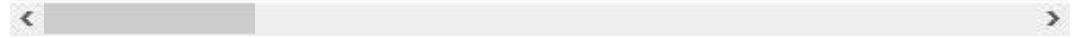


## Adding an MS Active Directory mapper

1. Run the create command on the components endpoint.
2. Set the providerType attribute to org.keycloak.storage.ldap.mappers.LDAPStorageMapper .
3. Set the parentId attribute to the ID of the LDAP provider instance.
4. Set the providerId attribute to msad-user-account-control-mapper .

For example:

```
$ kcadm.sh create components -r demorealm -s name=msad-us
```



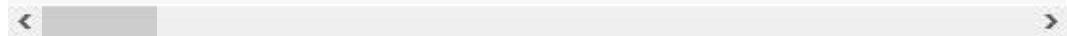
## Adding a user attribute LDAP mapper

1. Run the create command on the components endpoint.
2. Set the providerType attribute to org.keycloak.storage.ldap.mappers.LDAPStorageMapper .
3. Set the parentId attribute to the ID of the LDAP provider instance.
4. Set the providerId attribute to user-attribute-ldap-map-

per .

For example:

```
$ kcadm.sh create components -r demorealm -s name=user-at
```

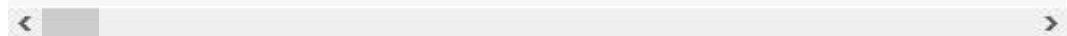
A horizontal scroll bar with a grey track and a white slider, indicating the command continues.

## Adding a group LDAP mapper

1. Run the `create` command on the `components` endpoint.
2. Set the `providerType` attribute to `org.keycloak.storage.ldap.mappers.LDAPStorageMapper`.
3. Set the `parentId` attribute to the ID of the LDAP provider instance.
4. Set the `providerId` attribute to `group-ldap-mapper`.

For example:

```
$ kcadm.sh create components -r demorealm -s name=group-1
```

A horizontal scroll bar with a grey track and a white slider, indicating the command continues.

## Adding a full name LDAP mapper

1. Run the `create` command on the `components` endpoint.
2. Set the `providerType` attribute to `org.keycloak.storage.ldap.mappers.LDAPStorageMapper`.
3. Set the `parentId` attribute to the ID of the LDAP provider instance.
4. Set the `providerId` attribute to `full-name-ldap-mapper`.

For example:

```
$ kcadm.sh create components -r demorealm -s name=full-na
```



## 19.14. Authentication operations

### Setting a password policy

1. Set the realm's `passwordPolicy` attribute to an enumeration expression that includes the specific policy provider ID and optional configuration.
2. Use the following example to set a password policy to default values. The default values include:
  - 27,500 hashing iterations
  - at least one special character
  - at least one uppercase character
  - at least one digit character
  - not be equal to a user's `username`
  - be at least eight characters long

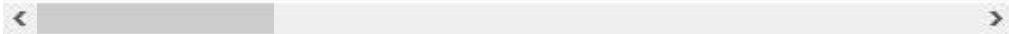
```
$ kcadm.sh update realms/demorealm -s 'passwordPolicy='
```



3. If you want to use values different from defaults, pass the configuration in brackets.
4. Use the following example to set a password policy to:
  - 25,000 hash iterations
  - at least two special characters

- at least two uppercase characters
- at least two lowercase characters
- at least two digits
- be at least nine characters long
- not be equal to a user's `username`
- not repeat for at least four changes back

```
$ kcadm.sh update realms/demorealm -s 'passwordPolicy='
```



## Getting the current password policy

Get the current realm configuration and filter everything but the `passwordPolicy` attribute.

Use the following example to display `passwordPolicy` for `demorealm`.

```
$ kcadm.sh get realms/demorealm --fields passwordPolicy
```

## Listing authentication flows

Run the `get` command on the `authentication/flows` endpoint.

For example:

```
$ kcadm.sh get authentication/flows -r demorealm
```

## Getting a specific authentication flow

Run the `get` command on the `authentication/flows/FL0W_ID` endpoint.

For example:

```
$ kcadm.sh get authentication/flows/febfd772-e1a1-42fb-b8ae-0
```



## Listing executions for a flow

Run the `get` command on the `authentication/flows/FL0W_ALI-AS/executions` endpoint.

For example:

```
$ kcadm.sh get authentication/flows/Copy%20of%20browser/execu
```

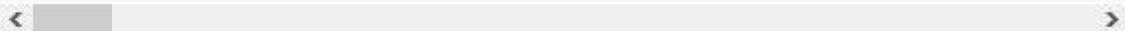


## Adding configuration to an execution

1. Get execution for a flow, and take note of its ID
2. Run the `create` command on the `authentication/executi-  
ons/{executionId}/config` endpoint.

For example:

```
$ kcadm create "authentication/executions/a3147129-c402-4760-
```



## Getting configuration for an execution

1. Get execution for a flow, and get its `authenticationConfig` attribute, containing the config ID.

2. Run the `get` command on the `authentication/config/ID` endpoint.

For example:

```
$ kcadm get "authentication/config/dd91611a-d25c-421a-87e2-22
```

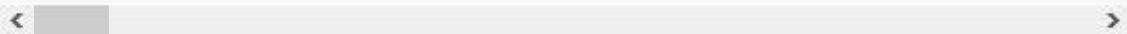


## Updating configuration for an execution

1. Get execution for a flow, and get its `authenticationConfig` attribute, containing the config ID.
2. Run the `update` command on the `authentication/config/ID` endpoint.

For example:

```
$ kcadm update "authentication/config/dd91611a-d25c-421a-87e2
```



## Deleting configuration for an execution

1. Get execution for a flow, and get its `authenticationConfig` attribute, containing the config ID.
2. Run the `delete` command on the `authentication/config/ID` endpoint.

For example:

```
$ kcadm delete "authentication/config/dd91611a-d25c-421a-87e2
```



Last updated 2019-06-13 12:48:56 MESZ

# Table of Contents

*{developerguide!}*

- 1. Preface
- 2. Admin REST API
  - 2.1. Example using CURL
- 3. Themes
  - 3.1. Theme Types
  - 3.2. Configure Theme
  - 3.3. Default Themes
  - 3.4. Creating a Theme
    - 3.4.1. Theme Properties
    - 3.4.2. Stylesheets
    - 3.4.3. Scripts
    - 3.4.4. Images
    - 3.4.5. Messages
    - 3.4.6. Internationalization
    - 3.4.7. HTML Templates
    - 3.4.8. Emails
  - 3.5. Deploying Themes
  - 3.6. Theme Selector
  - 3.7. Theme Resources
  - 3.8. Locale Selector
- 4. Custom User Attributes
  - 4.1. Registration Page
  - 4.2. Account Management Console
- 5. Identity Brokering APIs
  - 5.1. Retrieving External IDP Tokens

## 5.2. Client Initiated Account Linking

### 5.2.1. Refreshing External Tokens

## 6. Service Provider Interfaces (SPI)

### 6.1. Implementing an SPI

#### 6.1.1. Show info from your SPI implementation in admin console

### 6.2. Registering provider implementations

#### 6.2.1. Using the Keycloak Deployer

#### 6.2.2. Register a provider using Modules

#### 6.2.3. Disabling a provider

### 6.3. Leveraging Java EE

### 6.4. Available SPIs

## 7. User Storage SPI

### 7.1. Provider Interfaces

### 7.2. Provider Capability Interfaces

### 7.3. Model Interfaces

#### 7.3.1. Storage Ids

### 7.4. Packaging and Deployment

### 7.5. Simple Read-Only, Lookup Example

#### 7.5.1. Provider Class

#### 7.5.2. Provider Factory Implementation

#### 7.5.3. Packaging and Deployment

#### 7.5.4. Enabling the Provider in the Administration Console

### 7.6. Configuration Techniques

#### 7.6.1. Configuration Example

#### 7.6.2. Configuring the Provider in the Administration Console

### 7.7. Add/Remove User and Query Capability interfaces

#### 7.7.1. Implementing UserRegistrationProvider

- 7.7.2. Implementing UserQueryProvider
  - 7.8. Augmenting External Storage
    - 7.8.1. Augmentation Example
  - 7.9. Import Implementation Strategy
    - 7.9.1. ImportedUserValidation Interface
    - 7.9.2. ImportSynchronization Interface
  - 7.10. User Caches
    - 7.10.1. Managing the user cache
    - 7.10.2. OnUserCache Callback Interface
    - 7.10.3. Cache Policies
  - 7.11. Leveraging Java EE
  - 7.12. REST Management API
  - 7.13. Migrating from an Earlier User Federation SPI
    - 7.13.1. Import vs. Non-Import
    - 7.13.2. UserFederationProvider vs. UserStorageProvider
    - 7.13.3. UserFederationProviderFactory vs. UserStorageProviderFactory
    - 7.13.4. Upgrading to a New Model
-

# {developerguide!}

# 1. Preface

In some of the example listings, what is meant to be displayed on one line does not fit inside the available page width. These lines have been broken up. A '\' at the end of a line means that a break has been introduced to fit in the page, with the following lines indented. So:

```
Let's pretend to have an extremely \
long line that \
does not fit
This one is short
```

Is really:

```
Let's pretend to have an extremely long line that does not
fit
This one is short
```

---

## 2. Admin REST API

Keycloak comes with a fully functional Admin REST API with all features provided by the Admin Console.

To invoke the API you need to obtain an access token with the appropriate permissions. The required permissions are described in `{admin-guide_link}[{adminguide!}]`.

A token can be obtained by enabling authenticating to your application with Keycloak; see the `{adapterguide_link}[{adapterguide!}]`. You can also use direct access grant to obtain an access token.

For complete documentation see `{apidocs_link}[{apidocs_name}]`.

### 2.1. Example using CURL

Obtain access token for user in the realm `master` with username `admin` and password `password`:

```
curl \
  -d "client_id=admin-cli" \
  -d "username=admin" \
  -d "password=password" \
  -d "grant_type=password" \
  "http://localhost:8080/auth/realms/master/protocol/openid-connect/token"
```

By default this token expires in 1 minute

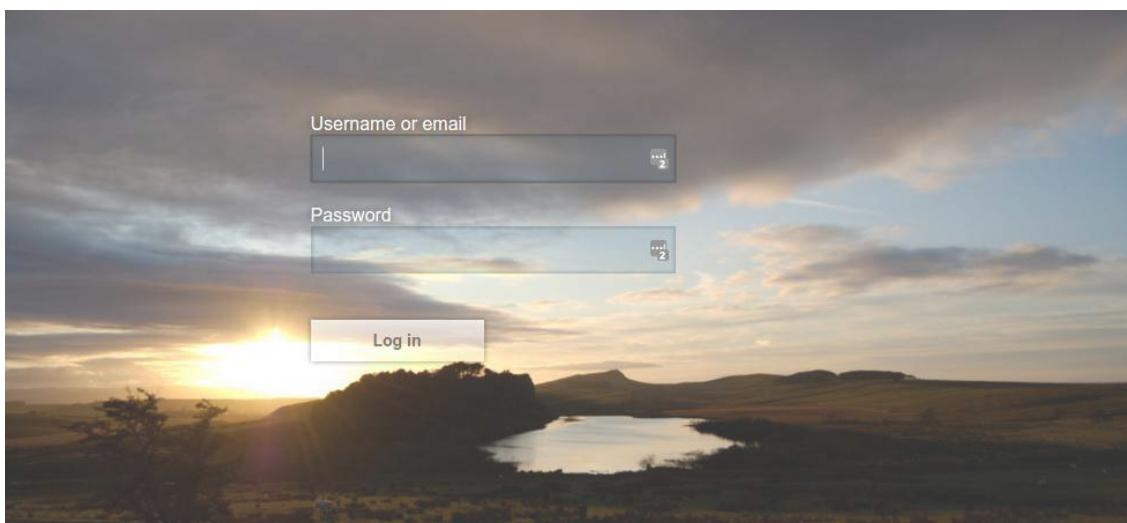
The result will be a JSON document. To invoke the API you need to extract the value of the `access_token` property. You can then invoke the API by including the value in the `Authorization` header of requests to the API.

The following example shows how to get the details of the master realm:

```
curl \  
  -H "Authorization: bearer eyJhbGciOiJSUz..." \  
  "http://localhost:8080/auth/admin/realms/master"
```

## 3. Themes

Keycloak provides theme support for web pages and emails. This allows customizing the look and feel of end-user facing pages so they can be integrated with your applications.



*Login page with sunrise example theme*

### 3.1. Theme Types

A theme can provide one or more types to customize different aspects of Keycloak. The types available are:

- Account - Account management
- Admin - Admin console
- Email - Emails
- Login - Login forms
- Welcome - Welcome page

## 3.2. Configure Theme

All theme types, except welcome, are configured through the `Admin Console`. To change the theme used for a realm open the `Admin Console`, select your realm from the drop-down box in the top left corner. Under `Realm Settings` click `Themes`.

To set the theme for the `master` admin console you need to set the admin console theme for the `master` realm. To see the changes to the admin console refresh the page.

To change the welcome theme you need to edit `standalone.xml`, `standalone-ha.xml`, or `domain.xml`. For more information on where these files reside see the [{installguide!}](#).

Add `welcomeTheme` to the theme element, for example:

```
<theme>
  ...
  <welcomeTheme>custom-theme</welcomeTheme>
  ...
</theme>
```

If the server is running you need to restart the server for the changes to the welcome theme to take effect.

## 3.3. Default Themes

Keycloak comes bundled with default themes in the server's root `themes` directory. To simplify upgrading you should not edit the bundled

themes directly. Instead create your own theme that extends one of the bundled themes.

## 3.4. Creating a Theme

A theme consists of:

- HTML templates ([Freemarker Templates](#))
- Images
- Message bundles
- Stylesheets
- Scripts
- Theme properties

Unless you plan to replace every single page you should extend another theme. Most likely you will want to extend the Keycloak theme, but you could also consider extending the base theme if you are significantly changing the look and feel of the pages. The base theme primarily consists of HTML templates and message bundles, while the Keycloak theme primarily contains images and stylesheets.

When extending a theme you can override individual resources (templates, stylesheets, etc.). If you decide to override HTML templates bear in mind that you may need to update your custom template when upgrading to a new release.

While creating a theme it's a good idea to disable caching as this makes it possible to edit theme resources directly from the `themes` directory

without restarting Keycloak. To do this edit `standalone.xml`. For `theme` set `staticMaxAge` to `-1` and both `cacheTemplates` and `cacheThemes` to `false`:

```
<theme>
  <staticMaxAge>-1</staticMaxAge>
  <cacheThemes>false</cacheThemes>
  <cacheTemplates>false</cacheTemplates>
  ...
</theme>
```

Remember to re-enable caching in production as it will significantly impact performance.

To create a new theme start by creating a new directory in the `themes` directory. The name of the directory becomes the name of the theme. For example to create a theme called `mytheme` create the directory `themes/mytheme`.

Inside the theme directory create a directory for each of the types your theme is going to provide. For example to add the login type to the `mytheme` theme create the directory `themes/mytheme/login`.

For each type create a file `theme.properties` which allows setting some configuration for the theme. For example to configure the theme `themes/mytheme/login` that we just created to extend the base theme and import some common resources create the file `themes/mytheme/login/theme.properties` with following contents:

```
parent=base
import=common/keycloak
```

You have now created a theme with support for the login type. To check that it works open the admin console. Select your realm and click on Themes . For Login Theme select mytheme and click Save . Then open the login page for the realm.

You can do this either by login through your application or by opening the Account Management console ( /realms/{realm name}/account ).

To see the effect of changing the parent theme, set parent=keycloak in theme.properties and refresh the login page.

### 3.4.1. Theme Properties

Theme properties are set in the file <THEME TYPE>/theme.properties in the theme directory.

- parent - Parent theme to extend
- import - Import resources from another theme
- styles - Space-separated list of styles to include
- locales - Comma-separated list of supported locales

There are a list of properties that can be used to change the css class used for certain element types. For a list of these properties look at the theme.properties file in the corresponding type of the keycloak theme ( themes/keycloak/<THEME TYPE>/theme.properties ).

You can also add your own custom properties and use them from custom templates.

### 3.4.2. Stylesheets

A theme can have one or more stylesheets. To add a stylesheet create a file in the `<THEME TYPE>/resources/css` directory of your theme. Then add it to the `styles` property in `theme.properties`.

For example to add `styles.css` to the `mytheme` create `themes/mytheme/login/resources/css/styles.css` with the following content:

```
.login-pf body {  
    background: DimGrey none;  
}
```

Then edit `themes/mytheme/login/theme.properties` and add:

```
styles=css/styles.css
```

To see the changes open the login page for your realm. You will notice that the only styles being applied are those from your custom stylesheet. To include the styles from the parent theme you need to load the styles from that theme as well. Do this by editing `themes/mytheme/login/theme.properties` and changing `styles` to:

```
styles=node_modules/patternfly/dist/css/patternfly.css  
node_modules/patternfly/dist/css/patternfly-additions.css  
lib/zocial/zocial.css css/login.css css/styles.css
```

To override styles from the parent stylesheets it's important that your stylesheet is listed last.

### 3.4.3. Scripts

A theme can have one or more scripts, to add a script create a file in the `<THEME TYPE>/resources/js` directory of your theme. Then add it to the `scripts` property in `theme.properties`.

For example to add `script.js` to the `mytheme` create `themes/my-theme/login/resources/js/script.js` with the following content:

```
alert('Hello');
```

Then edit `themes/mytheme/login/theme.properties` and add:

```
scripts=js/script.js
```

### 3.4.4. Images

To make images available to the theme add them to the `<THEME TYPE>/resources/img` directory of your theme. These can be used from within stylesheets or directly in HTML templates.

For example to add an image to the `mytheme` copy an image to `themes/mytheme/login/resources/img/image.jpg`.

You can then use this image from within a custom stylesheet with:

```
body {  
    background-image: url('../img/image.jpg');  
    background-size: cover;  
}
```

Or to use directly in HTML templates add the following to a custom HTML template:

```

```

### 3.4.5. Messages

Text in the templates is loaded from message bundles. A theme that extends another theme will inherit all messages from the parent's message bundle and you can override individual messages by adding `<THEME TYPE>/messages/messages_en.properties` to your theme.

For example to replace `Username` on the login form with `Your Username` for the `mytheme` create the file `themes/mytheme/login/messages/messages_en.properties` with the following content:

```
usernameOrEmail=Your Username
```

Within a message values like `{0}` and `{1}` are replaced with arguments when the message is used. For example `{0}` in `Log in to {0}` is replaced with the name of the realm.

### 3.4.6. Internationalization

KeyCloak supports internationalization. To enable internationalization for a realm see `{adminguide_link}[{adminguide!}]`. This section describes how you can add your own language.

To add a new language create the file `<THEME TYPE>/messages/messages_<LOCALE>.properties` in the directory of your theme. Then add it to the `locales` property in `<THEME TYPE>/theme.properties`:

ties. For a language to be available to users the realms `login`, `account` and `email` theme has to support the language, so you need to add your language for those theme types.

For example, to add Norwegian translations to the `mytheme` theme create the file `themes/mytheme/login/messages/messages_no.properties` with the following content:

```
usernameOrEmail=Brukernavn  
password=Passord
```

All messages you don't provide a translation for will use the default English translation.

Then edit `themes/mytheme/login/theme.properties` and add:

```
locales=en, no
```

You also need to do the same for the `account` and `email` theme types. To do this create `themes/mytheme/account/messages/messages_no.properties` and `themes/mytheme/email/messages/messages_no.properties`. Leaving these files empty will result in the English messages being used. Then copy `themes/mytheme/login/theme.properties` to `themes/mytheme/account/theme.properties` and `themes/mytheme/email/theme.properties`.

Finally you need to add a translation for the language selector. This is done by adding a message to the English translation. To do this add the

following to `themes/mytheme/account/messages/messages_en.properties` and `themes/mytheme/login/messages/messages_en.properties`:

```
locale_no=Norsk
```

By default message properties files should be encoded using ISO-8859-1. It's also possible to specify the encoding using a special header. For example to use UTF-8 encoding:

```
# encoding: UTF-8
usernameOrEmail=....
```

See [Locale Selector](#) on details on how the current locale is selected.

### 3.4.7. HTML Templates

KeyCloak uses [Freemarker Templates](#) in order to generate HTML. You can override individual templates in your own theme by creating `<THEME TYPE>/<TEMPLATE>.ftl`. For a list of templates used see `themes/base/<THEME TYPE>`.

When creating a custom template it is a good idea to copy the template from the base theme to your own theme, then applying the modifications you need. Bear in mind when upgrading to a new version of KeyCloak you may need to update your custom templates to apply changes to the original template if applicable.

For example to create a custom login form for the `mytheme` theme copy `themes/base/login/login.ftl` to `themes/mythe-`

`me/login` and open it in an editor. After the first line (`<#import ...>`) add `<h1>HELLO WORLD!</h1>` like so:

```
<#import "template.ftl" as layout>
<h1>HELLO WORLD!</h1>
...
...
```

Check out the [FreeMarker Manual](#) for more details on how to edit templates.

### 3.4.8. Emails

To edit the subject and contents for emails, for example password recovery email, add a message bundle to the `email` type of your theme. There are three messages for each email. One for the subject, one for the plain text body and one for the html body.

To see all emails available take a look at `themes/base/email/messages/messages_en.properties`.

For example to change the password recovery email for the `mytheme` theme create `themes/mytheme/email/messages/messages_en.properties` with the following content:

```
passwordResetSubject=My password recovery
passwordResetBody=Reset password link: {0}
passwordResetBodyHtml=<a href="{0}">Reset password</a>
```

## 3.5. Deploying Themes

Themes can be deployed to Keycloak by copying the theme directory to `themes` or it can be deployed as an archive. During development

you can copy the theme to the `themes` directory, but in production you may want to consider using an `archive`. An `archive` makes it simpler to have a versioned copy of the theme, especially when you have multiple instances of Keycloak for example with clustering.

To deploy a theme as an archive you need to create a JAR archive with the theme resources. You also need to add a file `META-INF/keycloak-themes.json` to the archive that lists the available themes in the archive as well as what types each theme provides.

For example for the `mytheme` theme create `mytheme.jar` with the contents:

- `META-INF/keycloak-themes.json`
- `theme/mytheme/login/theme.properties`
- `theme/mytheme/login/login.ftl`
- `theme/mytheme/login/resources/css/styles.css`
- `theme/mytheme/login/resources/img/image.png`
- `theme/mytheme/login/messages/messages_en.properties`
- `theme/mytheme/email/messages/messages_en.properties`

The contents of `META-INF/keycloak-themes.json` in this case would be:

```
{  
  "themes": [{  
    "name" : "mytheme",  
    "types": [ "login", "email" ]  
  }]
```

```
}
```

A single archive can contain multiple themes and each theme can support one or more types.

To deploy the archive to Keycloak simply drop it into the `standalone/deployments/` directory of Keycloak and it will be automatically loaded.

## 3.6. Theme Selector

By default the theme configured for the realm is used, with the exception of clients being able to override the login theme. This behavior can be changed through the Theme Selector SPI.

This could be used to select different themes for desktop and mobile devices by looking at the user agent header, for example.

To create a custom theme selector you need to implement `ThemeSelectorProviderFactory` and `ThemeSelectorProvider`.

Follow the steps in [Service Provider Interfaces](#) for more details on how to create and deploy a custom provider.

## 3.7. Theme Resources

When implementing custom providers in Keycloak there may often be a need to add additional templates and resources.

The easiest way to load additional theme resources is to create a JAR with templates in `theme-resources/templates` and resources in

`theme-resources/resources` and drop it into the `standalone/deployments/` directory of Keycloak.

If you want a more flexible way to load templates and resources that can be achieved through the `ThemeResourceSPI`. By implementing `ThemeResourceProviderFactory` and `ThemeResourceProvider` you can decide exactly how to load templates and resources.

Follow the steps in [Service Provider Interfaces](#) for more details on how to create and deploy a custom provider.

### 3.8. Locale Selector

By default, the locale is selected using the `DefaultLocaleSelectorProvider` which implements the `LocaleSelectorProvider` interface. English is the default language when internationalization is disabled. With internationalization enabled, the locale is resolved in the following priority:

1. `kc_locale` query parameter
2. `KEYCLOAK_LOCALE` cookie value
3. User's preferred locale if a user instance is available
4. `ui_locales` query parameter
5. `Accept-Language` request header
6. Realm's default language

This behaviour can be changed through the `LocaleSelectorSPI` by implementing the `LocaleSelectorProvider` and `LocaleSelect-`

`torProviderFactory`.

The `LocaleSelectorProvider` interface has a single method, `re-solveLocale`, which must return a locale given a `RealmModel` and a nullable `UserModel`. The actual request is available from the `KeycloakSession#getContext` method.

Custom implementations can extend the `DefaultLocaleSelectorProvider` in order to reuse parts of the default behaviour. For example to ignore the `Accept-Language` request header, a custom implementation could extend the default provider, override it's `getAcceptLanguageHeaderLocale`, and return a null value. As a result the locale selection will fall back on the realms's default language.

Follow the steps in [Service Provider Interfaces](#) for more details on how to create and deploy a custom provider.

## 4. Custom User Attributes

You can add custom user attributes to the registration page and account management console with a custom theme. This chapter describes how to add attributes to a custom theme, but you should refer to the [Themes](#) chapter on how to create a custom theme.

### 4.1. Registration Page

To be able to enter custom attributes in the registration page copy the template `themes/base/login/register.ftl` to the login type of your custom theme. Then open the copy in an editor.

As an example to add a mobile number to the registration page add the following snippet to the form:

```
<div class="form-group">
    <div class="${properties.kcLabelWrapperClass!}">
        <label for="user.attributes.mobile" class="${properties.kcLabelClass!}">Mobile number</label>
    </div>

    <div class="${properties.kcInputWrapperClass!}">
        <input type="text" class="${properties.kcInputClass!}" id="user.attributes.mobile" name="user.
        attributes.mobile"/>
    </div>
</div>
```

Ensure the name of the input html element starts with `user.attributes`. In the example above, the attribute will be stored by Keycloak with the name `mobile`.

To see the changes make sure your realm is using your custom theme for the login theme and open the registration page.

## 4.2. Account Management Console

To be able to manage custom attributes in the user profile page in the account management console copy the template `themes/base/account/account.ftl` to the account type of your custom theme. Then open the copy in an editor.

As an example to add a mobile number to the account page add the following snippet to the form:

```
<div class="form-group">
    <div class="col-sm-2 col-md-2">
        <label for="user.attributes.mobile" class="control-label">Mobile number</label>
        </div>

        <div class="col-sm-10 col-md-10">
            <input type="text" class="form-control" id="user.attributes.mobile" name="user.attributes.mobile" value="${(account.attributes.mobile! '')}"/>
        </div>
    </div>
```

Ensure the name of the input html element starts with `user.attributes..`

To see the changes make sure your realm is using your custom theme for the account theme and open the user profile page in the account management console.

---

## 5. Identity Brokering APIs

Keycloak can delegate authentication to a parent IDP for login. A typical example of this is the case where you want users to be able to login through a social provider like Facebook or Google. Keycloak also allows you to link existing accounts to a brokered IDP. This section talks about some APIs that your applications can use as it pertains to identity brokering.

### 5.1. Retrieving External IDP Tokens

Keycloak allows you to store tokens and responses from the authentication process with the external IDP. For that, you can use the `Store Token` configuration option on the IDP's settings page.

Application code can retrieve these tokens and responses to pull in extra user information, or to securely invoke requests on the external IDP. For example, an application might want to use the Google token to invoke on other Google services and REST APIs. To retrieve a token for a particular identity provider you need to send a request as follows:

```
GET /auth/realms/{realm}/broker/{provider_alias}/token
HTTP/1.1
Host: localhost:8080
Authorization: Bearer <KEYCLOAK ACCESS TOKEN>
```

An application must have authenticated with Keycloak and have received an access token. This access token will need to have the `broker` client-level role `read-token` set. This means that the user must have a

role mapping for this role and the client application must have that role within its scope. In this case, given that you are accessing a protected service in KeyCloak, you need to send the access token issued by KeyCloak during the user authentication. In the broker configuration page you can automatically assign this role to newly imported users by turning on the `Stored Tokens Readable` switch.

These external tokens can be re-established by either logging in again through the provider, or using the client initiated account linking API.

## 5.2. Client Initiated Account Linking

Some applications want to integrate with social providers like Facebook, but do not want to provide an option to login via these social providers. KeyCloak offers a browser-based API that applications can use to link an existing user account to a specific external IDP. This is called client-initiated account linking. Account linking can only be initiated by OIDC applications.

The way it works is that the application forwards the user's browser to a URL on the KeyCloak server requesting that it wants to link the user's account to a specific external provider (i.e. Facebook). The server initiates a login with the external provider. The browser logs in at the external provider and is redirected back to the server. The server establishes the link and redirects back to the application with a confirmation.

There are some preconditions that must be met by the client application before it can initiate this protocol:

- The desired identity provider must be configured and enabled for

the user's realm in the admin console.

- The user account must already be logged in as an existing user via the OIDC protocol
- The user must have an `account.manage-account` or `account.manage-account-links` role mapping.
- The application must be granted the scope for those roles within its access token
- The application must have access to its access token as it needs information within it to generate the redirect URL.

To initiate the login, the application must fabricate a URL and redirect the user's browser to this URL. The URL looks like this:

```
/{{auth-server-root}}/auth/realms/{{realm}}/broker/{{provider}}/link?client_id={{id}}&redirect_uri={{uri}}&nonce={{nonce}}&hash={{hash}}
```

Here's a description of each path and query param:

### **provider**

This is the provider alias of the external IDP that you defined in the `Identity Provider` section of the admin console.

### **client\_id**

This is the OIDC client id of your application. When you registered the application as a client in the admin console, you had to specify this client id.

## **redirect\_uri**

This is the application callback URL you want to redirect to after the account link is established. It must be a valid client redirect URI pattern. In other words, it must match one of the valid URL patterns you defined when you registered the client in the admin console.

## **nonce**

This is a random string that your application must generate

## **hash**

This is a Base64 URL encoded hash. This hash is generated by Base64 URL encoding a SHA\_256 hash of `nonce + token.getSessionState() + token.getIssuedFor() + provider`. The token variable are obtained from the OIDC access token. Basically you are hashing the random nonce, the user session id, the client id, and the identity provider alias you want to access.

Here's an example of Java Servlet code that generates the URL to establish the account link.

```
KeycloakSecurityContext session = (KeycloakSecurityContext) httpServletRequest.getAttribute(KeycloakSecurityContext.class.getName());
AccessToken token = session.getToken();
String clientId = token.getIssuedFor();
String nonce = UUID.randomUUID().toString();
MessageDigest md = null;
try {
    md = MessageDigest.getInstance("SHA-256");
} catch (NoSuchAlgorithmException e) {
    throw new RuntimeException(e);
}
String input = nonce + token.getSessionState() + clien-
```

```

tId + provider;
byte[] check = md.digest(input.getBytes(StandardCharsets.UTF_8));
String hash = Base64Url.encode(check);
request.getSession().setAttribute("hash", hash);
String redirectUri = ...;
String accountLinkUrl = KeycloakUriBuilder.fromUri(auth-
ServerRootUrl)
    .path("/auth/realms/{realm}/bro-
ker/{provider}/link")
    .queryParam("nonce", nonce)
    .queryParam("hash", hash)
    .queryParam("client_id", clientId)
    .queryParam("redirect_uri", redirectU-
ri).build(realm, provider).toString();

```

Why is this hash included? We do this so that the auth server is guaranteed to know that the client application initiated the request and no other rogue app just randomly asked for a user account to be linked to a specific provider. The auth server will first check to see if the user is logged in by checking the SSO cookie set at login. It will then try to regenerate the hash based on the current login and match it up to the hash sent by the application.

After the account has been linked, the auth server will redirect back to the `redirect_uri`. If there is a problem servicing the link request, the auth server may or may not redirect back to the `redirect_uri`. The browser may just end up at an error page instead of being redirected back to the application. If there is an error condition and the auth server deems it safe enough to redirect back to the client app, an additional `error` query parameter will be appended to the `redirect_uri`.

While this API guarantees that the application initiated the request, it does not completely prevent CSRF at-

tacks for this operation. The application is still responsible for guarding against CSRF attacks target at itself.

### 5.2.1. Refreshing External Tokens

If you are using the external token generated by logging into the provider (i.e. a Facebook or GitHub token), you can refresh this token by re-initiating the account linking API.

---

# 6. Service Provider Interfaces (SPI)

Keycloak is designed to cover most use-cases without requiring custom code, but we also want it to be customizable. To achieve this Keycloak has a number of Service Provider Interfaces (SPI) for which you can implement your own providers.

## 6.1. Implementing an SPI

To implement an SPI you need to implement its ProviderFactory and Provider interfaces. You also need to create a service configuration file.

For example, to implement the Theme Selector SPI you need to implement ThemeSelectorProviderFactory and ThemeSelectorProvider and also provide the file `META-INF/services/org.keycloak.theme.ThemeSelectorProviderFactory`.

Example ThemeSelectorProviderFactory:

```
package org.acme.provider;

import ...

public class MyThemeSelectorProviderFactory implements ThemeSelectorProviderFactory {

    @Override
    public ThemeSelectorProvider create(KeycloakSession session) {
        return new MyThemeSelectorProvider(session);
    }

    @Override
```

```

public void init(Config.Scope config) {
}

@Override
public void postInit(KeycloakSessionFactory factory) {
}

@Override
public void close() {
}

@Override
public String getId() {
    return "myThemeSelector";
}
}

```

Keycloak creates a single instance of provider factories which makes it possible to store state for multiple requests. Provider instances are created by calling `create` on the factory for each request so these should be lightweight object.

Example ThemeSelectorProvider:

```

package org.acme.provider;

import ...

public class MyThemeSelectorProvider implements ThemeSelectorProvider {

    public MyThemeSelectorProvider(KeycloakSession session)
{
}

@Override

```

```
public String getThemeName(Theme.Type type) {
    return "my-theme";
}

@Override
public void close() {
}
}
```

Example service configuration file ( `META-INF/services/org.keycloak.theme.ThemeSelectorProviderFactory` ):

```
org.acme.provider.MyThemeSelectorProviderFactory
```

You can configure your provider through `standalone.xml`, `standalone-ha.xml`, or `domain.xml`. See the [{installguide!}](#) for more details on where to find these files.

For example by adding the following to `standalone.xml`:

```
<spi name="themeSelector">
    <provider name="myThemeSelector" enabled="true">
        <properties>
            <property name="theme" value="my-theme"/>
        </properties>
    </provider>
</spi>
```

Then you can retrieve the config in the `ProviderFactory` init method:

```
public void init(Config.Scope config) {
    String themeName = config.get("theme");
}
```

Your provider can also lookup other providers if needed. For example:

```
public class MyThemeSelectorProvider implements EventListenerProvider {

    private KeycloakSession session;

    public MyThemeSelectorProvider(KeycloakSession session)
    {
        this.session = session;
    }

    @Override
    public String getThemeName(Theme.Type type) {
        return session.getContext().getRealm().getLoginTheme();
    }
}
```

### 6.1.1. Show info from your SPI implementation in admin console

Sometimes it is useful to show additional info about your Provider to a KeyCloak administrator. You can show provider build time information (eg. version of custom provider currently installed), current configuration of the provider (eg. url of remote system your provider talks to) or some operational info (average time of response from remote system your provider talks to). KeyCloak admin console provides Server Info page to show this kind of information.

To show info from your provider it is enough to implement `org.keycloak.provider.ServerInfoAwareProviderFactory` interface in your `ProviderFactory`.

Example implementation for `MyThemeSelectorProviderFactory`

from previous example:

```
package org.acme.provider;

import ...

public class MyThemeSelectorProviderFactory implements ThemeSelectorProviderFactory, ServerInfoAwareProviderFactory {
    ...

    @Override
    public Map<String, String> getOperationalInfo() {
        Map<String, String> ret = new LinkedHashMap<>();
        ret.put("theme-name", "my-theme");
        return ret;
    }
}
```

## 6.2. Registering provider implementations

There are two ways to register provider implementations. In most cases the simplest way is to use the Keycloak deployer approach as this handles a number of dependencies automatically for you. It also supports hot deployment as well as re-deployment.

The alternative approach is to deploy as a module.

If you are creating a custom SPI you will need to deploy it as a module, otherwise we recommend using the Keycloak deployer approach.

### 6.2.1. Using the Keycloak Deployer

If you copy your provider jar to the Keycloak `standalone/deployments/` directory, your provider will automatically be deployed. Hot deployment works too. Additionally, your provider jar works similarly

to other components deployed in a WildFly environment in that they can use facilities like the `jboss-deployment-structure.xml` file. This file allows you to set up dependencies on other components and load third-party jars and modules.

Provider jars can also be contained within other deployable units like EARs and WARs. Deploying with a EAR actually makes it really easy to use third party jars as you can just put these libraries in the EAR's `lib/` directory.

### 6.2.2. Register a provider using Modules

To register a provider using Modules first create a module. To do this you can either use the `jboss-cli` script or manually create a folder inside `KEYCLOAK_HOME/modules` and add your jar and a `module.xml`. For example to add the event listener sysout example provider using the `jboss-cli` script execute:

```
KEYCLOAK_HOME/bin/jboss-cli.sh --command="module add --name=org.acme.provider --resources=target/provider.jar --dependencies=org.keycloak.keycloak-core,org.keycloak.keycloak-server-spi"
```

Or to manually create it start by creating the folder `KEYCLOAK_HOME/modules/org/acme/provider/main`. Then copy `provider.jar` to this folder and create `module.xml` with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.3" name="org.acme.provider">
    <resources>
```

```
<resource-root path="provider.jar"/>
</resources>
<dependencies>
    <module name="org.keycloak.keycloak-core"/>
    <module name="org.keycloak.keycloak-server-spi"/>
</dependencies>
</module>
```

Once you've created the module you need to register this module with KeyCloak. This is done by editing the keycloak-server subsystem section of `standalone.xml`, `standalone-ha.xml`, or `domain.xml`, and adding it to the providers:

```
<subsystem xmlns="urn:jboss:domain:keycloak-server:1.1">
    <web-context>auth</web-context>
    <providers>
        <provider>module:org.keycloak.examples.event-sy-
sout</provider>
    </providers>
    ...

```

### 6.2.3. Disabling a provider

You can disable a provider by setting the enabled attribute for the provider to false in `standalone.xml`, `standalone-ha.xml`, or `domain.xml`. For example to disable the Infinispan user cache provider add:

```
<spi name="userCache">
    <provider name="infinispan" enabled="false"/>
</spi>
```

## 6.3. Leveraging Java EE

The service providers can be packaged within any Java EE component

so long as you set up the `META-INF/services` file correctly to point to your providers. For example, if your provider needs to use third party libraries, you can package up your provider within an ear and store these third party libraries in the ear's `lib/` directory. Also note that provider jars can make use of the `jboss-deployment-structure.xml` file that EJBs, WARS, and EARs can use in a WildFly environment. See the WildFly documentation for more details on this file. It allows you to pull in external dependencies among other fine grain actions.

`ProviderFactory` implementations are required to be plain java objects. But, we also currently support implementing provider classes as Stateful EJBs. This is how you would do it:

```
@Stateful
@Local(EjbExampleUserStorageProvider.class)
public class EjbExampleUserStorageProvider implements User-
StorageProvider,
    UserLookupProvider,
    UserRegistrationProvider,
    UserQueryProvider,
    CredentialInputUpdater,
    CredentialInputValidator,
    OnUserCache
{
    @PersistenceContext
    protected EntityManager em;

    protected ComponentModel model;
    protected KeycloakSession session;

    public void setModel(ComponentModel model) {
        this.model = model;
    }

    public void setSession(KeycloakSession session) {
```

```

        this.session = session;
    }

    @Remove
    @Override
    public void close() {
    }
}

...
}

```

You have to define the `@Local` annotation and specify your provider class there. If you don't do this, EJB will not proxy the provider instance correctly and your provider won't work.

You must put the `@Remove` annotation on the `close()` method of your provider. If you don't, the stateful bean will never be cleaned up and you may eventually see error messages.

Implementations of `ProviderFactory` are required to be plain java objects. Your factory class would perform a JNDI lookup of the Stateful EJB in its `create()` method.

```

public class EjbExampleUserStorageProviderFactory
    implements UserStorageProviderFactory<EjbExampleUserStorageProvider> {

    @Override
    public EjbExampleUserStorageProvider create(KeycloakSession session, ComponentModel model) {
        try {
            InitialContext ctx = new InitialContext();
            EjbExampleUserStorageProvider provider = (EjbExampleUserStorageProvider)ctx.lookup(
                "java:global/user-storage-jpa-example/" + EjbExampleUserStorageProvider.class.getSimpleName());
        }
    }
}

```

```
        provider.setModel(model);
        provider.setSession(session);
        return provider;
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

## 6.4. Available SPIs

If you want to see list of all available SPIs at runtime, you can check **Server Info** page in admin console as described in [Admin Console](#) section.

---

## 7. User Storage SPI

You can use the User Storage SPI to write extensions to Keycloak to connect to external user databases and credential stores. The built-in LDAP and ActiveDirectory support is an implementation of this SPI in action. Out of the box, Keycloak uses its local database to create, update, and look up users and validate credentials. Often though, organizations have existing external proprietary user databases that they cannot migrate to Keycloak's data model. For those situations, application developers can write implementations of the User Storage SPI to bridge the external user store and the internal user object model that Keycloak uses to log in users and manage them.

When the Keycloak runtime needs to look up a user, such as when a user is logging in, it performs a number of steps to locate the user. It first looks to see if the user is in the user cache; if the user is found it uses that in-memory representation. Then it looks for the user within the Keycloak local database. If the user is not found, it then loops through User Storage SPI provider implementations to perform the user query until one of them returns the user the runtime is looking for. The provider queries the external user store for the user and maps the external data representation of the user to Keycloak's user metamodel.

User Storage SPI provider implementations can also perform complex criteria queries, perform CRUD operations on users, validate and manage credentials, or perform bulk updates of many users at once. It depends on the capabilities of the external store.

User Storage SPI provider implementations are packaged and deployed similarly to (and often are) Java EE components. They are not enabled by default, but instead must be enabled and configured per realm under the `User Federation` tab in the administration console.

## 7.1. Provider Interfaces

When building an implementation of the User Storage SPI you have to define a provider class and a provider factory. Provider class instances are created per transaction by provider factories. Provider classes do all the heavy lifting of user lookup and other user operations. They must implement the `org.keycloak.storage.UserStorageProvider` interface.

```
package org.keycloak.storage;

public interface UserStorageProvider extends Provider {

    /**
     * Callback when a realm is removed. Implement this
     * if, for example, you want to do some
     * cleanup in your user storage when a realm is removed
     *
     * @param realm
     */
    default
    void preRemove(RealmModel realm) {

    }

    /**
     * Callback when a group is removed. Allows you to do
     * things like remove a user
     * group mapping in your external store if appropriate
     *
     * @param realm
     */
}
```

```

    * @param group
    */
default
void preRemove(RealmModel realm, GroupModel group) {

}

/**
 * Callback when a role is removed. Allows you to do
things like remove a user
 * role mapping in your external store if appropriate

 * @param realm
 * @param role
 */
default
void preRemove(RealmModel realm, RoleModel role) {

}

}

```

You may be thinking that the `UserStorageProvider` interface is pretty sparse? You'll see later in this chapter that there are other mix-in interfaces your provider class may implement to support the meat of user integration.

`UserStorageProvider` instances are created once per transaction. When the transaction is complete, the `UserStorageProvider.close()` method is invoked and the instance is then garbage collected. Instances are created by provider factories. Provider factories implement the `org.keycloak.storage.UserStorageProvider-Factory` interface.

```
package org.keycloak.storage;
```

```
/**
```

```

* @author <a href="mailto:bill@burkecentral.com">Bill
Burke</a>
* @version $Revision: 1 $
*/
public interface UserStorageProviderFactory<T extends User-
StorageProvider> extends ComponentFactory<T, UserStorage-
Provider> {

    /**
     * This is the name of the provider and will be shown
     in the admin console as an option.
     *
     * @return
     */
    @Override
    String getId();

    /**
     * called per Keycloak transaction.
     *
     * @param session
     * @param model
     * @return
     */
    T create(KeycloakSession session, ComponentModel
model);
    ...
}

```

Provider factory classes must specify the concrete provider class as a template parameter when implementing the `UserStorageProviderFactory`. This is a must as the runtime will introspect this class to scan for its capabilities (the other interfaces it implements). So for example, if your provider class is named `FileProvider`, then the factory class should look like this:

```

public class FileProviderFactory implements UserStoragePro-
viderFactory<FileProvider> {

```

```
public String getId() { return "file-provider"; }

public FileProvider create(KeycloakSession session,
ComponentModel model) {
    ...
}
```

The `getId()` method returns the name of the User Storage provider. This id will be displayed in the admin console's User Federation page when you want to enable the provider for a specific realm.

The `create()` method is responsible for allocating an instance of the provider class. It takes a `org.keycloak.models.KeycloakSession` parameter. This object can be used to look up other information and metadata as well as provide access to various other components within the runtime. The `ComponentModel` parameter represents how the provider was enabled and configured within a specific realm. It contains the instance id of the enabled provider as well as any configuration you may have specified for it when you enabled through the admin console.

The `UserStorageProviderFactory` has other capabilities as well which we will go over later in this chapter.

## 7.2. Provider Capability Interfaces

If you have examined the `UserStorageProvider` interface closely you might notice that it does not define any methods for locating or managing users. These methods are actually defined in other *capability interfaces* depending on what scope of capabilities your external user store can provide and execute on. For example, some external stores are read-only and can only do simple queries and credential validation. You will only be required to implement the *capability interfaces* for the fea-

tures you are able to. You can implement these interfaces:

SPI	Description
<code>org.keycloak.storage.user.UserLookupProvider</code>	<p>This interface is required if you want to be able to log in with users from this external store. Most (all?) providers implement this interface.</p>
<code>org.keycloak.storage.user.UserQueryProvider</code>	<p>Defines complex queries that are used to locate one or more users. You must implement this interface if you want to view and manage users from the administration console.</p>
<code>org.keycloak.storage.user.UserRegistrationProvider</code>	<p>Implement this interface if your provider supports adding and removing users.</p>
<code>org.keycloak.storage.user.UserBulkUpdateProvider</code>	<p>Implement this interface if your provider supports bulk update of a set of users.</p>
<code>org.keycloak.credential.CredentialInputValidator</code>	<p>Implement this interface if your provider can validate one or more different credential types (for example, if your provider can validate a password).</p>

`org.keycloak.credential.CredentialInputUpdater`

Implement this interface if your provider supports updating one or more different credential types.

## 7.3. Model Interfaces

Most of the methods defined in the *capability interfaces* either return or are passed in representations of a user. These representations are defined by the `org.keycloak.models.UserModel` interface. App developers are required to implement this interface. It provides a mapping between the external user store and the user metamodel that KeyCloak uses.

```
package org.keycloak.models;

public interface UserModel extends RoleMapperModel {
    String getId();

    String getUsername();
    void setUsername(String username);

    String getFirstName();
    void setFirstName(String firstName);

    String getLastName();
    void setLastName(String lastName);

    String getEmail();
    void setEmail(String email);
    ...
}
```

`UserModel` implementations provide access to read and update metadata about the user including things like username, name, email, role

and group mappings, as well as other arbitrary attributes.

There are other model classes within the `org.keycloak.models` package that represent other parts of the Keycloak metamodel: `RealmModel`, `RoleModel`, `GroupModel`, and `ClientModel`.

### 7.3.1. Storage Ids

One important method of `UserModel` is the `getId()` method. When implementing `UserModel` developers must be aware of the user id format. The format must be:

```
"f:" + component id + ":" + external id
```

The Keycloak runtime often has to look up users by their user id. The user id contains enough information so that the runtime does not have to query every single `UserStorageProvider` in the system to find the user.

The component id is the id returned from `ComponentModel.getId()`. The `ComponentModel` is passed in as a parameter when creating the provider class so you can get it from there. The external id is information your provider class needs to find the user in the external store. This is often a username or a uid. For example, it might look something like this:

```
f:332a234e31234:wburke
```

When the runtime does a lookup by id, the id is parsed to obtain the component id. The component id is used to locate the `UserStorage-`

`Provider` that was originally used to load the user. That provider is then passed the id. The provider again parses the id to obtain the external id and it will use to locate the user in external user storage.

## 7.4. Packaging and Deployment

User Storage providers are packaged in a JAR and deployed or undeployed to the Keycloak runtime in the same way you would deploy something in the WildFly application server. You can either copy the JAR directly to the `standalone/deployments/` directory of the server, or use the JBoss CLI to execute the deployment.

In order for Keycloak to recognize the provider, you need to add a file to the JAR: `META-INF/services/org.keycloak.storage.UserStorageProviderFactory`. This file must contain a line-separated list of fully qualified classnames of the `UserStorageProviderFactory` implementations:

```
org.keycloak.examples.federation.properties.ClasspathPropertiesStorageFactory  
org.keycloak.examples.federation.properties.FilePropertiesStorageFactory
```

Keycloak supports hot deployment of these provider JARs. You'll also see later in this chapter that you can package it within and as Java EE components.

## 7.5. Simple Read-Only, Lookup Example

To illustrate the basics of implementing the User Storage SPI let's walk through a simple example. In this chapter you'll see the implementation

of a simple `UserStorageProvider` that looks up users in a simple property file. The property file contains username and password definitions and is hardcoded to a specific location on the classpath. The provider will be able to look up the user by ID and username and also be able to validate passwords. Users that originate from this provider will be read-only.

### 7.5.1. Provider Class

The first thing we will walk through is the `UserStorageProvider` class.

```
public class PropertyFileUserStorageProvider implements
    UserStorageProvider,
    UserLookupProvider,
    CredentialInputValidator,
    CredentialInputUpdater
{
    ...
}
```

Our provider class, `PropertyFileUserStorageProvider`, implements many interfaces. It implements the `UserStorageProvider` as that is a base requirement of the SPI. It implements the `UserLookupProvider` interface because we want to be able to log in with users stored by this provider. It implements the `CredentialInputValidator` interface because we want to be able to validate passwords entered in using the login screen. Our property file is read-only. We implement the `CredentialInputUpdater` because we want to post an error condition when the user attempts to update his password.

```
protected KeycloakSession session;
```

```

protected Properties properties;
protected ComponentModel model;
// map of loaded users in this transaction
protected Map<String, UserModel> loadedUsers = new
HashMap<>();

public PropertyFileUserStorageProvider(KeycloakSession session, ComponentModel model, Properties properties) {
    this.session = session;
    this.model = model;
    this.properties = properties;
}

```

The constructor for this provider class is going to store the reference to the `KeycloakSession`, `ComponentModel`, and property file. We'll use all of these later. Also notice that there is a map of loaded users. Whenever we find a user we will store it in this map so that we avoid re-creating it again within the same transaction. This is a good practice to follow as many providers will need to do this (that is, any provider that integrates with JPA). Remember also that provider class instances are created once per transaction and are closed after the transaction completes.

## UserLookupProvider Implementation

```

@Override
public UserModel getUserByUsername(String username, Re-
almModel realm) {
    UserModel adapter = loadedUsers.get(username);
    if (adapter == null) {
        String password = properties.getProperty(user-
name);
        if (password != null) {
            adapter = createAdapter(realm, username);
            loadedUsers.put(username, adapter);
        }
    }
    return adapter;
}

```

```

    }

    protected UserModel createAdapter(RealmModel realm,
String username) {
    return new AbstractUserAdapter(session, realm,
model) {
        @Override
public String getUsername() {
            return username;
        }
    };
}

@Override
public UserModel getUserById(String id, RealmModel
realm) {
    StorageId storageId = new StorageId(id);
    String username = storageId.getExternalId();
    return getUserByUsername(username, realm);
}

@Override
public UserModel getUserByEmail(String email, RealmMo-
del realm) {
    return null;
}

```

The `getUserByUsername()` method is invoked by the Keycloak login page when a user logs in. In our implementation we first check the `loadedUsers` map to see if the user has already been loaded within this transaction. If it hasn't been loaded we look in the property file for the username. If it exists we create an implementation of `UserModel`, store it in `loadedUsers` for future reference, and return this instance.

The `createAdapter()` method uses the helper class `org.keycloak.storage.adapter.AbstractUserAdapter`. This provides a base implementation for `UserModel`. It automatically generates a user id based on the required storage id format using the username of

the user as the external id.

```
"f:" + component_id + ":" + username
```

Every get method of `AbstractUserAdapter` either returns null or empty collections. However, methods that return role and group mappings will return the default roles and groups configured for the realm for every user. Every set method of `AbstractUserAdapter` will throw a `org.keycloak.storage.ReadOnlyException`. So if you attempt to modify the user in the administration console, you will get an error.

The `getUserById()` method parses the `id` parameter using the `org.keycloak.storage.StorageId` helper class. The `StorageId.getExternalId()` method is invoked to obtain the username embedded in the `id` parameter. The method then delegates to `getUserByUsername()`.

Emails are not stored, so the `getUserByEmail()` method returns null.

## CredentialInputValidator Implementation

Next let's look at the method implementations for `CredentialInputValidator`.

```
@Override  
public boolean isConfiguredFor(RealmModel realm, User-  
Model user, String credentialType) {  
    String password = properties.getProperty(user.  
getUsername());  
    return credentialType.equals(CredentialModel.PASS-  
WORD) && password != null;  
}
```

```

    @Override
    public boolean supportsCredentialType(String credentialType) {
        return credentialType.equals(CredentialModel.PASSWORD);
    }

    @Override
    public boolean isValid(RealmModel realm, UserModel user, CredentialInput input) {
        if (!supportsCredentialType(input.getType()) || !(input instanceof UserCredentialModel)) return false;

        UserCredentialModel cred = (UserCredentialModel)input;
        String password = properties.getProperty(user.getUsername());
        if (password == null) return false;
        return password.equals(cred.getValue());
    }

```

The `isConfiguredFor()` method is called by the runtime to determine if a specific credential type is configured for the user. This method checks to see that the password is set for the user.

The `supportsCredentialType()` method returns whether validation is supported for a specific credential type. We check to see if the credential type is `password`.

The `isValid()` method is responsible for validating passwords. The `CredentialInput` parameter is really just an abstract interface for all credential types. We make sure that we support the credential type and also that it is an instance of `UserCredentialModel`. When a user logs in through the login page, the plain text of the password input is put into an instance of `UserCredentialModel`. The `isValid()` method

checks this value against the plain text password stored in the properties file. A return value of `true` means the password is valid.

## CredentialInputUpdater Implementation

As noted before, the only reason we implement the `CredentialInputUpdater` interface in this example is to forbid modifications of user passwords. The reason we have to do this is because otherwise the runtime would allow the password to be overridden in Keycloak local storage. We'll talk more about this later in this chapter.

```
@Override
public boolean updateCredential(RealmModel realm, UserModel user, CredentialInput input) {
    if (input.getType().equals(CredentialModel.PASSWORD)) throw new ReadOnlyException("user is read only for
this update");

    return false;
}

@Override
public void disableCredentialType(RealmModel realm,
UserModel user, String credentialType) {

}

@Override
public Set<String> getDisableableCredentialTypes(Realm-
Model realm, UserModel user) {
    return Collections.EMPTY_SET;
}
```

The `updateCredential()` method just checks to see if the credential type is password. If it is, a `ReadOnlyException` is thrown.

### 7.5.2. Provider Factory Implementation

Now that the provider class is complete, we now turn our attention to the provider factory class.

```
public class PropertyFileUserStorageProviderFactory
    implements UserStorageProviderFactory<PropertyFileUserStorageProvider> {

    public static final String PROVIDER_NAME = "readonly-
property-file";

    @Override
    public String getId() {
        return PROVIDER_NAME;
    }
}
```

First thing to notice is that when implementing the `UserStorageProviderFactory` class, you must pass in the concrete provider class implementation as a template parameter. Here we specify the provider class we defined before: `PropertyFileUserStorageProvider`.

If you do not specify the template parameter, your provider will not function. The runtime does class introspection to determine the *capability interfaces* that the provider implements.

The `getId()` method identifies the factory in the runtime and will also be the string shown in the admin console when you want to enable a user storage provider for the realm.

## Initialization

```
private static final Logger logger = Logger.getLog-
ger(PropertyFileUserStorageProviderFactory.class);
```

```

protected Properties properties = new Properties();

@Override
public void init(Config.Scope config) {
    InputStream is = getClass().getClassLoader().getRe-
sourceAsStream("/users.properties");

    if (is == null) {
        logger.warn("Could not find users.properties in
classpath");
    } else {
        try {
            properties.load(is);
        } catch (IOException ex) {
            logger.error("Failed to load users.proper-
ties file", ex);
        }
    }
}

@Override
public PropertyFileUserStorageProvider create(Keycloak-
Session session, ComponentModel model) {
    return new PropertyFileUserStorageProvider(session,
model, properties);
}

```

The `UserStorageProviderFactory` interface has an optional `init()` method you can implement. When KeyCloak boots up, only one instance of each provider factory is created. Also at boot time, the `init()` method is called on each of these factory instances. There's also a `postInit()` method you can implement as well. After each factory's `init()` method is invoked, their `postInit()` methods are called.

In our `init()` method implementation, we find the property file containing our user declarations from the classpath. We then load the `pro-`  
`perties` field with the username and password combinations stored

there.

The `Config.Scope` parameter is factory configuration that can be set up within `standalone.xml`, `standalone-ha.xml`, or `domain.xml`. For more information on where these files reside see the [{installguide!}](#).

For example, by adding the following to `standalone.xml`:

```
<spi name="storage">
    <provider name="readonly-property-file" enabled="true">
        <properties>
            <property name="path" value="/other-users.properties"/>
        </properties>
    </provider>
</spi>
```

We can specify the classpath of the user property file instead of hardcoding it. Then you can retrieve the configuration in the `PropertyFileUserStorageProviderFactory.init()`:

```
public void init(Config.Scope config) {
    String path = config.get("path");
    InputStream is = getClass().getClassLoader().getResourceAsStream(path);

    ...
}
```

## Create Method

Our last step in creating the provider factory is the `create()` method.

```
@Override  
public PropertyFileUserStorageProvider create(KeycloakSession session, ComponentModel model) {  
    return new PropertyFileUserStorageProvider(session,  
model, properties);  
}
```

We simply allocate the `PropertyFileUserStorageProvider` class. This create method will be called once per transaction.

### 7.5.3. Packaging and Deployment

The class files for our provider implementation should be placed in a jar. You also have to declare the provider factory class within the `META-INF/services/org.keycloak.storage.UserStorageProviderFactory` file.

```
org.keycloak.examples.federation.properties.FilePropertiesStorageFactory
```

Once you create the jar you can deploy it using regular WildFly means: copy the jar into the `standalone/deployments/` directory or using the JBoss CLI.

### 7.5.4. Enabling the Provider in the Administration Console

You enable user storage providers per realm within the `User Federation` page in the administration console.

Select the provider we just created from the list: `readonly-property-file`. It brings you to the configuration page for our provider. We do not have anything to configure, so click **Save**.

When you go back to the main `User Federation` page, you now see your provider listed.

You will now be able to log in with a user declared in the `users.properties` file. This user will only be able to view the account page after logging in.

## 7.6. Configuration Techniques

Our `PropertyFileUserStorageProvider` example is bit contrived. It is hardcoded to a property file that is embedded in the jar of the provider, which is not terribly useful. We might want to make the location of this file configurable per instance of the provider. In other words, we might want to reuse this provider multiple times in multiple different realms and point to completely different user property files. We'll also want to perform this configuration within the administration console UI.

The `UserStorageProviderFactory` has additional methods you can implement that handle provider configuration. You describe the variables you want to configure per provider and the administration console automatically renders a generic input page to gather this configuration. When implemented, callback methods also validate the configuration before it is saved, when a provider is created for the first time, and when it is updated. `UserStorageProviderFactory` inherits these methods from the `org.keycloak.component.ComponentFactory` interface.

```
List<ProviderConfigProperty> getConfigProperties();  
  
default  
void validateConfiguration(KeycloakSession session, Re-
```

```

    almModel realm, ComponentModel model)
        throws ComponentValidationException
    {

    }

    default
    void onCreate(KeycloakSession session, RealmModel
realm, ComponentModel model) {

}

    default
    void onUpdate(KeycloakSession session, RealmModel
realm, ComponentModel model) {

}

```

The `ComponentFactory.getConfigProperties()` method returns a list of `org.keycloak.provider.ProviderConfigProperty` instances. These instances declare metadata that is needed to render and store each configuration variable of the provider.

### 7.6.1. Configuration Example

Let's expand our `PropertyFileUserStorageProviderFactory` example to allow you to point a provider instance to a specific file on disk.

#### *PropertyFileUserStorageProviderFactory*

```

public class PropertyFileUserStorageProviderFactory
    implements UserStorageProviderFacto-
ry<PropertyFileUserStorageProvider> {

    protected static final List<ProviderConfigProperty>
configMetadata;

    static {
        configMetadata = ProviderConfigurationBuilder.crea-

```

```

        te()
            .property().name("path")
            .type(ProviderConfigProperty.STRING_TYPE)
            .label("Path")
            .defaultValue("${jboss.server.config.
dir}/example-users.properties")
            .helpText("File path to properties file")
            .add().build();
    }

    @Override
    public List<ProviderConfigProperty> getConfigProperties() {
        return configMetadata;
    }
}

```

The `ProviderConfigurationBuilder` class is a great helper class to create a list of configuration properties. Here we specify a variable named `path` that is a String type. On the administration console configuration page for this provider, this configuration variable is labeled as `Path` and has a default value of `${jboss.server.config.dir}/example-users.properties`. When you hover over the tooltip of this configuration option, it displays the help text, `File path to properties file`.

The next thing we want to do is to verify that this file exists on disk. We do not want to enable an instance of this provider in the realm unless it points to a valid user property file. To do this, we implement the `validateConfiguration()` method.

```

@Override
public void validateConfiguration(KeycloakSession session, RealmModel realm, ComponentModel config)
    throws ComponentValidationException {
    String fp = config.getConfig().getFirst("path");
    if (fp == null) throw new ComponentValidationExcep-
}

```

```

        tion("user property file does not exist");
        fp = EnvUtil.replace(fp);
        File file = new File(fp);
        if (!file.exists()) {
            throw new ComponentValidationException("user
property file does not exist");
        }
    }
}

```

In the `validateConfiguration()` method we get the configuration variable from the `ComponentModel` and we check to see if that file exists on disk. Notice that we use the `org.keycloak.common.util.EnvUtil.replace()` method. With this method any string that has  `${}` within it will replace that with a system property value. The  `${jboss.server.config.dir}` string corresponds to the `configuration/` directory of our server and is really useful for this example.

Next thing we have to do is remove the old `init()` method. We do this because user property files are going to be unique per provider instance. We move this logic to the `create()` method.

```

@Override
public PropertyFileUserStorageProvider create(Keycloak-
Session session, ComponentModel model) {
    String path = model.getConfig().getFirst("path");

    Properties props = new Properties();
    try {
        InputStream is = new FileInputStream(path);
        props.load(is);
        is.close();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }

    return new PropertyFileUserStorageProvider(session,
model, props);
}

```

```
}
```

This logic is, of course, inefficient as every transaction reads the entire user property file from disk, but hopefully this illustrates, in a simple way, how to hook in configuration variables.

### 7.6.2. Configuring the Provider in the Administration Console

Now that the configuration is enabled, you can set the `path` variable when you configure the provider in the administration console.

## 7.7. Add/Remove User and Query Capability interfaces

One thing we have not done with our example is allow it to add and remove users or change passwords. Users defined in our example are also not queryable or viewable in the administration console. To add these enhancements, our example provider must implement the `UserQueryProvider` and `UserRegistrationProvider` interfaces.

### 7.7.1. Implementing UserRegistrationProvider

To implement adding and removing users from this particular store, we first have to be able to save our properties file to disk.

#### *PropertyFileUserStorageProvider*

```
public void save() {
    String path = model.getConfig().getFirst("path");
    path = EnvUtil.replace(path);
    try {
        FileOutputStream fos = new FileOutputStream(path);
        properties.store(fos, "");
        fos.close();
    } catch (IOException e) {
```

```

        throw new RuntimeException(e);
    }
}

```

Then, the implementation of the `addUser()` and `removeUser()` methods becomes simple.

### *PropertyFileUserStorageProvider*

```

public static final String UNSET_PASSWORD="#$!-UNSET-
PASSWORD";

@Override
public UserModel addUser(RealmModel realm, String user-
name) {
    synchronized (properties) {
        properties.setProperty(username, UNSET_PASS-
WORD);
        save();
    }
    return createAdapter(realm, username);
}

@Override
public boolean removeUser(RealmModel realm, UserModel
user) {
    synchronized (properties) {
        if (properties.remove(user.getUsername()) ==
null) return false;
        save();
        return true;
    }
}

```

Notice that when adding a user we set the password value of the property map to be `UNSET_PASSWORD`. We do this as we can't have null values for a property in the property value. We also have to modify the `CredentialInputValidator` methods to reflect this.

The `addUser()` method will be called if the provider implements the `UserRegistrationProvider` interface. If your provider has a configuration switch to turn off adding a user, returning `null` from this method will skip the provider and call the next one.

### *PropertyFileUserStorageProvider*

```
@Override  
public boolean isValid(RealmModel realm, UserModel  
user, CredentialInput input) {  
    if (!supportsCredentialType(input.getType()) || !  
(input instanceof UserCredentialModel)) return false;  
  
    UserCredentialModel cred = (UserCredentialMo-  
del)input;  
    String password = properties.getProperty(user.  
getUsername());  
    if (password == null || UNSET_PASSWORD.equals(pass-  
word)) return false;  
    return password.equals(cred.getValue());  
}
```

Since we can now save our property file, it also makes sense to allow password updates.

### *PropertyFileUserStorageProvider*

```
@Override  
public boolean updateCredential(RealmModel realm, User-  
Model user, CredentialInput input) {  
    if (!(input instanceof UserCredentialModel)) return  
false;  
    if (!input.getType().equals(CredentialModel.PASS-  
WORD)) return false;  
    UserCredentialModel cred = (UserCredentialMo-  
del)input;  
    synchronized (properties) {  
        properties.setProperty(user.getUsername(),  
cred.getValue());
```

```
        save();
    }
    return true;
}
```

We can now also implement disabling a password.

### *PropertyFileUserStorageProvider*

```
@Override
public void disableCredentialType(RealmModel realm,
 UserModel user, String credentialType) {
    if (!credentialType.equals(CredentialModel.PASS-
WORD)) return;
    synchronized (properties) {
        properties.setProperty(user.getUsername(), UN-
SET_PASSWORD);
        save();
    }

}

private static final Set<String> disableableTypes = new
HashSet<>();

static {
    disableableTypes.add(CredentialModel.PASSWORD);
}

@Override
public Set<String> getDisableableCredentialTypes(Realm-
Model realm, UserModel user) {

    return disableableTypes;
}
```

With these methods implemented, you'll now be able to change and disable the password for the user in the administration console.

#### **7.7.2. Implementing UserQueryProvider**

Without implementing `UserQueryProvider` the administration console would not be able to view and manage users that were loaded by our example provider. Let's look at implementing this interface.

### *PropertyFileUserStorageProvider*

```
@Override  
public int getUsersCount(RealmModel realm) {  
    return properties.size();  
}  
  
@Override  
public List<UserModel> getUsers(RealmModel realm) {  
    return getUsers(realm, 0, Integer.MAX_VALUE);  
}  
  
@Override  
public List<UserModel> getUsers(RealmModel realm, int  
firstResult, int maxResults) {  
    List<UserModel> users = new LinkedList<>();  
    int i = 0;  
    for (Object obj : properties.keySet()) {  
        if (i++ < firstResult) continue;  
        String username = (String)obj;  
        UserModel user = getUserByUsername(username,  
realm);  
        users.add(user);  
        if (users.size() >= maxResults) break;  
    }  
    return users;  
}
```

The `getUsers()` method iterates over the key set of the property file, delegating to `getUserByUsername()` to load a user. Notice that we are indexing this call based on the `firstResult` and `maxResults` parameter. If your external store does not support pagination, you will have to do similar logic.

## *PropertyFileUserStorageProvider*

```
    @Override
    public List<UserModel> searchForUser(String search, Re-
almModel realm) {
        return searchForUser(search, realm, 0, Integer.MA-
X_VALUE);
    }

    @Override
    public List<UserModel> searchForUser(String search, Re-
almModel realm, int firstResult, int maxResults) {
        List<UserModel> users = new LinkedList<>();
        int i = 0;
        for (Object obj : properties.keySet()) {
            String username = (String)obj;
            if (!username.contains(search)) continue;
            if (i++ < firstResult) continue;
            UserModel user = getUserByUsername(username,
realm);
            users.add(user);
            if (users.size() >= maxResults) break;
        }
        return users;
    }
```

The first declaration of `searchForUser()` takes a `String` parameter. This is supposed to be a string that you use to search username and email attributes to find the user. This string can be a substring, which is why we use the `String.contains()` method when doing our search.

## *PropertyFileUserStorageProvider*

```
    @Override
    public List<UserModel> searchForUser(Map<String,
String> params, RealmModel realm) {
        return searchForUser(params, realm, 0, Integer.MA-
X_VALUE);
    }
```

```

@Override
public List<UserModel> searchForUser(Map<String,
String> params, RealmModel realm, int firstResult, int max-
Results) {
    // only support searching by username
    String usernameSearchString = params.get("userna-
me");
    if (usernameSearchString == null) return Collections.EMPTY_LIST;
    return searchForUser(usernameSearchString, realm,
firstResult, maxResults);
}

```

The `searchForUser()` method that takes a `Map` parameter can search for a user based on first, last name, username, and email. We only store usernames, so we only search based on usernames. We delegate to `searchForUser()` for this.

### *PropertyFileUserStorageProvider*

```

@Override
public List<UserModel> getGroupMembers(RealmModel
realm, GroupModel group, int firstResult, int maxResults) {
    return Collections.EMPTY_LIST;
}

@Override
public List<UserModel> getGroupMembers(RealmModel
realm, GroupModel group) {
    return Collections.EMPTY_LIST;
}

@Override
public List<UserModel> searchForUserByUserAttribu-
te(String attrName, String attrValue, RealmModel realm) {
    return Collections.EMPTY_LIST;
}

```

We do not store groups or attributes, so the other methods return an

empty list.

## 7.8. Augmenting External Storage

The `PropertyFileUserStorageProvider` example is really limited. While we will be able to login with users stored in a property file, we won't be able to do much else. If users loaded by this provider need special role or group mappings to fully access particular applications there is no way for us to add additional role mappings to these users. You also can't modify or add additional important attributes like email, first and last name.

For these types of situations, KeyCloak allows you to augment your external store by storing extra information in KeyCloak's database. This is called federated user storage and is encapsulated within the `org.keycloak.storage.federated.UserFederatedStorageProvider` class.

### *UserFederatedStorageProvider*

```
package org.keycloak.storage.federated;

public interface UserFederatedStorageProvider extends Provider {

    Set<GroupModel> getGroups(RealmModel realm, String user);
    void joinGroup(RealmModel realm, String userId, GroupModel group);
    void leaveGroup(RealmModel realm, String userId, GroupModel group);
    List<String> getMembership(RealmModel realm, GroupModel group, int firstResult, int max);

    ...
}
```

The `UserFederatedStorageProvider` instance is available on the `KeycloakSession.userFederatedStorage()` method. It has all different kinds of methods for storing attributes, group and role mappings, different credential types, and required actions. If your external store's datamodel cannot support the full Keycloak feature set, then this service can fill in the gaps.

Keycloak comes with a helper class `org.keycloak.storage.adapter.AbstractUserAdapterFederatedStorage` that will delegate every single `UserModel` method except get/set of username to user federated storage. Override the methods you need to override to delegate to your external storage representations. It is strongly suggested you read the javadoc of this class as it has smaller protected methods you may want to override. Specifically surrounding group membership and role mappings.

### 7.8.1. Augmentation Example

In our `PropertyFileUserStorageProvider` example, we just need a simple change to our provider to use the `AbstractUserAdapterFederatedStorage`.

#### *PropertyFileUserStorageProvider*

```
protected UserModel createAdapter(RealmModel realm,
String username) {
    return new AbstractUserAdapterFederatedStorage(session, realm, model) {
        @Override
        public String getUsername() {
            return username;
        }

        @Override
```

```

public void setUsername(String username) {
    String pw = (String)properties.remove(user-
name);
    if (pw != null) {
        properties.put(username, pw);
        save();
    }
}
};

}

```

We instead define an anonymous class implementation of `AbstractUserAdapterFederatedStorage`. The `setUsername()` method makes changes to the properties file and saves it.

## 7.9. Import Implementation Strategy

When implementing a user storage provider, there's another strategy you can take. Instead of using user federated storage, you can create a user locally in the Keycloak built-in user database and copy attributes from your external store into this local copy. There are many advantages to this approach.

- Keycloak basically becomes a persistence user cache for your external store. Once the user is imported you'll no longer hit the external store thus taking load off of it.
- If you are moving to Keycloak as your official user store and deprecating the old external store, you can slowly migrate applications to use Keycloak. When all applications have been migrated, unlink the imported user, and retire the old legacy external store.

There are some obvious disadvantages though to using an import strategy:

- Looking up a user for the first time will require multiple updates to Keycloak database. This can be a big performance loss under load and put a lot of strain on the Keycloak database. The user federated storage approach will only store extra data as needed and may never be used depending on the capabilities of your external store.
- With the import approach, you have to keep local Keycloak storage and external storage in sync. The User Storage SPI has capability interfaces that you can implement to support synchronization, but this can quickly become painful and messy.

To implement the import strategy you simply check to see first if the user has been imported locally. If so return the local user, if not create the user locally and import data from the external store. You can also proxy the local user so that most changes are automatically synchronized.

This will be a bit contrived, but we can extend our `PropertyFileUserStorageProvider` to take this approach. We begin first by modifying the `createAdapter()` method.

### *PropertyFileUserStorageProvider*

```
protected UserModel createAdapter(RealmModel realm,
String username) {
    UserModel local = session.userLocalStorage().getUserByUsername(username, realm);
    if (local == null) {
        local = session.userLocalStorage().addUser(realm, username);
        local.setFederationLink(model.getId());
    }
    return new UserModelDelegate(local) {
        @Override
```

```

    public void setUsername(String username) {
        String pw = (String)properties.remove(user-
name);
        if (pw != null) {
            properties.put(username, pw);
            save();
        }
        super.setUsername(username);
    }
}

```

In this method we call the `KeycloakSession.userLocalStorage()` method to obtain a reference to local Keycloak user storage. We see if the user is stored locally, if not, we add it locally. Do not set the `id` of the local user. Let Keycloak automatically generate the `id`. Also note that we call `UserModel.setFederationLink()` and pass in the ID of the `ComponentModel` of our provider. This sets a link between the provider and the imported user.

When a user storage provider is removed, any user imported by it will also be removed. This is one of the purposes of calling `UserModel.setFederationLink()`.

Another thing to note is that if a local user is linked, your storage provider will still be delegated to for methods that it implements from the `CredentialInputValidator` and `CredentialInputUpdater` interfaces. Returning `false` from a validation or update will just result in Keycloak seeing if it can validate or update using local storage.

Also notice that we are proxying the local user using the `org.`

`keycloak.models.utils.UserModelDelegate` class. This class is an implementation of `UserModel`. Every method just delegates to the `UserModel` it was instantiated with. We override the `setUsername()` method of this delegate class to synchronize automatically with the property file. For your providers, you can use this to *intercept* other methods on the local `UserModel` to perform synchronization with your external store. For example, get methods could make sure that the local store is in sync. Set methods keep the external store in sync with the local one. One thing to note is that the `getId()` method should always return the id that was auto generated when you created the user locally. You should not return a federated id as shown in the other non-import examples.

If your provider is implementing the `UserRegistrationProvider` interface, your `removeUser()` method does not need to remove the user from local storage. The runtime will automatically perform this operation. Also note that `removeUser()` will be invoked before it is removed from local storage.

### 7.9.1. ImportedUserValidation Interface

If you remember earlier in this chapter, we discussed how querying for a user worked. Local storage is queried first, if the user is found there, then the query ends. This is a problem for our above implementation as we want to proxy the local `UserModel` so that we can keep usernames in sync. The User Storage SPI has a callback for whenever a linked local user is loaded from the local database.

```

package org.keycloak.storage.user;
public interface ImportedUserValidation {
    /**
     * If this method returns null, then the user in local
     storage will be removed
     *
     * @param realm
     * @param user
     * @return null if user no longer valid
     */
    UserModel validate(RealmModel realm, UserModel user);
}

```

Whenever a linked local user is loaded, if the user storage provider class implements this interface, then the `validate()` method is called.

Here you can proxy the local user passed in as a parameter and return it. That new `UserModel` will be used. You can also optionally do a check to see if the user still exists in the external store. If `validate()` returns `null`, then the local user will be removed from the database.

### 7.9.2. ImportSynchronization Interface

With the import strategy you can see that it is possible for the local user copy to get out of sync with external storage. For example, maybe a user has been removed from the external store. The User Storage SPI has an additional interface you can implement to deal with this, `org.keycloak.storage.user.ImportSynchronization`:

```

package org.keycloak.storage.user;

public interface ImportSynchronization {
    SynchronizationResult sync(KeycloakSessionFactory ses-
    sionFactory, String realmId, UserStorageProviderModel
    model);
    SynchronizationResult syncSince(Date lastSync, Keyclo-
    akSessionFactory sessionFactory, String realmId, UserStora-

```

```
    geProviderModel model);
}
```

This interface is implemented by the provider factory. Once this interface is implemented by the provider factory, the administration console management page for the provider shows additional options. You can manually force a synchronization by clicking a button. This invokes the `ImportSynchronization.sync()` method. Also, additional configuration options are displayed that allow you to automatically schedule a synchronization. Automatic synchronizations invoke the `syncSince()` method.

## 7.10. User Caches

When a user object is loaded by ID, username, or email queries it is cached. When a user object is being cached, it iterates through the entire `UserModel` interface and pulls this information to a local in-memory-only cache. In a cluster, this cache is still local, but it becomes an invalidation cache. When a user object is modified, it is evicted. This eviction event is propagated to the entire cluster so that the other nodes' user cache is also invalidated.

### 7.10.1. Managing the user cache

You can access the user cache by calling `KeycloakSession.userCache()`.

```
/**
 * All these methods effect an entire cluster of Keycloak
instances.
*
* @author <a href="mailto:bill@burkecentral.com">Bill
Burke</a>
```

```

* @version $Revision: 1 $
*/
public interface UserCache extends UserProvider {
    /**
     * Evict user from cache.
     *
     * @param user
     */
    void evict(RealmModel realm, UserModel user);

    /**
     * Evict users of a specific realm
     *
     * @param realm
     */
    void evict(RealmModel realm);

    /**
     * Clear cache entirely.
     *
     */
    void clear();
}

```

There are methods for evicting specific users, users contained in a specific realm, or the entire cache.

### 7.10.2. OnUserCache Callback Interface

You might want to cache additional information that is specific to your provider implementation. The User Storage SPI has a callback whenever a user is cached: `org.keycloak.models.cache.OnUserCache`.

```

public interface OnUserCache {
    void onCache(RealmModel realm, CachedUserModel user,
    UserModel delegate);
}

```

Your provider class should implement this interface if it wants this callback. The `UserModel` delegate parameter is the `UserModel` instance returned by your provider. The `CachedUserModel` is an expanded `UserModel` interface. This is the instance that is cached locally in local storage.

```
public interface CachedUserModel extends UserModel {  
  
    /**  
     * Invalidates the cache for this user and returns a  
     * delegate that represents the actual data provider  
     *  
     * @return  
     */  
    UserModel getDelegateForUpdate();  
  
    boolean isMarkedForEviction();  
  
    /**  
     * Invalidate the cache for this model  
     *  
     */  
    void invalidate();  
  
    /**  
     * When was the model was loaded from database.  
     *  
     * @return  
     */  
    long getCacheTimestamp();  
  
    /**  
     * Returns a map that contains custom things that are  
     * cached along with this model. You can write to this map.  
     *  
     * @return  
     */  
    ConcurrentHashMap getCachedWith();  
}
```

This `CachedUserModel` interface allows you to evict the user from the cache and get the provider `UserModel` instance. The `getCachedWith()` method returns a map that allows you to cache additional information pertaining to the user. For example, credentials are not part of the `UserModel` interface. If you wanted to cache credentials in memory, you would implement `OnUserCache` and cache your user's credentials using the `getCachedWith()` method.

### 7.10.3. Cache Policies

On the administration console management page for your user storage provider, you can specify a unique cache policy.

## 7.11. Leveraging Java EE

The user storage providers can be packaged within any Java EE component if you set up the `META-INF/services` file correctly to point to your providers. For example, if your provider needs to use third-party libraries, you can package up your provider within an EAR and store these third-party libraries in the `lib/` directory of the EAR. Also note that provider JARs can make use of the `jboss-deployment-structure.xml` file that EJBs, WARS, and EARs can use in a WildFly environment. For more details on this file, see the WildFly documentation. It allows you to pull in external dependencies among other fine-grained actions.

Provider implementations are required to be plain java objects. But we also currently support implementing `UserStorageProvider` classes as Stateful EJBs. This is especially useful if you want to use JPA to connect to a relational store. This is how you would do it:

```

@Stateful
@Local(EjbExampleUserStorageProvider.class)
public class EjbExampleUserStorageProvider implements User-
StorageProvider,
    UserLookupProvider,
    UserRegistrationProvider,
    UserQueryProvider,
    CredentialInputUpdater,
    CredentialInputValidator,
    OnUserCache
{
    @PersistenceContext
    protected EntityManager em;

    protected ComponentModel model;
    protected KeycloakSession session;

    public void setModel(ComponentModel model) {
        this.model = model;
    }

    public void setSession(KeycloakSession session) {
        this.session = session;
    }

    @Remove
    @Override
    public void close() {
    }
    ...
}

```

You have to define the `@Local` annotation and specify your provider class there. If you do not do this, EJB will not proxy the user correctly and your provider won't work.

You must put the `@Remove` annotation on the `close()` method of your provider. If you do not, the stateful bean will never be cleaned up

and you might eventually see error messages.

Implementations of `UserStorageProvider` are required to be plain Java objects. Your factory class would perform a JNDI lookup of the Stateful EJB in its `create()` method.

```
public class EjbExampleUserStorageProviderFactory
    implements UserStorageProviderFactory<EjbExampleUserStorageProvider> {

    @Override
    public EjbExampleUserStorageProvider create(KeycloakSession session, ComponentModel model) {
        try {
            InitialContext ctx = new InitialContext();
            EjbExampleUserStorageProvider provider = (EjbExampleUserStorageProvider)ctx.lookup(
                "java:global/user-storage-jpa-example/" + EjbExampleUserStorageProvider.class.getSimpleName());
            provider.setModel(model);
            provider.setSession(session);
            return provider;
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

This example also assumes that you have defined a JPA deployment in the same JAR as the provider. This means a `persistence.xml` file as well as any JPA `@Entity` classes.

When using JPA any additional datasource must be an XA datasource. The Keycloak datasource is not an XA datasource. If you interact with two or more non-XA

datasources in the same transaction, the server returns an error message. Only one non-XA resource is permitted in a single transaction. See the WildFly manual for more details on deploying an XA datasource.

CDI is not supported.

## 7.12. REST Management API

You can create, remove, and update your user storage provider deployments through the administrator REST API. The User Storage SPI is built on top of a generic component interface so you will be using that generic API to manage your providers.

The REST Component API lives under your realm admin resource.

```
/admin/realms/{realm-name}/components
```

We will only show this REST API interaction with the Java client. Hopefully you can extract how to do this from `curl` from this API.

```
public interface ComponentsResource {
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<ComponentRepresentation> query();

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<ComponentRepresentation> query(@Query-
Param("parent") String parent);

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<ComponentRepresentation> query(@Query-
```

```

Param("parent") String parent, @QueryParam("type") String
type);

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<ComponentRepresentation> query(@Query-
Param("parent") String parent,
                                              @Query-
Param("type") String type,
                                              @Query-
Param("name") String name);

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    Response add(ComponentRepresentation rep);

    @Path("{id}")
    ComponentResource component(@PathParam("id") String
id);
}

public interface ComponentResource {
    @GET
    public ComponentRepresentation toRepresentation();

    @PUT
    @Consumes(MediaType.APPLICATION_JSON)
    public void update(ComponentRepresentation rep);

    @DELETE
    public void remove();
}

```

To create a user storage provider, you must specify the provider id, a provider type of the string `org.keycloak.storage.UserStorageProvider`, as well as the configuration.

```

import org.keycloak.admin.client.Keycloak;
import org.keycloak.representations.idm.RealmRepresentati-
on;

```

```

...
Keycloak keycloak = Keycloak.getInstance(
    "http://localhost:8080/auth",
    "master",
    "admin",
    "password",
    "admin-cli");
RealmResource realmResource = keycloak.realm("master");
RealmRepresentation realm = realmResource.toRepresentation();

ComponentRepresentation component = new ComponentRepresentation();
component.setName("home");
component.setProviderId("readonly-property-file");
component.setProviderType("org.keycloak.storage.UserStorageProvider");
component.setParentId(realm.getId());
component.setConfig(new MultivaluedHashMap());
component.getConfig().putSingle("path", "~/users.properties");

realmResource.components().add(component);

// retrieve a component

List<ComponentRepresentation> components = realmResource.components().query(realm.getId(),
    "org.keycloak.storage.UserStorageProvider",
    "home");
component = components.get(0);

// Update a component

component.getConfig().putSingle("path", "~/my-users.properties");
realmResource.components().component(component.getId()).update(component);

// Remove a component

```

```
realmREsource.components().component(component.getId()).remove();
```

## 7.13. Migrating from an Earlier User Federation SPI

This chapter is only applicable if you have implemented a provider using the earlier (and now removed) User Federation SPI.

In Keycloak version 2.4.0 and earlier there was a User Federation SPI. Red Hat Single Sign-On version 7.0, although unsupported, had this earlier SPI available as well. This earlier User Federation SPI has been removed from Keycloak version 2.5.0 and Red Hat Single Sign-On version 7.1. However, if you have written a provider with this earlier SPI, this chapter discusses some strategies you can use to port it.

### 7.13.1. Import vs. Non-Import

The earlier User Federation SPI required you to create a local copy of a user in the Keycloak's database and import information from your external store to the local copy. However, this is no longer a requirement. You can still port your earlier provider as-is, but you should consider whether a non-import strategy might be a better approach.

Advantages of the import strategy:

- Keycloak basically becomes a persistence user cache for your external store. Once the user is imported you'll no longer hit the external store, thus taking load off of it.

- If you are moving to Keycloak as your official user store and deprecating the earlier external store, you can slowly migrate applications to use Keycloak. When all applications have been migrated, unlink the imported user, and retire the earlier legacy external store.

There are some obvious disadvantages though to using an import strategy:

- Looking up a user for the first time will require multiple updates to Keycloak database. This can be a big performance loss under load and put a lot of strain on the Keycloak database. The user federated storage approach will only store extra data as needed and might never be used depending on the capabilities of your external store.
- With the import approach, you have to keep local Keycloak storage and external storage in sync. The User Storage SPI has capability interfaces that you can implement to support synchronization, but this can quickly become painful and messy.

### 7.13.2. `UserFederationProvider` vs. `UserStorageProvider`

The first thing to notice is that `UserFederationProvider` was a complete interface. You implemented every method in this interface. However, `UserStorageProvider` has instead broken up this interface into multiple capability interfaces that you implement as needed.

`UserFederationProvider.getUserByUsername()` and `getUserByEmail()` have exact equivalents in the new SPI. The difference between the two is how you import. If you are going to continue with an import strategy, you no longer call `KeycloakSession.userStorage().addUser()` to create the user locally. Instead you call `KeycloakSession`.

`akSession.userLocalStorage().addUser()`. The `userStorage()` method no longer exists.

The `UserFederationProvider.validateAndProxy()` method has been moved to an optional capability interface, `ImportedUserValidation`. You want to implement this interface if you are porting your earlier provider as-is. Also note that in the earlier SPI, this method was called every time the user was accessed, even if the local user is in the cache. In the later SPI, this method is only called when the local user is loaded from local storage. If the local user is cached, then the `ImportedUserValidation.validate()` method is not called at all.

The `UserFederationProvider.isValid()` method no longer exists in the later SPI.

The `UserFederationProvider` methods `synchronizeRegistrations()`, `registerUser()`, and `removeUser()` have been moved to the `UserRegistrationProvider` capability interface. This new interface is optional to implement so if your provider does not support creating and removing users, you don't have to implement it. If your earlier provider had switch to toggle support for registering new users, this is supported in the new SPI, returning `null` from `UserRegistrationProvider.addUser()` if the provider doesn't support adding users.

The earlier `UserFederationProvider` methods centered around credentials are now encapsulated in the `CredentialInputValidator` and `CredentialInputUpdater` interfaces, which are also optional to implement depending on if you support validating or updating credenti-

als. Credential management used to exist in `UserModel` methods. These also have been moved to the `CredentialInputValidator` and `CredentialInputUpdater` interfaces. One thing to note that if you do not implement the `CredentialInputUpdater` interface, then any credentials provided by your provider can be overridden locally in Keycloak storage. So if you want your credentials to be read-only, implement the `CredentialInputUpdater.updateCredential()` method and return a `ReadOnlyException`.

The `UserFederationProvider` query methods such as `searchByAttributes()` and `getGroupMembers()` are now encapsulated in an optional interface `UserQueryProvider`. If you do not implement this interface, then users will not be viewable in the admin console. You'll still be able to login though.

### 7.13.3. UserFederationProviderFactory vs. UserStorageProviderFactory

The synchronization methods in the earlier SPI are now encapsulated within an optional `ImportSynchronization` interface. If you have implemented synchronization logic, then have your new `UserStorageProviderFactory` implement the `ImportSynchronization` interface.

### 7.13.4. Upgrading to a New Model

The User Storage SPI instances are stored in a different set of relational tables. Keycloak automatically runs a migration script. If any earlier User Federation providers are deployed for a realm, they are converted to the later storage model as is, including the `id` of the data. This migration will only happen if a User Storage provider exists with the same

provider ID (i.e., "ldap", "kerberos") as the earlier User Federation provider.

So, knowing this there are different approaches you can take.

1. You can remove the earlier provider in your earlier Keycloak deployment. This will remove the local linked copies of all users you imported. Then, when you upgrade Keycloak, just deploy and configure your new provider for your realm.
2. The second option is to write your new provider making sure it has the same provider ID: `UserStorageProviderFactory.getId()`. Make sure this provider is in the `standalone/deployments/` directory of the new Keycloak installation. Boot the server, and have the built-in migration script convert from the earlier data model to the later data model. In this case all your earlier linked imported users will work and be the same.

If you have decided to get rid of the import strategy and rewrite your User Storage provider, we suggest that you remove the earlier provider before upgrading Keycloak. This will remove linked local imported copies of any user you imported.

Last updated 2019-06-13 12:48:57 MESZ

# Table of Contents

{authorizationguide!}

## 1. Overview

### 1.1. Architecture

#### 1.1.1. The Authorization Process

#### 1.1.2. Authorization Services

### 1.2. Terminology

#### 1.2.1. Resource Server

#### 1.2.2. Resource

#### 1.2.3. Scope

#### 1.2.4. Permission

#### 1.2.5. Policy

#### 1.2.6. Policy Provider

#### 1.2.7. Permission Ticket

## 2. Getting Started

### 2.1. Securing a Servlet Application

### 2.2. Creating a Realm and a User

### 2.3. Enabling Authorization Services

### 2.4. Build, Deploy, and Test Your Application

#### 2.4.1. Obtaining the Adapter Configuration

#### 2.4.2. Building and Deploying the Application

#### 2.4.3. Testing the Application

#### 2.4.4. Next Steps

### 2.5. Authorization Quickstarts

## 3. Managing Resource Servers

### 3.1. Creating a Client Application

### 3.2. Enabling Authorization Services

- 3.2.1. Resource Server Settings
- 3.3. Default Configuration
  - 3.3.1. Changing the Default Configuration
- 3.4. Export and Import Authorization Configuration
  - 3.4.1. Exporting a Configuration File
  - 3.4.2. Importing a Configuration File
- 4. Managing Resources and Scopes
  - 4.1. Viewing Resources
  - 4.2. Creating Resources
    - 4.2.1. Resource Attributes
    - 4.2.2. Typed Resources
    - 4.2.3. Resource Owners
    - 4.2.4. Managing Resources Remotely
- 5. Managing Policies
  - 5.1. User-Based Policy
    - 5.1.1. Configuration
  - 5.2. Role-Based Policy
    - 5.2.1. Configuration
    - 5.2.2. Defining a Role as Required
  - 5.3. JavaScript-Based Policy
    - 5.3.1. Configuration
    - 5.3.2. Examples
  - 5.4. Rule-Based Policy
    - 5.4.1. Configuration
    - 5.4.2. Examples
  - 5.5. Time-Based Policy
    - 5.5.1. Configuration
  - 5.6. Aggregated Policy

- 5.6.1. Configuration
- 5.6.2. Decision Strategy for Aggregated Policies
- 5.7. Client-Based Policy
  - 5.7.1. Configuration
- 5.8. Group-Based Policy
  - 5.8.1. Configuration
  - 5.8.2. Extending Access to Child Groups
- 5.9. Positive and Negative Logic
- 5.10. Policy Evaluation API
  - 5.10.1. The Evaluation Context
- 6. Managing Permissions
  - 6.1. Creating Resource-Based Permissions
    - 6.1.1. Configuration
    - 6.1.2. Typed Resource Permission
  - 6.2. Creating Scope-Based Permissions
    - 6.2.1. Configuration
  - 6.3. Policy Decision Strategies
- 7. Evaluating and Testing Policies
  - 7.1. Providing Identity Information
  - 7.2. Providing Contextual Information
  - 7.3. Providing the Permissions
- 8. Authorization Services
  - 8.1. Discovering Authorization Services Endpoints and Metadata
  - 8.2. Obtaining Permissions
    - 8.2.1. Client Authentication Methods
    - 8.2.2. Pushing Claims
  - 8.3. User-Managed Access

8.3.1. Authorization Process

8.3.2. Submitting Permission Requests

8.3.3. Managing Access to Users Resources

## 8.4. Protection API

8.4.1. What is a PAT and How to Obtain It

8.4.2. Managing Resources

8.4.3. Managing Permission Requests

8.4.4. Managing Resource Permissions using the Policy API

## 8.5. Requesting Party Token

8.5.1. Introspecting a Requesting Party Token

8.5.2. Obtaining Information about an RPT

8.5.3. Do I Need to Invoke the Server Every Time I Want to Introspect an RPT?

## 8.6. Authorization Client Java API

8.6.1. Maven Dependency

8.6.2. Configuration

8.6.3. Creating the Authorization Client

8.6.4. Obtaining User Entitlements

8.6.5. Creating a Resource Using the Protection API

8.6.6. Introspecting an RPT

# 9. Policy Enforcers

## 9.1. Configuration

## 9.2. Claim Information Point

9.2.1. Obtaining information from the HTTP Request

9.2.2. Obtaining information from an External HTTP Service

9.2.3. Static Claims

9.2.4. Claim Information Provider SPI

9.3. Obtaining the Authorization Context

9.4. Using the AuthorizationContext to obtain an Authorization Client Instance

9.5. JavaScript Integration

9.5.1. Handling Authorization Responses from a UMA-Protected Resource Server

9.5.2. Obtaining Entitlements

9.5.3. Authorization Request

9.5.4. Obtaining the RPT

9.6. Setting Up TLS/HTTPS

---

# {authorizationguide!}

# 1. Overview

Keycloak supports fine-grained authorization policies and is able to combine different access control mechanisms such as:

- **Attribute-based access control (ABAC)**
- **Role-based access control (RBAC)**
- **User-based access control (UBAC)**
- **Context-based access control (CBAC)**
- **Rule-based access control**
  - Using JavaScript
  - Using JBoss Drools
- **Time-based access control**
- **Support for custom access control mechanisms (ACMs) through a Policy Provider Service Provider Interface (SPI)**

Keycloak is based on a set of administrative UIs and a RESTful API, and provides the necessary means to create permissions for your protected resources and scopes, associate those permissions with authorization policies, and enforce authorization decisions in your applications and services.

Resource servers (applications or services serving protected resources) usually rely on some kind of information to decide if access should be granted to a protected resource. For RESTful-based resource servers,

that information is usually obtained from a security token, usually sent as a bearer token on every request to the server. For web applications that rely on a session to authenticate users, that information is usually stored in a user's session and retrieved from there for each request.

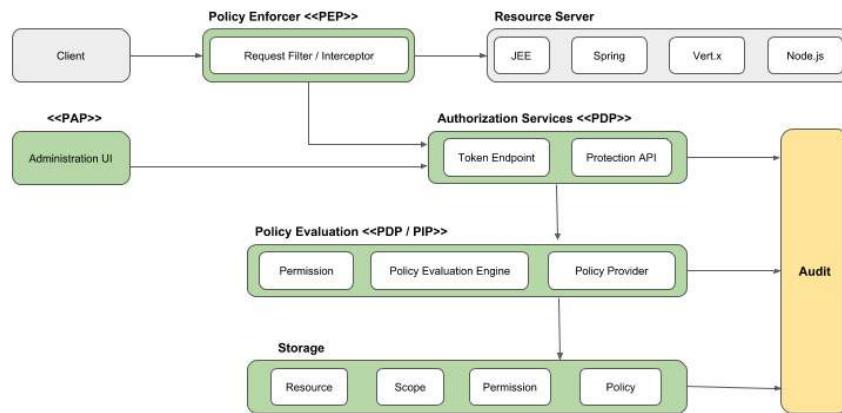
Frequently, resource servers only perform authorization decisions based on role-based access control (RBAC), where the roles granted to the user trying to access protected resources are checked against the roles mapped to these same resources. While roles are very useful and used by applications, they also have a few limitations:

- Resources and roles are tightly coupled and changes to roles (such as adding, removing, or changing an access context) can impact multiple resources
- Changes to your security requirements can imply deep changes to application code to reflect these changes
- Depending on your application size, role management might become difficult and error-prone
- It is not the most flexible access control mechanism. Roles do not represent who you are and lack contextual information. If you have been granted a role, you have at least some access.

Considering that today we need to consider heterogeneous environments where users are distributed across different regions, with different local policies, using different devices, and with a high demand for information sharing, KeyCloak Authorization Services can help you improve the authorization capabilities of your applications and services by providing:

- Resource protection using fine-grained authorization policies and different access control mechanisms
- Centralized Resource, Permission, and Policy Management
- Centralized Policy Decision Point
- REST security based on a set of REST-based authorization services
- Authorization workflows and User-Managed Access
- The infrastructure to help avoid code replication across projects (and redeloys) and quickly adapt to changes in your security requirements.

## 1.1. Architecture



From a design perspective, Authorization Services is based on a well-defined set of authorization patterns providing these capabilities:

- **Policy Administration Point (PAP)**

Provides a set of UIs based on the KeyCloak Administration Conso-

le to manage resource servers, resources, scopes, permissions, and policies. Part of this is also accomplished remotely through the use of the [Protection API](#).

- **Policy Decision Point (PDP)**

Provides a distributable policy decision point to where authorization requests are sent and policies are evaluated accordingly with the permissions being requested. For more information, see [Obtaining Permissions](#).

- **Policy Enforcement Point (PEP)**

Provides implementations for different environments to actually enforce authorization decisions at the resource server side. Keycloak provides some built-in [Policy Enforcers](#).

- **Policy Information Point (PIP)**

Being based on Keycloak Authentication Server, you can obtain attributes from identities and runtime environment during the evaluation of authorization policies.

### 1.1.1. The Authorization Process

Three main processes define the necessary steps to understand how to use Keycloak to enable fine-grained authorization to your applications:

- **Resource Management**
- **Permission and Policy Management**
- **Policy Enforcement**

### Resource Management

**Resource Management** involves all the necessary steps to define what is being protected.



First, you need to specify Keycloak what are you looking to protect, which usually represents a web application or a set of one or more services. For more information on resource servers see [Terminology](#).

Resource servers are managed using the Keycloak Administration Console. There you can enable any registered client application as a resource server and start managing the resources and scopes you want to protect.



A resource can be a web page, a RESTful resource, a file in your file system, an EJB, and so on. They can represent a group of resources (just like a Class in Java) or they can represent a single and specific resource.

For instance, you might have a *Bank Account* resource that represents all banking accounts and use it to define the authorization policies that are common to all banking accounts. However, you might want to define specific policies for *Alice Account* (a resource instance that belongs to a customer), where only the owner is allowed to access some information or perform an operation.

Resources can be managed using the Keycloak Administration Console or the [Protection API](#). In the latter case, resource servers are able to manage their resources remotely.

Scopes usually represent the actions that can be performed on a resource, but they are not limited to that. You can also use scopes to represent one or more attributes within a resource.

## Permission and Policy Management

Once you have defined your resource server and all the resources you want to protect, you must set up permissions and policies.

This process involves all the necessary steps to actually define the security and access requirements that govern your resources.



Policies define the conditions that must be satisfied to access or perform operations on something (resource or scope), but they are not tied to what they are protecting. They are generic and can be reused to build permissions or even more complex policies.

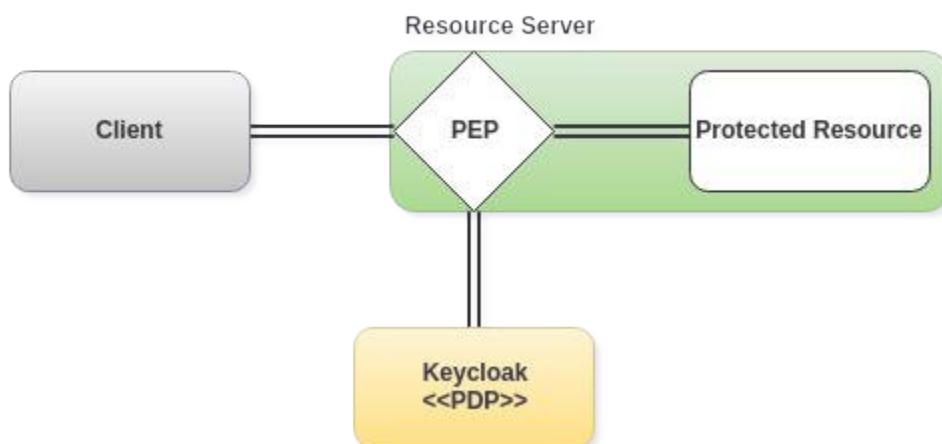
For instance,to allow access to a group of resources only for users granted with a role "User Premium", you can use RBAC (Role-based Access Control).

KeyCloak provides a few built-in policy types (and their respective policy providers) covering the most common access control mechanisms. You can even create policies based on rules written using JavaScript or JBoss Drools.

Once you have your policies defined, you can start defining your permissions. Permissions are coupled with the resource they are protecting. Here you specify what you want to protect (resource or scope) and the policies that must be satisfied to grant or deny permission.

## Policy Enforcement

**Policy Enforcement** involves the necessary steps to actually enforce authorization decisions to a resource server. This is achieved by enabling a **Policy Enforcement Point** or PEP at the resource server that is capable of communicating with the authorization server, ask for authorization data and control access to protected resources based on the decisions and permissions returned by the server.



KeyCloak provides some built-in [Policy Enforcers](#) implementations that you can use to protect your applications depending on the platform they are running on.

### 1.1.2. Authorization Services

Authorization services consist of the following RESTful endpoints:

- **Token Endpoint**
- **Resource Management Endpoint**
- **Permission Management Endpoint**

Each of these services provides a specific API covering the different steps involved in the authorization process.

#### Token Endpoint

OAuth2 clients (such as front end applications) can obtain access tokens from the server using the token endpoint and use these same tokens to access resources protected by a resource server (such as back end services). In the same way, KeyCloak Authorization Services provide extensions to OAuth2 to allow access tokens to be issued based on the processing of all policies associated with the resource(s) or scope(s) being requested. This means that resource servers can enforce access to their protected resources based on the permissions granted by the server and held by an access token. In KeyCloak Authorization Services the access token with permissions is called a Requesting Party Token or RPT for short.

For more information, see [Obtaining Permissions](#).

## Protection API

The **Protection API** is a set of [UMA-compliant](#) endpoint-providing operations for resource servers to help them manage their resources, scopes, permissions, and policies associated with them. Only resource servers are allowed to access this API, which also requires a **uma\_protection** scope.

The operations provided by the Protection API can be organized in two main groups:

- **Resource Management**

- Create Resource
- Delete Resource
- Find by Id
- Query

- **Permission Management**

- Issue Permission Tickets

By default, Remote Resource Management is enabled. You can change that using the Keycloak Administration Console and only allow resource management through the console.

When using the UMA protocol, the issuance of Permission Tickets by the Protection API is an important part of the whole authorization process. As described in a subsequent section, they represent the permissi-

ons being requested by the client and that are sent to the server to obtain a final token with all permissions granted during the evaluation of the permissions and policies associated with the resources and scopes being requested.

For more information, see [Protection API](#).

## 1.2. Terminology

Before going further, it is important to understand these terms and concepts introduced by KeyCloak Authorization Services.

### 1.2.1. Resource Server

Per OAuth2 terminology, a resource server is the server hosting the protected resources and capable of accepting and responding to protected resource requests.

Resource servers usually rely on some kind of information to decide whether access to a protected resource should be granted. For RESTful-based resource servers, that information is usually carried in a security token, typically sent as a bearer token along with every request to the server. Web applications that rely on a session to authenticate users usually store that information in the user's session and retrieve it from there for each request.

In KeyCloak, any **confidential** client application can act as a resource server. This client's resources and their respective scopes are protected and governed by a set of authorization policies.

### 1.2.2. Resource

A resource is part of the assets of an application and the organization. It can be a set of one or more endpoints, a classic web resource such as an HTML page, and so on. In authorization policy terminology, a resource is the *object* being protected.

Every resource has a unique identifier that can represent a single resource or a set of resources. For instance, you can manage a *Banking Account Resource* that represents and defines a set of authorization policies for all banking accounts. But you can also have a different resource named *Alice's Banking Account*, which represents a single resource owned by a single customer, which can have its own set of authorization policies.

### 1.2.3. Scope

A resource's scope is a bounded extent of access that is possible to perform on a resource. In authorization policy terminology, a scope is one of the potentially many *verbs* that can logically apply to a resource.

It usually indicates what can be done with a given resource. Examples of scopes are view, edit, delete, and so on. However, scope can also be related to specific information provided by a resource. In this case, you can have a project resource and a cost scope, where the cost scope is used to define specific policies and permissions for users to access a project's cost.

### 1.2.4. Permission

Consider this simple and very common permission:

A permission associates the object being protected with the policies that must be evaluated to determine whether access is granted.

- **X CAN DO Y ON RESOURCE Z**
  - where ...
    - **X** represents one or more users, roles, or groups, or a combination of them. You can also use claims and context here.
    - **Y** represents an action to be performed, for example, write, view, and so on.
    - **Z** represents a protected resource, for example, "/accounts".

KeyCloak provides a rich platform for building a range of permission strategies ranging from simple to very complex, rule-based dynamic permissions. It provides flexibility and helps to:

- Reduce code refactoring and permission management costs
- Support a more flexible security model, helping you to easily adapt to changes in your security requirements
- Make changes at runtime; applications are only concerned about the resources and scopes being protected and not how they are protected.

### 1.2.5. Policy

A policy defines the conditions that must be satisfied to grant access to an object. Unlike permissions, you do not specify the object being protected but rather the conditions that must be satisfied for access to a given object (for example, resource, scope, or both). Policies are strongly related to the different access control mechanisms (ACMs) that you can use to protect your resources. With policies, you can implement strategies for attribute-based access control (ABAC), role-based access

control (RBAC), context-based access control, or any combination of these.

Keycloak leverages the concept of policies and how you define them by providing the concept of aggregated policies, where you can build a "policy of policies" and still control the behavior of the evaluation. Instead of writing one large policy with all the conditions that must be satisfied for access to a given resource, the policies implementation in Keycloak Authorization Services follows the divide-and-conquer technique. That is, you can create individual policies, then reuse them with different permissions and build more complex policies by combining individual policies.

### 1.2.6. Policy Provider

Policy providers are implementations of specific policy types. Keycloak provides built-in policies, backed by their corresponding policy providers, and you can create your own policy types to support your specific requirements.

Keycloak provides a SPI (Service Provider Interface) that you can use to plug in your own policy provider implementations.

### 1.2.7. Permission Ticket

A permission ticket is a special type of token defined by the User-Managed Access (UMA) specification that provides an opaque structure whose form is determined by the authorization server. This structure represents the resources and/or scopes being requested by a client, the access context, as well as the policies that must be applied to a request for authorization data (requesting party token [RPT]).

In UMA, permission tickets are crucial to support person-to-person sharing and also person-to-organization sharing. Using permission tickets for authorization workflows enables a range of scenarios from simple to complex, where resource owners and resource servers have complete control over their resources based on fine-grained policies that govern the access to these resources.

In the UMA workflow, permission tickets are issued by the authorization server to a resource server, which returns the permission ticket to the client trying to access a protected resource. Once the client receives the ticket, it can make a request for an RPT (a final token holding authorization data) by sending the ticket back to the authorization server.

For more information on permission tickets, see [User-Managed Access](#) and the [UMA](#) specification.

## 2. Getting Started

Before you can use this tutorial, you need to complete the installation of KeyCloak and create the initial admin user as shown in the [{gettingstarted\\_name}](#) tutorial. There is one caveat to this. You have to run a separate WildFly instance on the same machine as KeyCloak Server. This separate instance will run your Java Servlet application. Because of this you will have to run the KeyCloak under a different port so that there are no port conflicts when running on the same machine. Use the `jboss.socket.binding.port-offset` system property on the command line. The value of this property is a number that will be added to the base value of every port opened by KeyCloak Server.

To boot KeyCloak Server:

### *Linux/Unix*

```
$ .../bin/standalone.sh -Djboss.socket.binding.port-offset=100
```

### *Windows*

```
> ...\\bin\\standalone.bat -Djboss.socket.binding.port-offset=100
```

For more details about how to install and configure a WildFly, please follow the steps on the [{adapterguide!}](#) tutorial.

After installing and booting both servers you should be able to access

Keycloak Admin Console at <http://localhost:8180/auth/admin/> and also the WildFly instance at <http://localhost:8080>.

## 2.1. Securing a Servlet Application

The purpose of this getting started guide is to get you up and running as quickly as possible so that you can experiment with and test various authorization features provided by Keycloak. This quick tour relies heavily on the default database and server configurations and does not cover complex deployment options. For more information on features or configuration options, see the appropriate sections in this documentation.

This guide explains key concepts about Keycloak Authorization Services:

- Enabling fine-grained authorization for a client application
- Configuring a client application to be a resource server, with protected resources
- Defining permissions and authorization policies to govern access to protected resources
- Enabling policy enforcement in your applications.

## 2.2. Creating a Realm and a User

The first step in this tutorial is to create a realm and a user in that realm. Then, within the realm we will create a single client application, which then becomes a [resource server](#) for which you need to enable authorization services.

To create a realm and a user complete the following steps:

1. Create a realm with a name **hello-world-authz**. Once created, a page similar to the following is displayed:

*Realm hello-world-authz*

2. Create a user for your newly created realm. Click **Users**. The user list page opens.
3. On the right side of the empty user list, click **Add User**.
4. To create a new user, complete the **Username**, **Email**, **First Name**, and **Last Name** fields. Click the **User Enabled** switch to **On**, and then click **Save**.

*Add User*

5. Set a password for the user by clicking the **Credentials** tab.

*Set User Password*

6. Complete the **New Password** and **Password Confirmation** fields with a password and click the **Temporary** switch to **OFF**.
7. Click **Reset Password** to set the user's password.

## 2.3. Enabling Authorization Services

You can enable authorization services in an existing client application configured to use the OpenID Connect Protocol. You can also create a new client.

To create a new client, complete the following steps:

1. Click **Clients** to start creating a new client application and fill in the **Client ID**, **Client Protocol**, and **Root URL** fields.

*Create Client Application*

2. Click **Save**. The Client Details page is displayed.

*Client Details*

3. On the Client Details page, click the **Authorization Enabled** switch to **ON**, and then click **Save**. A new **Authorization** tab is displayed for the client.
4. Click the **Authorization** tab and an Authorization Settings page similar to the following is displayed:

*Authorization Settings*

When you enable authorization services for a client application, Keycloak automatically creates several [default settings](#) for your client authorization configuration.

For more information about authorization configuration, see [Enabling Authorization Services](#).

## 2.4. Build, Deploy, and Test Your Application

Now that the **app-authz-vanilla** resource server (or client) is properly configured and authorization services are enabled, it can be deployed to the server.

The project and code for the application you are going to deploy is available in `{quickstartRepo_name}`. You will need the following installed on your machine and available in your PATH before you can continue:

- Java JDK 8
- Apache Maven 3.1.1 or higher
- Git

Follow these steps to download the code.

### *Clone Project*

```
$ git clone {quickstartRepo_link}
```

The application we are about to build and deploy is located at

```
$ cd {quickstartRepo_dir}/app-authz-jee-vanilla
```

#### **2.4.1. Obtaining the Adapter Configuration**

You must first obtain the adapter configuration before building and deploying the application.

To obtain the adapter configuration from the Keycloak Administration Console, complete the following steps.

1. Click **Clients**. In the client listing, click the **app-authz-vanilla** client application. The Client Details page opens.

### *Client Details*

2. Click the **Installation** tab. From the Format Option dropdown list, select **Keycloak OIDC JSON**. The adapter configuration is displayed in JSON format. Click **Download**.

#### *Adapter Configuration*

3. Move the file `keycloak.json` to the `app-authz-jee-vanilla/config` directory.
4. (optional) By default, the policy enforcer responds with a `403` status code when the user lacks permission to access protected resources on the resource server. However, you can also specify a redirection URL for unauthorized users. To specify a redirection URL, edit the `keycloak.json` file you updated in step 3 and replace the `policy-enforcer` configuration with the following:

```
"policy-enforcer": {  
    "on-deny-redirect-to" : "/app-authz-vanilla/error.jsp"  
}
```

This change specifies to the policy enforcer to redirect users to a `/app-authz-vanilla/error.jsp` page if a user does not have the necessary permissions to access a protected resource, rather than an unhelpful `403 Unauthorized` message.

#### **2.4.2. Building and Deploying the Application**

To build and deploy the application execute the following command:

```
$ cd redhat-sso-quickstarts/app-authz-jee-vanilla
```

```
$ mvn clean package wildfly:deploy
```

### 2.4.3. Testing the Application

If your application was successfully deployed you can access it at <http://localhost:8080/app-authz-vanilla>. The Keycloak Login page opens.

#### *Login Page*



Log in as **alice** using the password you specified for that user. After authenticating, the following page is displayed:

#### *Hello World Authz Main Page*



The [default settings](#) defined by Keycloak when you enable authorization services for a client application provide a simple policy that always grants access to the resources protected by this policy.

You can start by changing the default permissions and policies and test how your application responds, or even create new policies using the different [policy types](#) provided by Keycloak.

There are a plenty of things you can do now to test this application. For example, you can change the default policy by clicking the Authorization tab for the client, then `Policies` tab, then click on `Default Policy` in the list to allow you to change it as follows:

```
// The default value is $evaluation.grant(),  
// let's see what happens when we change it to $evaluati-
```

```
on.deny()  
$evaluation.deny();
```

Now, log out of the demo application and log in again. You can no longer access the application.



Let's fix that now, but instead of changing the `Default Policy` code we are going to change the `Logic` to `Negative` using the dropdown list below the policy code text area. That re-enables access to the application as we are negating the result of that policy, which is by default denying all requests for access. Again, before testing this change, be sure to log out and log in again.

#### 2.4.4. Next Steps

There are additional things you can do, such as:

- Create a scope, define a policy and permission for it, and test it on the application side. Can the user perform an action (or anything else represented by the scope you created)?
- Create different types of policies such as [rule-based](#), and associate these policies with the `Default Permission`.
- Apply multiple policies to the `Default Permission` and test the behavior. For example, combine multiple policies and change the `Decision Strategy` accordingly.
- For more information about how to view and test permissions inside your application see [Obtaining the Authorization Context](#).

## 2.5. Authorization Quickstarts

In addition to the **app-authz-jee-vanilla** quickstart that was used as a sample application in the previous section, the `{quickstartRepo_name}` contains other applications that make use of the authorization services described in this documentation.

The authorization quickstarts have been designed so that authorization services are displayed in different scenarios and using different technologies and integrations. It is not meant as a comprehensive set of all the possible use cases involving authorization but they should provide a starting point for users interested in understanding how the authorization services can be used in their own applications.

Each quickstart has a `README` file with instructions on how to build, deploy, and test the sample application. The following table provides a brief description of the available authorization quickstarts:

*Table 1. Authorization Quickstarts*

Name	Description
<a href="#">app-authz-jee-servlet</a>	Demonstrates how to enable fine-grained authorization to a Java EE application in order to protect specific resources and build a dynamic menu based on the permissions obtained from a Keycloak Server.
<a href="#">app-authz-jee-vanilla</a>	Demonstrates how to enable fi-

	ne-grained authorization to a Java EE application and use the default authorization settings to protect all resources in the application.
<a href="#"><u>app-authz-rest-springboot</u></a>	Demonstrates how to protect a SpringBoot REST service using Keycloak Authorization Services.
<a href="#"><u>app-authz-springboot</u></a>	Demonstrates how to write a SpringBoot Web application where both authentication and authorization aspects are managed by Keycloak.
<a href="#"><u>app-authz-uma-photoz</u></a>	A simple application based on HTML5+AngularJS+JAX-RS that demonstrates how to enable User-Managed Access to your application and let users to manage permissions for their resources.

---

## 3. Managing Resource Servers

According to the OAuth2 specification, a resource server is a server hosting the protected resources and capable of accepting and responding to protected resource requests.

In Keycloak, resource servers are provided with a rich platform for enabling fine-grained authorization for their protected resources, where authorization decisions can be made based on different access control mechanisms.

Any client application can be configured to support fine-grained permissions. In doing so, you are conceptually turning the client application into a resource server.

### 3.1. Creating a Client Application

The first step to enable Keycloak Authorization Services is to create the client application that you want to turn into a resource server.

To create a client application, complete the following steps:

1. Click **Clients**.

*Clients*



2. On this page, click **Create**.

*Create Client*



3. Type the `Client ID` of the client. For example, *my-resource-server*.
4. Type the `Root URL` for your application. For example:

```
http://${host}:${port}/my-resource-server
```

5. Click **Save**. The client is created and the client Settings page opens. A page similar to the following is displayed:

### *Client Settings*

## 3.2. Enabling Authorization Services

To turn your OIDC Client Application into a resource server and enable fine-grained authorization, click the **Authorization Enabled** switch to **ON** and click **Save**.

### *Enabling Authorization Services*

A new Authorization tab is displayed for this client. Click the **Authorization** tab and a page similar to the following is displayed:

### *Resource Server Settings*

The Authorization tab contains additional sub-tabs covering the different steps that you must follow to actually protect your application's resources. Each tab is covered separately by a specific topic in this documentation. But here is a quick description about each one:

- **Settings**

General settings for your resource server. For more details about this page see the [Resource Server Settings](#) section.

- **Resource**

From this page, you can manage your application's [resources](#).

- **Authorization Scopes**

From this page, you can manage [scopes](#).

- **Policies**

From this page, you can manage [authorization policies](#) and define the conditions that must be met to grant a permission.

- **Permissions**

From this page, you can manage the [permissions](#) for your protected resources and scopes by linking them with the policies you created.

- **Evaluate**

From this page, you can [simulate authorization requests](#) and view the result of the evaluation of the permissions and authorization policies you have defined.

- **Export Settings**

From this page, you can [export](#) the authorization settings to a JSON file.

### 3.2.1. Resource Server Settings

On the Resource Server Settings page, you can configure the policy enforcement mode, allow remote resource management, and export the

authorization configuration settings.

- **Policy Enforcement Mode**

Specifies how policies are enforced when processing authorization requests sent to the server.

- **Enforcing**

(default mode) Requests are denied by default even when there is no policy associated with a given resource.

- **Permissive**

Requests are allowed even when there is no policy associated with a given resource.

- **Disabled**

Disables the evaluation of all policies and allows access to all resources.

- **Remote Resource Management**

Specifies whether resources can be managed remotely by the resource server. If false, resources can be managed only from the administration console.

### 3.3. Default Configuration

When you create a resource server, Keycloak creates a default configuration for your newly created resource server.

The default configuration consists of:

- A default protected resource representing all resources in your appli-

cation.

- A policy that always grants access to the resources protected by this policy.
- A permission that governs access to all resources based on the default policy.

The default protected resource is referred to as the **default resource** and you can view it if you navigate to the **Resources** tab.

### *Default Resource*



This resource defines a `Type`, namely `urn:my-resource-server:resources:default` and a `URI */*`. Here, the `URI` field defines a wildcard pattern that indicates to Keycloak that this resource represents all the paths in your application. In other words, when enabling [policy enforcement](#) for your application, all the permissions associated with the resource will be examined before granting access.

The `Type` mentioned previously defines a value that can be used to create [typed resource permissions](#) that must be applied to the default resource or any other resource you create using the same type.

The default policy is referred to as the **only from realm policy** and you can view it if you navigate to the **Policies** tab.

### *Default Policy*



This policy is a [JavaScript-based policy](#) defining a condition that always

grants access to the resources protected by this policy. If you click this policy you can see that it defines a rule as follows:

```
// by default, grants any permission associated with this  
policy  
$evaluation.grant();
```

Lastly, the default permission is referred to as the **default permission** and you can view it if you navigate to the **Permissions** tab.

### *Default Permission*

This permission is a [resource-based permission](#), defining a set of one or more policies that are applied to all resources with a given type.

#### 3.3.1. Changing the Default Configuration

You can change the default configuration by removing the default resource, policy, or permission definitions and creating your own.

The default resource is created with an **URI** that maps to any resource or path in your application using a `/*` pattern. Before creating your own resources, permissions and policies, make sure the default configuration doesn't conflict with your own settings.

The default configuration defines a resource that maps to all paths in your application. If you are about to write permissions to your own resources, be sure to remove the **Default Resource** or change its `URIS` fields to a more specific paths in your application. Otherwise, the

policy associated with the default resource (which by default always grants access) will allow Keycloak to grant access to any protected resource.

## 3.4. Export and Import Authorization Configuration

The configuration settings for a resource server (or client) can be exported and downloaded. You can also import an existing configuration file for a resource server. Importing and exporting a configuration file is helpful when you want to create an initial configuration for a resource server or to update an existing configuration. The configuration file contains definitions for:

- Protected resources and scopes
- Policies
- Permissions

### 3.4.1. Exporting a Configuration File

To export a configuration file, complete the following steps:

1. Navigate to the **Resource Server Settings** page.
2. Click the **Export Settings** tab.
3. On this page, click **Export**.

#### *Export Settings*



The configuration file is exported in JSON format and displayed in a text area, from which you can copy and paste. You can also click

**Download** to download the configuration file and save it.

### 3.4.2. Importing a Configuration File

To import a configuration file, complete the following steps:

1. Navigate to the **Resource Server Settings** page.

*Import Settings*



To import a configuration file for a resource server, click **Select file** to select a file containing the configuration you want to import.

---

## 4. Managing Resources and Scopes

Resource management is straightforward and generic. After creating a resource server, you can start creating the resources and scopes that you want to protect. Resources and scopes can be managed by navigating to the **Resource** and **Scope** tabs, respectively.

### 4.1. Viewing Resources

On the **Resource** page, you see a list of the resources associated with a resource server.

#### *Resources*



The resource list provides information about the protected resources, such as:

- Type
- URIS
- Owner
- Associated scopes, if any
- Associated permissions

From this list, you can also directly create a permission by clicking **Create Permission** for the resource for which you want to create the permission.

Before creating permissions for your resources, be sure you have already defined the policies that you want to associate with the permission.

## 4.2. Creating Resources

Creating a resource is straightforward and generic. Your main concern is the granularity of the resources you create. In other words, resources can be created to represent a set of one or more resources and the way you define them is crucial to managing permissions.

To create a new resource, click **Create** in the right upper corner of the resource listing.

### *Add Resource*

In Keycloak, a resource defines a small set of information that is common to different types of resources, such as:

- **Name**

A human-readable and unique string describing this resource.

- **Type**

A string uniquely identifying the type of a set of one or more resources. The type is a *string* used to group different resource instances.

For example, the default type for the default resource that is automatically created is `urn:resource-server-name:resources:default`

- **URIS**

URIS that provides the locations/addresses for the resource. For HTTP resources, the URIS are usually the relative paths used to serve these resources.

- **Scopes**

One or more scopes to associate with the resource.

#### 4.2.1. Resource Attributes

Resources may have attributes associated with them. These attributes can be used to provide additional information about a resource and to provide additional information to policies when evaluating permissions associated with a resource.

Each attribute is a key and value pair where the value can be a set of one or many strings. Multiple values can be defined for an attribute by separating each value with a comma.

#### 4.2.2. Typed Resources

The type field of a resource can be used to group different resources together, so they can be protected using a common set of permissions.

#### 4.2.3. Resource Owners

Resources also have an owner. By default, resources are owned by the resource server.

However, resources can also be associated with users, so you can create permissions based on the resource owner. For example, only the resource owner is allowed to delete or update a given resource.

#### 4.2.4. Managing Resources Remotely

Resource management is also exposed through the [Protection API](#) to allow resource servers to remotely manage their resources.

When using the Protection API, resource servers can be implemented to manage resources owned by their users. In this case, you can specify the user identifier to configure a resource as belonging to a specific user.

Keycloak provides resource servers complete control over their resources. In the future, we should be able to allow users to control their own resources as well as approve authorization requests and manage permissions, especially when using the UMA protocol.

---

## 5. Managing Policies

As mentioned previously, policies define the conditions that must be satisfied before granting access to an object.

You can view all policies associated with a resource server by clicking the **Policy** tab when editing a resource server.

### *Policies*



On this tab, you can view the list of previously created policies as well as create and edit a policy.

To create a new policy, in the upper right corner of the policy list, select a policy type from the `Create policy` dropdown list. Details about each policy type are described in this section.

### 5.1. User-Based Policy

You can use this type of policy to define conditions for your permissions where a set of one or more users is permitted to access an object.

To create a new user-based policy, select **User** in the dropdown list in the upper right corner of the policy listing.

#### *Add a User-Based Policy*



##### 5.1.1. Configuration

- **Name**

A human-readable and unique string identifying the policy. A best practice is to use names that are closely related to your business and security requirements, so you can identify them more easily.

- **Description**

A string containing details about this policy.

- **Users**

Specifies which users are given access by this policy.

- **Logic**

The [Logic](#) of this policy to apply after the other conditions have been evaluated.

## 5.2. Role-Based Policy

You can use this type of policy to define conditions for your permissions where a set of one or more roles is permitted to access an object.

By default, roles added to this policy are not specified as required and the policy will grant access if the user requesting access has been granted any of these roles. However, you can specify a specific role as [required](#) if you want to enforce a specific role. You can also combine required and non-required roles, regardless of whether they are realm or client roles.

Role policies can be useful when you need more restricted role-based access control (RBAC), where specific roles must be enforced to grant access to an object. For instance, you can enforce that a user must con-

sent to allowing a client application (which is acting on the user's behalf) to access the user's resources. You can use Keycloak Client Scope Mapping to enable consent pages or even enforce clients to explicitly provide a scope when obtaining access tokens from a Keycloak server.

To create a new role-based policy, select **Role** in the dropdown list in the upper right corner of the policy listing.

### *Add Role-Based Policy*

#### 5.2.1. Configuration

- **Name**

A human-readable and unique string describing the policy. A best practice is to use names that are closely related to your business and security requirements, so you can identify them more easily.

- **Description**

A string containing details about this policy.

- **Realm Roles**

Specifies which **realm** roles are permitted by this policy.

- **Client Roles**

Specifies which **client** roles are permitted by this policy. To enable this field must first select a **Client**.

- **Logic**

The [Logic](#) of this policy to apply after the other conditions have been evaluated.

### 5.2.2. Defining a Role as Required

When creating a role-based policy, you can specify a specific role as **Required**. When you do that, the policy will grant access only if the user requesting access has been granted **all** the **required** roles. Both realm and client roles can be configured as such.

#### *Example of Required Role*

To specify a role as required, select the **Required** checkbox for the role you want to configure as required.

Required roles can be useful when your policy defines multiple roles but only a subset of them are mandatory. In this case, you can combine realm and client roles to enable an even more fine-grained role-based access control (RBAC) model for your application. For example, you can have policies specific for a client and require a specific client role associated with that client. Or you can enforce that access is granted only in the presence of a specific realm role. You can also combine both approaches within the same policy.

## 5.3. JavaScript-Based Policy

You can use this type of policy to define conditions for your permissions using JavaScript. It is one of the rule-based policy types supported by Keycloak, and provides flexibility to write any policy based on the [Evaluation API](#).

To create a new JavaScript-based policy, select **JavaScript** in the drop-down list in the upper right corner of the policy listing.

## Add JavaScript Policy

### 5.3.1. Configuration

- **Name**

A human-readable and unique string describing the policy. A best practice is to use names that are closely related to your business and security requirements, so you can identify them more easily.

- **Description**

A string containing details about this policy.

- **Code**

The JavaScript code providing the conditions for this policy.

- **Logic**

The [Logic](#) of this policy to apply after the other conditions have been evaluated.

### 5.3.2. Examples

#### Checking for attributes from the evaluation context

Here is a simple example of a JavaScript-based policy that uses attribute-based access control (ABAC) to define a condition based on an attribute obtained from the execution context:

```
var context = $evaluation.getContext();
var contextAttributes = context.getAttributes();

if (contextAttributes.containsValue('kc.client.network.
ip_address', '127.0.0.1')) {
    $evaluation.grant();
```

```
}
```

## Checking for attributes from the current identity

Here is a simple example of a JavaScript-based policy that uses attribute-based access control (ABAC) to define a condition based on an attribute obtained associated with the current identity:

```
var context = $evaluation.getContext();
var identity = context.getIdentity();
var attributes = identity.getAttributes();
var email = attributes.getValue('email').asString(0);

if (email.endsWith('@keycloak.org')) {
    $evaluation.grant();
}
```

Where these attributes are mapped from whatever claim is defined in the token that was used in the authorization request.

## Checking for roles granted to the current identity

You can also use Role-Based Access Control (RBAC) in your policies. In the example below, we check if a user is granted with a `keycloak_user realm` role:

```
var context = $evaluation.getContext();
var identity = context.getIdentity();

if (identity.hasRealmRole('keycloak_user')) {
    $evaluation.grant();
}
```

Or you can check if a user is granted with a `my-client-role client` role, where `my-client` is the client id of the client application:

```
var context = $evaluation.getContext();
var identity = context.getIdentity();

if (identity.hasClientRole('my-client', 'my-client-role'))
{
    $evaluation.grant();
}
```

## Checking for roles granted to an user

To check for realm roles granted to an user:

```
var realm = $evaluation.getRealm();

if (realm.isUserInRealmRole('marta', 'role-a')) {
    $evaluation.grant();
}
```

Or for client roles granted to an user:

```
var realm = $evaluation.getRealm();

if (realm.isUserInClientRole('marta', 'my-client', 'some-client-role')) {
    $evaluation.grant();
}
```

## Checking for roles granted to a group

To check for realm roles granted to a group:

```
var realm = $evaluation.getRealm();

if (realm.isGroupInRole('/Group A/Group D', 'role-a')) {
    $evaluation.grant();
}
```

## Pushing arbitrary claims to the resource server

To push arbitrary claims to the resource server in order to provide additional information on how permissions should be enforced:

```
var permission = $evaluation.getPermission();

// decide if permission should be granted

if (granted) {
    permission.addClaim('claim-a', 'claim-a');
    permission.addClaim('claim-a', 'claim-a1');
    permission.addClaim('claim-b', 'claim-b');
}
```

## Checking for group membership

```
var realm = $evaluation.getRealm();

if (realm.isUserInGroup('marta', '/Group A/Group B')) {
    $evaluation.grant();
}
```

## Mixing different access control mechanisms

You can also use a combination of several access control mechanisms. The example below shows how roles(RBAC) and claims/attributes(ABAC) checks can be used within the same policy. In this case we check if user is granted with `admin` role or has an e-mail from `keycloak.org` domain:

```
var context = $evaluation.getContext();
var identity = context.getIdentity();
var attributes = identity.getAttributes();
var email = attributes.getValue('email').asString(0);

if (identity.hasRealmRole('admin') || email.endsWith('@')
```

```
keycloak.org')) {  
    $evaluation.grant();  
}
```

When writing your own rules, keep in mind that the **\$evaluation** object is an object implementing **org.keycloak.authorization.policy.evaluation.Evaluation**. For more information about what you can access from this interface, see the [Evaluation API](#).

## 5.4. Rule-Based Policy

With this type of policy you can define conditions for your permissions using [Drools](#), which is a rule evaluation environment. It is one of the *Rule-Based* policy types supported by KeyCloak, and provides flexibility to write any policy based on the [Evaluation API](#).

To create a new Rule-based policy, in the dropdown list in the right upper corner of the policy listing, select **Rule**.

### Add Rule Policy

#### 5.4.1. Configuration

- **Name**

A human-readable and unique string describing the policy. We strongly suggest that you use names that are closely related with your business and security requirements, so you can identify them more easily and also know what they actually mean.

- **Description**

A string with more details about this policy.

- **Policy Maven Artifact**

A Maven groupId-artifactId-version (GAV) pointing to an artifact where the rules are defined. Once you have provided the GAV, you can click **Resolve** to load both **Module** and **Session** fields.

- **Group Id**

The groupId of the artifact.

- **Artifact Id**

The artifactId of the artifact.

- **Version**

The version of the artifact.

- **Module**

The module used by this policy. You must provide a module to select a specific session from which rules will be loaded.

- **Session**

The session used by this policy. The session provides all the rules to evaluate when processing the policy.

- **Update Period**

Specifies an interval for scanning for artifact updates.

- **Logic**

The Logic of this policy to apply after the other conditions have

been evaluated.

#### 5.4.2. Examples

Here is a simple example of a Drools-based policy that uses attribute-based access control (ABAC) to define a condition that evaluates to a GRANT only if the authenticated user is the owner of the requested resource:

```
import org.keycloak.authorization.policy.evaluation.Evaluation;
rule "Authorize Resource Owner"
    dialect "mvel"
    when
        $evaluation : Evaluation(
            $identity: context.identity,
            $permission: permission,
            $permission.resource != null && $permission.
resource.owner.equals($identity.id)
        )
    then
        $evaluation.grant();
end
```

You can even use another variant of ABAC to obtain attributes from the identity and define a condition accordingly:

```
import org.keycloak.authorization.policy.evaluation.Evaluation;
rule "Authorize Using Identity Information"
    dialect "mvel"
    when
        $evaluation : Evaluation(
            $identity: context.identity,
            identity.attributes.containsValue("someAttribute", "you_can_access")
        )
    then
```

```
    $evaluation.grant();  
end
```

For more information about what you can access from the `org.keycloak.authorization.policy.evaluation.Evaluation` interface, see [Evaluation API](#).

## 5.5. Time-Based Policy

You can use this type of policy to define time conditions for your permissions.

To create a new time-based policy, select **Time** in the dropdown list in the upper right corner of the policy listing.

### *Add Time Policy*

#### 5.5.1. Configuration

- **Name**

A human-readable and unique string describing the policy. A best practice is to use names that are closely related to your business and security requirements, so you can identify them more easily.

- **Description**

A string containing details about this policy.

- **Not Before**

Defines the time before which access must **not** be granted. Permission is granted only if the current date/time is later than or equal to this value.

- **Not On or After**

Defines the time after which access must **not** be granted. Permission is granted only if the current date/time is earlier than or equal to this value.

- **Day of Month**

Defines the day of month that access must be granted. You can also specify a range of dates. In this case, permission is granted only if the current day of the month is between or equal to the two values specified.

- **Month**

Defines the month that access must be granted. You can also specify a range of months. In this case, permission is granted only if the current month is between or equal to the two values specified.

- **Year**

Defines the year that access must be granted. You can also specify a range of years. In this case, permission is granted only if the current year is between or equal to the two values specified.

- **Hour**

Defines the hour that access must be granted. You can also specify a range of hours. In this case, permission is granted only if current hour is between or equal to the two values specified.

- **Minute**

Defines the minute that access must be granted. You can also specify a range of minutes. In this case, permission is granted only if the

current minute is between or equal to the two values specified.

- **Logic**

The [Logic](#) of this policy to apply after the other conditions have been evaluated.

Access is only granted if all conditions are satisfied. Keycloak will perform an *AND* based on the outcome of each condition.

## 5.6. Aggregated Policy

As mentioned previously, Keycloak allows you to build a policy of policies, a concept referred to as policy aggregation. You can use policy aggregation to reuse existing policies to build more complex ones and keep your permissions even more decoupled from the policies that are evaluated during the processing of authorization requests.

To create a new aggregated policy, select **Aggregated** in the dropdown list located in the right upper corner of the policy listing.

### Add an Aggregated Policy



Let's suppose you have a resource called *Confidential Resource* that can be accessed only by users from the `keycloak.org` domain and from a certain range of IP addresses. You can create a single policy with both conditions. However, you want to reuse the domain part of this policy to apply to permissions that operates regardless of the originating network.

You can create separate policies for both domain and network conditions and create a third policy based on the combination of these two po-

licies. With an aggregated policy, you can freely combine other policies and then apply the new aggregated policy to any permission you want.

When creating aggregated policies, be mindful that you are not introducing a circular reference or dependency between policies. If a circular dependency is detected, you cannot create or update the policy.

### 5.6.1. Configuration

- **Name**

A human-readable and unique string describing the policy. We strongly suggest that you use names that are closely related with your business and security requirements, so you can identify them more easily and also know what they mean.

- **Description**

A string with more details about this policy.

- **Apply Policy**

Defines a set of one or more policies to associate with the aggregated policy. To associate a policy you can either select an existing policy or create a new one by selecting the type of the policy you want to create.

- **Decision Strategy**

The decision strategy for this permission.

- **Logic**

The [Logic](#) of this policy to apply after the other conditions have

been evaluated.

### 5.6.2. Decision Strategy for Aggregated Policies

When creating aggregated policies, you can also define the decision strategy that will be used to determine the final decision based on the outcome from each policy.

- **Unanimous**

The default strategy if none is provided. In this case, *all* policies must evaluate to a positive decision for the final decision to be also positive.

- **Affirmative**

In this case, *at least one* policy must evaluate to a positive decision in order for the final decision to be also positive.

- **Consensus**

In this case, the number of positive decisions must be greater than the number of negative decisions. If the number of positive and negative decisions is the same, the final decision will be negative.

## 5.7. Client-Based Policy

You can use this type of policy to define conditions for your permissions where a set of one or more clients is permitted to access an object.

To create a new client-based policy, select **Client** in the dropdown list in the upper right corner of the policy listing.

*Add a Client-Based Policy*

### Add Client Policy

Name \*

Description

Clients \*

clientId	Actions
photoz-html5-client	<input type="button" value="Remove"/>

Logic

## 5.7.1. Configuration

- **Name**

A human-readable and unique string identifying the policy. A best practice is to use names that are closely related to your business and security requirements, so you can identify them more easily.

- **Description**

A string containing details about this policy.

- **Clients**

Specifies which clients are given access by this policy.

- **Logic**

The Logic of this policy to apply after the other conditions have been evaluated.

## 5.8. Group-Based Policy

You can use this type of policy to define conditions for your permissions where a set of one or more groups (and their hierarchies) is permitted to access an object.

To create a new group-based policy, select **Group** in the dropdown list in the upper right corner of the policy listing.

### *Add Group-Based Policy*

#### 5.8.1. Configuration

- **Name**

A human-readable and unique string describing the policy. A best practice is to use names that are closely related to your business and security requirements, so you can identify them more easily.

- **Description**

A string containing details about this policy.

- **Groups Claim**

Specifies the name of the claim in the token holding the group names and/or paths. Usually, authorization requests are processed based on an ID Token or Access Token previously issued to a client acting on behalf of some user. If defined, the token must include a claim from where this policy is going to obtain the groups the user is a member of. If not defined, user's groups are obtained from your realm configuration.

- **Groups**

Allows you to select the groups that should be enforced by this policy when evaluating permissions. After adding a group, you can extend access to children of the group by marking the checkbox **Extend to Children**. If left unmarked, access restrictions only applies

to the selected group.

- **Logic**

The [Logic](#) of this policy to apply after the other conditions have been evaluated.

### 5.8.2. Extending Access to Child Groups

By default, when you add a group to this policy, access restrictions will only apply to members of the selected group.

Under some circumstances, it might be necessary to allow access not only to the group itself but to any child group in the hierarchy. For any group added you can mark a checkbox **Extend to Children** in order to extend access to child groups.

#### *Extending Access to Child Groups*



In the example above, the policy is granting access for any user member of **IT** or any of its children.

## 5.9. Positive and Negative Logic

Policies can be configured with positive or negative logic. Briefly, you can use this option to define whether the policy result should be kept as it is or be negated.

For example, suppose you want to create a policy where only users **not** granted with a specific role should be given access. In this case, you can create a role-based policy using that role and set its **Logic** field to **Negative**. If you keep **Positive**, which is the default behavior, the policy

result will be kept as it is.

## 5.10. Policy Evaluation API

When writing rule-based policies using JavaScript or JBoss Drools, Keycloak provides an Evaluation API that provides useful information to help determine whether a permission should be granted.

This API consists of a few interfaces that provide you access to information, such as

- The permission being evaluated, representing both the resource and scopes being requested.
- The attributes associated with the resource being requested
- Runtime environment and any other attribute associated with the execution context
- Information about users such as group membership and roles

The main interface is **org.keycloak.authorization.policy.evaluation.Evaluation**, which defines the following contract:

```
public interface Evaluation {  
  
    /**  
     * Returns the {@link ResourcePermission} to be evaluated.  
     *  
     * @return the permission to be evaluated  
     */  
    ResourcePermission getPermission();  
  
    /**  
     * Returns the {@link EvaluationContext}. Which provi-
```

```

des access to the whole evaluation runtime context.

*
* @return the evaluation context
*/
EvaluationContext getContext();

/**
 * Returns a {@link Realm} that can be used by policies
to query information.
*
* @return a {@link Realm} instance
*/
Realm getRealm();

/**
 * Grants the requested permission to the caller.
*/
void grant();

/**
 * Denies the requested permission.
*/
void deny();
}

```

When processing an authorization request, Keycloak creates an `Evaluation` instance before evaluating any policy. This instance is then passed to each policy to determine whether access is **GRANT** or **DENY**.

Policies determine this by invoking the `grant()` or `deny()` methods on an `Evaluation` instance. By default, the state of the `Evaluation` instance is denied, which means that your policies must explicitly invoke the `grant()` method to indicate to the policy evaluation engine that permission should be granted.

For more information about the Evaluation API see the {apidocs\_link}

[JavaDocs].

### 5.10.1. The Evaluation Context

The evaluation context provides useful information to policies during their evaluation.

```
public interface EvaluationContext {  
  
    /**  
     * Returns the {@link Identity} that represents an entity (person or non-person) to which the permissions must be granted, or not.  
     *  
     * @return the identity to which the permissions must be granted, or not  
     */  
    Identity getIdentity();  
  
    /**  
     * Returns all attributes within the current execution and runtime environment.  
     *  
     * @return the attributes within the current execution and runtime environment  
     */  
    Attributes getAttributes();  
}
```

From this interface, policies can obtain:

- The authenticated `Identity`
- Information about the execution context and runtime environment

The `Identity` is built based on the OAuth2 Access Token that was sent along with the authorization request, and this construct has access to all claims extracted from the original token. For example, if you are

using a *Protocol Mapper* to include a custom claim in an OAuth2 Access Token you can also access this claim from a policy and use it to build your conditions.

The `EvaluationContext` also gives you access to attributes related to both the execution and runtime environments. For now, there are only a few built-in attributes.

**Table 2. Execution and Runtime Attributes**

Name	Description	Type
<code>kc.time.date_time</code>	Current date and time	String. Format MM/dd/yyyy hh:mm:ss
<code>kc.client.network.ip_address</code>	IPv4 address of the client	String
<code>kc.client.network.host</code>	Client's host name	String
<code>kc.client.id</code>	The client id	String
<code>kc.client.user_agent</code>	The value of the 'User-Agent' HTTP header	String[]
<code>kc.realm.name</code>	The name of the realm	String

---

## 6. Managing Permissions

A permission associates the object being protected and the policies that must be evaluated to decide whether access should be granted.

After creating the resources you want to protect and the policies you want to use to protect these resources, you can start managing permissions. To manage permissions, click the **Permissions** tab when editing a resource server.

### *Permissions*



Permissions can be created to protect two main types of objects:

- **Resources**
- **Scopes**

To create a permission, select the permission type you want to create from the dropdown list in the upper right corner of the permission listing. The following sections describe these two types of objects in more detail.

### 6.1. Creating Resource-Based Permissions

A resource-based permission defines a set of one or more resources to protect using a set of one or more authorization policies.

To create a new resource-based permission, select **Resource-based** in

the dropdown list in the upper right corner of the permission listing.

### *Add Resource-Based Permission*

#### 6.1.1. Configuration

- **Name**

A human-readable and unique string describing the permission. A best practice is to use names that are closely related to your business and security requirements, so you can identify them more easily.

- **Description**

A string containing details about this permission.

- **Apply To Resource Type**

Specifies if the permission is applied to all resources with a given type. When selecting this field, you are prompted to enter the resource type to protect.

- **Resource Type**

Defines the resource type to protect. When defined, this permission is evaluated for all resources matching that type.

- **Resources**

Defines a set of one or more resources to protect.

- **Apply Policy**

Defines a set of one or more policies to associate with a permission. To associate a policy you can either select an existing policy or crea-

te a new one by selecting the type of the policy you want to create.

- **Decision Strategy**

The [Decision Strategy](#) for this permission.

### 6.1.2. Typed Resource Permission

Resource permissions can also be used to define policies that are to be applied to all resources with a given [type](#). This form of resource-based permission can be useful when you have resources sharing common access requirements and constraints.

Frequently, resources within an application can be categorized (or typed) based on the data they encapsulate or the functionality they provide. For example, a financial application can manage different banking accounts where each one belongs to a specific customer. Although they are different banking accounts, they share common security requirements and constraints that are globally defined by the banking organization. With typed resource permissions, you can define common policies to apply to all banking accounts, such as:

- Only the owner can manage his account
- Only allow access from the owner's country and/or region
- Enforce a specific authentication method

To create a typed resource permission, click [Apply to Resource Type](#) when creating a new resource-based permission. With `Apply to Resource Type` set to `On`, you can specify the type that you want to protect as well as the policies that are to be applied to govern access to all resources with type you have specified.

## *Example of a Typed Resource Permission*

## 6.2. Creating Scope-Based Permissions

A scope-based permission defines a set of one or more scopes to protect using a set of one or more authorization policies. Unlike resource-based permissions, you can use this permission type to create permissions not only for a resource, but also for the scopes associated with it, providing more granularity when defining the permissions that govern your resources and the actions that can be performed on them.

To create a new scope-based permission, select **Scope-based** in the dropdown list in the upper right corner of the permission listing.

### *Add Scope-Based Permission*

#### 6.2.1. Configuration

- **Name**

A human-readable and unique string describing the permission. A best practice is to use names that are closely related to your business and security requirements, so you can identify them more easily.

- **Description**

A string containing details about this permission.

- **Resource**

Restricts the scopes to those associated with the selected resource. If none is selected, all scopes are available.

- **Scopes**

Defines a set of one or more scopes to protect.

- **Apply Policy**

Defines a set of one or more policies to associate with a permission.

To associate a policy you can either select an existing policy or create a new one by selecting the type of the policy you want to create.

- **Decision Strategy**

The [Decision Strategy](#) for this permission.

## 6.3. Policy Decision Strategies

When associating policies with a permission, you can also define a decision strategy to specify how to evaluate the outcome of the associated policies to determine access.

- **Unanimous**

The default strategy if none is provided. In this case, *all* policies must evaluate to a positive decision for the final decision to be also positive.

- **Affirmative**

In this case, *at least one* policy must evaluate to a positive decision for the final decision to be also positive.

- **Consensus**

In this case, the number of positive decisions must be greater than the number of negative decisions. If the number of positive and negative decisions is equal, the final decision will be negative.



---

## 7. Evaluating and Testing Policies

When designing your policies, you can simulate authorization requests to test how your policies are being evaluated.

You can access the Policy Evaluation Tool by clicking the `Evaluate` tab when editing a resource server. There you can specify different inputs to simulate real authorization requests and test the effect of your policies.



### 7.1. Providing Identity Information

The **Identity Information** filters can be used to specify the user requesting permissions.

### 7.2. Providing Contextual Information

The **Contextual Information** filters can be used to define additional attributes to the evaluation context, so that policies can obtain these same attributes.

### 7.3. Providing the Permissions

The **Permissions** filters can be used to build an authorization request. You can request permissions for a set of one or more resources and scopes. If you want to simulate authorization requests based on all protected resources and scopes, click **Add** without specifying any `Resources` or `Scopes`.

When you've specified your desired values, click **Evaluate**.

---

## 8. Authorization Services

KeyCloak Authorization Services are built on top of well-known standards such as the OAuth2 and User-Managed Access specifications.

OAuth2 clients (such as front end applications) can obtain access tokens from the server using the token endpoint and use these same tokens to access resources protected by a resource server (such as back end services). In the same way, KeyCloak Authorization Services provide extensions to OAuth2 to allow access tokens to be issued based on the processing of all policies associated with the resource(s) or scope(s) being requested. This means that resource servers can enforce access to their protected resources based on the permissions granted by the server and held by an access token. In KeyCloak Authorization Services the access token with permissions is called a Requesting Party Token or RPT for short.

In addition to the issuance of RPTs, KeyCloak Authorization Services also provides a set of RESTful endpoints that allow resources servers to manage their protected resources, scopes, permissions and policies, helping developers to extend or integrate these capabilities into their applications in order to support fine-grained authorization.

### 8.1. Discovering Authorization Services Endpoints and Metadata

KeyCloak provides a discovery document from which clients can obtain all necessary information to interact with KeyCloak Authorization Ser-

vices, including endpoint locations and capabilities.

The discovery document can be obtained from:

```
curl -X GET \
  http://${host}:${port}/auth/realms/${realm}/.well-
known/uma2-configuration
```

Where `${host}:${port}` is the hostname (or IP address) and port where Keycloak is running and `${realm}` is the name of a realm in Keycloak.

As a result, you should get a response as follows:

```
{
  // some claims are expected here

  // these are the main claims in the discovery document
  // about Authorization Services endpoints location
  "token_endpoint": "http://${host}:${post}/auth/re-
  alms/${realm}/protocol/openid-connect/token",
  "token_introspection_endpoint":
  "http://${host}:${post}/auth/realms/${realm}/proto-
  col/openid-connect/token/introspect",
  "resource_registration_endpoint":
  "http://${host}:${post}/auth/realms/${realm}/authz/protec-
  tion/resource_set",
  "permission_endpoint": "http://${host}:${post}/auth/re-
  alms/${realm}/authz/protection/permission",
  "policy_endpoint": "http://${host}:${post}/auth/re-
  alms/${realm}/authz/protection/uma-policy"
}
```

Each of these endpoints expose a specific set of capabilities:

- **token\_endpoint**

A OAuth2-compliant Token Endpoint that supports the `urn:ietf:params:oauth:grant-type:uma-ticket` grant type. Through this endpoint clients can send authorization requests and obtain an RPT with all permissions granted by KeyCloak.

- **token\_introspection\_endpoint**

A OAuth2-compliant Token Introspection Endpoint which clients can use to query the server to determine the active state of an RPT and to determine any other information associated with the token, such as the permissions granted by KeyCloak.

- **resource\_registration\_endpoint**

A UMA-compliant Resource Registration Endpoint which resource servers can use to manage their protected resources and scopes. This endpoint provides operations create, read, update and delete resources and scopes in KeyCloak.

- **permission\_endpoint**

A UMA-compliant Permission Endpoint which resource servers can use to manage permission tickets. This endpoint provides operations create, read, update, and delete permission tickets in KeyCloak.

## 8.2. Obtaining Permissions

To obtain permissions from KeyCloak you send an authorization request to the token endpoint. As a result, KeyCloak will evaluate all policies associated with the resource(s) and scope(s) being requested and issue an RPT with all permissions granted by the server.

Clients are allowed to send authorization requests to the token endpoint using the following parameters:

- **grant\_type**

This parameter is **required**. Must be `urn:ietf:params:oauth:grant-type:uma-ticket`.

- **ticket**

This parameter is **optional**. The most recent permission ticket received by the client as part of the UMA authorization process.

- **claim\_token**

This parameter is **optional**. A string representing additional claims that should be considered by the server when evaluating permissions for the resource(s) and scope(s) being requested. This parameter allows clients to push claims to Keycloak. For more details about all supported token formats see `claim_token_format` parameter.

- **claim\_token\_format**

This parameter is **optional**. A string indicating the format of the token specified in the `claim_token` parameter. Keycloak supports two token formats: `urn:ietf:params:oauth:token-type:jwt` and [https://openid.net/specs/openid-connect-core-1\\_0.html#IDToken](https://openid.net/specs/openid-connect-core-1_0.html#IDToken). The `urn:ietf:params:oauth:token-type:jwt` format indicates that the `claim_token` parameter references an access token. The [https://openid.net/specs/openid-connect-core-1\\_0.html#IDToken](https://openid.net/specs/openid-connect-core-1_0.html#IDToken) indicates that the `claim_token` parameter references an OpenID Connect ID Token.

- **rpt**

This parameter is **optional**. A previously issued RPT which permissions should also be evaluated and added in a new one. This parameter allows clients in possession of an RPT to perform incremental authorization where permissions are added on demand.

- **permission**

This parameter is **optional**. A string representing a set of one or more resources and scopes the client is seeking access. This parameter can be defined multiple times in order to request permission for multiple resource and scopes. This parameter is an extension to `urn:ietf:params:oauth:grant-type:uma-ticket` grant type in order to allow clients to send authorization requests without a permission ticket. The format of the string must be: `RESOURCE_ID#SCOPE_ID`. For instance: `Resource A#Scope A, Resource A#Scope A, Scope B, Scope C, Resource A, #Scope A`.

- **audience**

This parameter is **optional**. The client identifier of the resource server to which the client is seeking access. This parameter is mandatory in case the `permission` parameter is defined. It serves as a hint to Keycloak to indicate the context in which permissions should be evaluated.

- **response\_include\_resource\_name**

This parameter is **optional**. A boolean value indicating to the server whether resource names should be included in the RPT's permissions. If false, only the resource identifier is included.

- **response\_permissions\_limit**

This parameter is **optional**. An integer N that defines a limit for the amount of permissions an RPT can have. When used together with `rpt` parameter, only the last N requested permissions will be kept in the RPT.

- **submit\_request**

This parameter is **optional**. A boolean value indicating whether the server should create permission requests to the resources and scopes referenced by a permission ticket. This parameter only have effect if used together with the `ticket` parameter as part of a UMA authorization process.

- **response\_mode**

This parameter is **optional**. A string value indicating how the server should respond to authorization requests. This parameter is specially useful when you are mainly interested in either the overall decision or the permissions granted by the server, instead of a standard OAuth2 response. Possible values are:

- `decision`

Indicates that responses from the server should only represent the overall decision by returning a JSON with the following format:

```
{  
  'result': true  
}
```

If the authorization request does not map to any permission, a `403` HTTP status code is returned instead.

- `permissions`

Indicates that responses from the server should contain any permission granted by the server by returning a JSON with the following format:

```
[  
  {  
    'rsid': 'My Resource'  
    'scopes': ['view', 'update']  
  },  
  ...  
]
```

If the authorization request does not map to any permission, a **403** HTTP status code is returned instead.

Example of a authorization request when a client is seeking access to two resources protected by a resource server.

```
curl -X POST \  
  http://${host}:${port}/auth/realms/${realm}/proto-  
  col/openid-connect/token \  
  -H "Authorization: Bearer ${access_token}" \  
  --data "grant_type=urn:ietf:params:oauth:grant-type:uma-  
  ticket" \  
  --data "audience={resource_server_client_id}" \  
  --data "permission=Resource A#Scope A" \  
  --data "permission=Resource B#Scope B"
```

Example of a authorization request when a client is seeking access to any resource and scope protected by a resource server.

```
curl -X POST \  
  http://${host}:${port}/auth/realms/${realm}/proto-  
  col/openid-connect/token \  
  --data "grant_type=urn:ietf:params:oauth:grant-type:uma-  
  ticket" \  
  --data "audience={resource_server_client_id}" \  
  --data "scope=Resource A#Scope A" \  
  --data "scope=Resource B#Scope B"
```

```
-H "Authorization: Bearer ${access_token}" \
--data "grant_type=urn:ietf:params:oauth:grant-type:uma-ticket" \
--data "audience={resource_server_client_id}"
```

Example of an authorization request when a client is seeking access to a UMA protected resource after receiving a permission ticket from the resource server as part of the authorization process:

```
curl -X POST \
http://${host}:${port}/auth/realms/${realm}/proto-
col/openid-connect/token \
-H "Authorization: Bearer ${access_token}" \
--data "grant_type=urn:ietf:params:oauth:grant-type:uma-
ticket" \
--data "ticket=${permission_ticket}"
```

If KeyCloak assessment process results in issuance of permissions, it issues the RPT with which it has associated the permissions:

### *KeyCloak responds to the client with the RPT*

```
HTTP/1.1 200 OK
Content-Type: application/json
...
{
    "access_token": "${rpt}",
}
```

The response from the server is just like any other response from the token endpoint when using some other grant type. The RPT can be obtained from the `access_token` response parameter. If the client is not authorized, KeyCloak responds with a `403` HTTP status code:

## *KeyCloak denies the authorization request*

```
HTTP/1.1 403 Forbidden
Content-Type: application/json
...
{
    "error": "access_denied",
    "error_description": "request_denied"
}
```

### **8.2.1. Client Authentication Methods**

Clients need to authenticate to the token endpoint in order to obtain an RPT. When using the `urn:ietf:params:oauth:grant-type:uma-ticket` grant type, clients can use any of these authentication methods:

- **Bearer Token**

Clients should send an access token as a Bearer credential in an HTTP Authorization header to the token endpoint.

*Example: an authorization request using an access token to authenticate to the token endpoint*

```
curl -X POST \
  http://${host}:${port}/auth/realms/${realm}/proto-
  col/openid-connect/token \
  -H "Authorization: Bearer ${access_token}" \
  --data "grant_type=urn:ietf:params:oauth:grant-ty-
  pe:uma-ticket"
```

This method is especially useful when the client is acting on behalf of a user. In this case, the bearer token is an access token previously issued by KeyCloak to some client acting on behalf of a user (or on behalf of itself). Permissions will be evaluated considering the access context represented by the access token. For instance, if the ac-

cess token was issued to Client A acting on behalf of User A, permissions will be granted depending on the resources and scopes to which User A has access.

- **Client Credentials**

Client can use any of the client authentication methods supported by KeyCloak. For instance, `client_id/client_secret` or `JWT`.

*Example: an authorization request using client id and client secret to authenticate to the token endpoint*

```
curl -X POST \
    http://${host}:${port}/auth/realms/${realm}/proto-
    col/openid-connect/token \
    -H "Authorization: Basic cGhvdGg6L7Jl13RmfWgtkk==pOnN-
    lY3JldA==" \
    --data "grant_type=urn:ietf:params:oauth:grant-type-
    uma-ticket"
```

### 8.2.2. Pushing Claims

When obtaining permissions from the server you can push arbitrary claims in order to have these claims available to your policies when evaluating permissions.

If you are obtaining permissions from the server **without** using a permission ticket (UMA flow), you can send an authorization request to the token endpoint as follows:

```
curl -X POST \
    http://${host}:${port}/auth/realms/${realm}/proto-
    col/openid-connect/token \
    --data "grant_type=urn:ietf:params:oauth:grant-type:uma-
    ticket" \
    --data "claim_token=ewogICAi...jogWyJhY21l-
```

```
I10KfQ==" \
--data "claim_token_format=urn:ietf:params:oauth:token-type:jwt" \
--data "client_id={resource_server_client_id}" \
--data "client_secret={resource_server_client_secret}" \
--data "audience={resource_server_client_id}"
```

The `claim_token` parameter expects a BASE64 encoded JSON with a format similar to the example below:

```
{
    "organization" : ["acme"]
}
```

The format expects one or more claims where the value for each claim must be an array of strings.

### Pushing Claims Using UMA

For more details about how to push claims when using UMA and permission tickets, please take a look at [Permission API](#)

## 8.3. User-Managed Access

KeyCloak Authorization Services is based on User-Managed Access or UMA for short. UMA is a specification that enhances OAuth2 capabilities in the following ways:

- **Privacy**

Nowadays, user privacy is becoming a huge concern, as more and more data and devices are available and connected to the cloud.

With UMA and KeyCloak, resource servers can enhance their capabilities in order to improve how their resources are protected in re-

spect to user privacy where permissions are granted based on policies defined by the user.

- **Party-to-Party Authorization**

Resource owners (e.g.: regular end-users) can manage access to their resources and authorize other parties (e.g: regular end-users) to access these resources. This is different than OAuth2 where consent is given to a client application acting on behalf of a user, with UMA resource owners are allowed to consent access to other users, in a completely asynchronous manner.

- **Resource Sharing**

Resource owners are allowed to manage permissions to their resources and decide who can access a particular resource and how. KeyCloak can then act as a sharing management service from which resource owners can manage their resources.

KeyCloak is a UMA 2.0 compliant authorization server that provides most UMA capabilities.

As an example, consider a user Alice (resource owner) using an Internet Banking Service (resource server) to manage his Bank Account (resource). One day, Alice decides to open her bank account to Bob (requesting party), a accounting professional. However, Bob should only have access to view (scope) Alice's account.

As a resource server, the Internet Banking Service must be able to protect Alice's Bank Account. For that, it relies on KeyCloak Resource Registration Endpoint to create a resource in the server representing Alice's Bank Account.

At this moment, if Bob tries to access Alice's Bank Account, access will be denied. The Internet Banking Service defines a few default policies for banking accounts. One of them is that only the owner, in this case Alice, is allowed to access her bank account.

However, Internet Banking Service in respect to Alice's privacy also allows her to change specific policies for the banking account. One of these policies that she can change is to define which people are allowed to view her bank account. For that, Internet Banking Service relies on KeyCloak to provide to Alice a space where she can select individuals and the operations (or data) they are allowed to access. At any time, Alice can revoke access or grant additional permissions to Bob.

### 8.3.1. Authorization Process

In UMA, the authorization process starts when a client tries to access a UMA protected resource server.

A UMA protected resource server expects a bearer token in the request where the token is an RPT. When a client requests a resource at the resource server without a permission ticket:

*Client requests a protected resource without sending an RPT*

```
curl -X GET \
  http://${host}:${port}/my-resource-server/resource/1b-
  fdfef8-a4e1-4c2d-b142-fc92b75b986f
```

The resource server sends a response back to the client with a permission ticket and a `as_uri` parameter with the location of a KeyCloak server to where the ticket should be sent in order to obtain an RPT.

### *Resource server responds with a permission ticket*

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: UMA realm="${realm}",
    as_uri="https://${host}:${port}/auth/realms/${realm}",
    ticket="016f84e8-f9b9-11e0-bd6f-0021cc6004de"
```

The permission ticket is a special type of token issued by KeyCloak Permission API. They represent the permissions being requested (e.g.: resources and scopes) as well any other information associated with the request. Only resource servers are allowed to create those tokens.

Now that the client has a permission ticket and also the location of a KeyCloak server, the client can use the discovery document to obtain the location of the token endpoint and send an authorization request.

### *Client sends an authorization request to the token endpoint to obtain an RPT*

```
curl -X POST \
    http://${host}:${port}/auth/realms/${realm}/proto-
    col/openid-connect/token \
    -H "Authorization: Bearer ${access_token}" \
    --data "grant_type=urn:ietf:params:oauth:grant-type:uma-
    ticket" \
    --data "ticket=${permission_ticket}"
```

If KeyCloak assessment process results in issuance of permissions, it issues the RPT with which it has associated the permissions:

### *KeyCloak responds to the client with the RPT*

```
HTTP/1.1 200 OK
Content-Type: application/json
...
```

```
{  
    "access_token": "${rpt}",  
}
```

The response from the server is just like any other response from the token endpoint when using some other grant type. The RPT can be obtained from the `access_token` response parameter. In case the client is not authorized to have permissions KeyCloak responds with a `403` HTTP status code:

#### *KeyCloak denies the authorization request*

```
HTTP/1.1 403 Forbidden  
Content-Type: application/json  
...  
{  
    "error": "access_denied",  
    "error_description": "request_denied"  
}
```

#### **8.3.2. Submitting Permission Requests**

As part of the authorization process, clients need first to obtain a permission ticket from a UMA protected resource server in order to exchange it with an RPT at the KeyCloak Token Endpoint.

By default, KeyCloak responds with a `403` HTTP status code and a `request_denied` error in case the client can not be issued with an RPT.

#### *KeyCloak denies the authorization request*

```
HTTP/1.1 403 Forbidden  
Content-Type: application/json  
...
```

```
{  
  "error": "access_denied",  
  "error_description": "request_denied"  
}
```

Such response implies that Keycloak could not issue an RPT with the permissions represented by a permission ticket.

In some situations, client applications may want to start an asynchronous authorization flow and let the owner of the resources being requested decide whether or not access should be granted. For that, clients can use the `submit_request` request parameter along with an authorization request to the token endpoint:

```
curl -X POST \  
  http://{$host}:{$port}/auth/realms/{$realm}/proto-  
  col/openid-connect/token \  
  -H "Authorization: Bearer ${access_token}" \  
  --data "grant_type=urn:ietf:params:oauth:grant-type:uma-  
  ticket" \  
  --data "ticket=${permission_ticket}" \  
  --data "submit_request=true"
```

When using the `submit_request` parameter, Keycloak will persist a permission request for each resource to which access was denied. Once created, resource owners can check their account and manage their permissions requests.

You can think about this functionality as a `Request Access` button in your application, where users can ask other users for access to their resources.

### 8.3.3. Managing Access to Users Resources

Users can manage access to their resources using the Keycloak User Account Service. To enable this functionality, you must first enable User-Managed Access for your realm. To do so, open the realm settings page in Keycloak Administration Console and enable the User-Managed Access switch.



On the left side menu, the **My Resources** option leads to a page where users are able to:

- Manage Permission Requests that **Need my approval**

This section contains a list of all permission requests awaiting approval. These requests are connected to the parties (users) requesting access to a particular resource. Users are allowed to approve or deny these requests.

- Manage **My resources**

This section contains a list of all resources owned by the user. Users can click on a resource for more details and share the resource with others.

- Manage **Resources shared with me**

This section contains a list of all resources shared with the user.

- Manage **Your requests waiting approval**

This section contains a list of permission requests sent by the user that are waiting for the approval of another user or resource owner.

When the user choose to detail own of his resources by clicking on any

resource in the "My resources" listing, he is redirected to a page as follows:



From this page the users are able to:

- **Manage People with access to this resource**

This section contains a list of people with access to this resource.

Users are allowed to revoke access by clicking on the `Revoke` button or by removing a specific `Permission`.

- Share the resource with others

By typing the username or e-mail of another user, the user is able to share the resource and select the permissions he wants to grant access.

## 8.4. Protection API

The Protection API provides a UMA-compliant set of endpoints providing:

- **Resource Management**

With this endpoint, resource servers can manage their resources remotely and enable [policy enforcers](#) to query the server for the resources that need protection.

- **Permission Management**

In the UMA protocol, resource servers access this endpoint to create permission tickets. Keycloak also provides endpoints to manage the

state of permissions and query permissions.

- **Policy API**

KeyCloak leverages the UMA Protection API to allow resource servers to manage permissions for their users. In addition to the Resource and Permission APIs, KeyCloak provides a Policy API from where permissions can be set to resources by resource servers on behalf of their users.

An important requirement for this API is that *only* resource servers are allowed to access its endpoints using a special OAuth2 access token called a protection API token (PAT). In UMA, a PAT is a token with the scope **uma\_protection**.

#### 8.4.1. What is a PAT and How to Obtain It

A **protection API token** (PAT) is a special OAuth2 access token with a scope defined as **uma\_protection**. When you create a resource server, KeyCloak automatically creates a role, *uma\_protection*, for the corresponding client application and associates it with the client's service account.

*Service Account granted with **uma\_protection** role*

Resource servers can obtain a PAT from KeyCloak like any other OAuth2 access token. For example, using curl:

```
curl -X POST \
-H "Content-Type: application/x-www-form-urlencoded" \
-d 'grant_type=client_credentials&client_id=${client_id}&client_secret=${client_secret}' \
```

```
"http://localhost:8080/auth/realms/${realm_name}/proto-col/openid-connect/token"
```

The example above is using the **client\_credentials** grant type to obtain a PAT from the server. As a result, the server returns a response similar to the following:

```
{  
    "access_token": "${PAT}",  
    "expires_in": 300,  
    "refresh_expires_in": 1800,  
    "refresh_token": "${refresh_token}",  
    "token_type": "bearer",  
    "id_token": "${id_token}",  
    "not-before-policy": 0,  
    "session_state": "cce4a55-9aec-4024-b11c-44f6f168439e"  
}
```

Keycloak can authenticate your client application in different ways. For simplicity, the **client\_credentials** grant type is used here, which requires a *client\_id* and a *client\_secret*. You can choose to use any supported authentication method.

#### 8.4.2. Managing Resources

Resource servers can manage their resources remotely using a UMA-compliant endpoint.

```
http://${host}:${port}/auth/realms/${realm_name}/authz/protection/resource_set
```

This endpoint provides operations outlined as follows (entire path omit-

ted for clarity):

- Create resource set description: POST /resource\_set
- Read resource set description: GET /resource\_set/{\_id}
- Update resource set description: PUT /resource\_set/{\_id}
- Delete resource set description: DELETE /resource\_set/{\_id}
- List resource set descriptions: GET /resource\_set

For more information about the contract for each of these operations, see [UMA Resource Registration API](#).

## Creating a Resource

To create a resource you must send an HTTP POST request as follows:

```
curl -v -X POST \
  http://{$host}:{$port}/auth/realms/{$realm_na-
me}/authz/protection/resource_set \
  -H 'Authorization: Bearer '$pat \
  -H 'Content-Type: application/json' \
  -d '{
    "name": "Tweedl Social Service",
    "type": "http://www.example.com/rsrcts/socialstream/140-
compatible",
    "icon_uri": "http://www.example.com/icons/sharesocial.
png",
    "resource_scopes": [
      "read-public",
      "post-updates",
      "read-private",
      "http://www.example.com/scopes/all"
    ]
}'
```

By default, the owner of a resource is the resource server. If you want to define a different owner, such as an specific user, you can send a request as follows:

```
curl -v -X POST \
  http://${host}:${port}/auth/realms/${realm_na-
me}/authz/protection/resource_set \
  -H 'Authorization: Bearer '$pat \
  -H 'Content-Type: application/json' \
  -d '{
    "name": "Alice Resource",
    "owner": "alice"
}'
```

Where the property `owner` can be set with the username or the identifier of the user.

## Creating User-Managed Resources

By default, resources created via Protection API can not be managed by resource owners through the [User Account Service](#).

To create resources and allow resource owners to manage these resources, you must set `ownerManagedAccess` property as follows:

```
curl -v -X POST \
  http://${host}:${port}/auth/realms/${realm_na-
me}/authz/protection/resource_set \
  -H 'Authorization: Bearer '$pat \
  -H 'Content-Type: application/json' \
  -d '{
    "name": "Alice Resource",
    "owner": "alice",
    "ownerManagedAccess": true
}'
```

## Updating Resources

To update an existing resource, send an HTTP PUT request as follows:

```
curl -v -X PUT \
  http://${host}:${port}/auth/realms/${realm_na-
me}/authz/protection/resource_set/{resource_id} \
  -H 'Authorization: Bearer '$pat \
  -H 'Content-Type: application/json' \
  -d '{
    "_id": "Alice Resource",
    "name": "Alice Resource",
    "resource_scopes": [
      "read"
    ]
}'
```

## Deleting Resources

To delete an existing resource, send an HTTP DELETE request as follows:

```
curl -v -X DELETE \
  http://${host}:${port}/auth/realms/${realm_na-
me}/authz/protection/resource_set/{resource_id} \
  -H 'Authorization: Bearer '$pat
```

## Querying Resources

To query the resources by `id`, send an HTTP GET request as follows:

```
http://${host}:${port}/auth/realms/${realm_name}/authz/pro-
tection/resource_set/{resource_id}
```

To query resources given a `name`, send an HTTP GET request as follows:

```
http://${host}:${port}/auth/realms/${realm_name}/authz/protection/resource_set?name=Alice Resource
```

To query resources given an `uri`, send an HTTP GET request as follows:

```
http://${host}:${port}/auth/realms/${realm_name}/authz/protection/resource_set?uri=/api/alice
```

To query resources given an `owner`, send an HTTP GET request as follows:

```
http://${host}:${port}/auth/realms/${realm_name}/authz/protection/resource_set?owner=alice
```

To query resources given an `type`, send an HTTP GET request as follows:

```
http://${host}:${port}/auth/realms/${realm_name}/authz/protection/resource_set?type=albums
```

To query resources given an `scope`, send an HTTP GET request as follows:

```
http://${host}:${port}/auth/realms/${realm_name}/authz/protection/resource_set?scope=read
```

When querying the server for permissions use parameters `first` and `max` results to limit the result.

### 8.4.3. Managing Permission Requests

Resource servers using the UMA protocol can use a specific endpoint to manage permission requests. This endpoint provides a UMA-compliant flow for registering permission requests and obtaining a permission ticket.

```
http://${host}:${port}/auth/realms/${realm_name}/authz/protection/permission
```

A [permission ticket](#) is a special security token type representing a permission request. Per the UMA specification, a permission ticket is:

A correlation handle that is conveyed from an authorization server to a resource server, from a resource server to a client, and ultimately from a client back to an authorization server, to enable the authorization server to assess the correct policies to apply to a request for authorization data.

In most cases, you won't need to deal with this endpoint directly. KeyCloak provides a [policy enforcer](#) that enables UMA for your resource server so it can obtain a permission ticket from the authorization server, return this ticket to client application, and enforce authorization decisions based on a final requesting party token (RPT).

The process of obtaining permission tickets from KeyCloak is performed by resource servers and not regular client applications, where permission tickets are obtained when a client tries to access a protected resource without the necessary grants to access the resource. The issuance of permission tickets is an important aspects when using UMA as it allows resource servers to:

- Abstract from clients the data associated with the resources protected by the resource server
- Register in the KeyCloak authorization requests which in turn can be used later in workflows to grant access based on the resource's owner consent
- Decouple resource servers from authorization servers and allow them to protect and manage their resources using different authorization servers

Client wise, a permission ticket has also important aspects that its worthy to highlight:

- Clients don't need to know about how authorization data is associated with protected resources. A permission ticket is completely opaque to clients.
- Clients can have access to resources on different resource servers and protected by different authorization servers

These are just some of the benefits brought by UMA where other aspects of UMA are strongly based on permission tickets, specially regarding privacy and user controlled access to their resources.

## Creating Permission Ticket

To create a permission ticket, send an HTTP POST request as follows:

```
curl -X POST \
  http://${host}:${port}/auth/realms/${realm_na- \
me}/authz/protection/permission \
  -H 'Authorization: Bearer '$pat \
  -H 'Content-Type: application/json' \
```

```
-d '['  
{  
    "resource_id": "{resource_id}",  
    "resource_scopes": [  
        "view"  
    ]  
}  
'
```

When creating tickets you can also push arbitrary claims and associate these claims with the ticket:

```
curl -X POST \  
    http://${host}:${port}/auth/realms/${realm_na-  
me}/authz/protection/permission \  
    -H 'Authorization: Bearer '$pat \  
    -H 'Content-Type: application/json' \  
    -d '['  
    {  
        "resource_id": "{resource_id}",  
        "resource_scopes": [  
            "view"  
        ],  
        "claims": {  
            "organization": ["acme"]  
        }  
    }  
'
```

Where these claims will be available to your policies when evaluating permissions for the resource and scope(s) associated with the permission ticket.

## Other non UMA-compliant endpoints

### Creating permission ticket

To grant permissions for a specific resource with id {resource\_id} to a

user with id {user\_id}, as an owner of the resource send an HTTP POST request as follows:

```
curl -X POST \
    http://${host}:${port}/auth/realms/${realm_na-
me}/authz/protection/permission/ticket \
    -H 'Authorization: Bearer '$access_token \
    -H 'Content-Type: application/json' \
    -d '{
        "resource": "{resource_id}",
        "requester": "{user_id}",
        "granted": true,
        "scopeName": "view"
    }'
```

## Getting permission tickets

```
curl http://${host}:${port}/auth/realms/${realm_na-
me}/authz/protection/permission/ticket \
    -H 'Authorization: Bearer '$access_token
```

You can use any of these query parameters:

- `scopeId`
- `resourceId`
- `owner`
- `requester`
- `granted`
- `returnNames`
- `first`
- `max`

## Updating permission ticket

```
curl -X PUT \
    http://${host}:${port}/auth/realms/${realm_name}/authz/protection/permission/ticket \
    -H 'Authorization: Bearer '$access_token \
    -H 'Content-Type: application/json' \
    -d '{
        "id": "{ticket_id}",
        "resource": "{resource_id}",
        "requester": "{user_id}",
        "granted": false,
        "scopeName": "view"
    }'
```

## Deleting permission ticket

```
curl -X DELETE http://${host}:${port}/auth/realms/${realm_name}/authz/protection/permission/ticket/{ticket_id} \
    -H 'Authorization: Bearer '$access_token
```

### 8.4.4. Managing Resource Permissions using the Policy API

KeyCloak leverages the UMA Protection API to allow resource servers to manage permissions for their users. In addition to the Resource and Permission APIs, KeyCloak provides a Policy API from where permissions can be set to resources by resource servers on behalf of their users.

The Policy API is available at:

```
http://${host}:${port}/auth/realms/${realm_name}/authz/protection/uma-policy/{resource_id}
```

This API is protected by a bearer token that must represent a consent granted by the user to the resource server to manage permissions on his behalf. The bearer token can be a regular access token obtained from

the token endpoint using:

- Resource Owner Password Credentials Grant Type
- Token Exchange, in order to exchange an access token granted to some client (public client) for a token where audience is the resource server

## Associating a Permission with a Resource

To associate a permission with a specific resource you must send a HTTP POST request as follows:

```
curl -X POST \
  http://localhost:8180/auth/realms/photoz/authz/protection/uma-policy/{resource_id} \
  -H 'Authorization: Bearer '$access_token \
  -H 'Cache-Control: no-cache' \
  -H 'Content-Type: application/json' \
  -d '{
    "name": "Any people manager",
    "description": "Allow access to any people manager",
    "scopes": ["read"],
    "roles": ["people-manager"]
}'
```

In the example above we are creating and associating a new permission to a resource represented by `resource_id` where any user with a role `people-manager` should be granted with the `read` scope.

You can also create policies using other access control mechanisms, such as using groups:

```
curl -X POST \
```

```
http://localhost:8180/auth/realms/photoz/authz/protection/uma-policy/{resource_id} \
-H 'Authorization: Bearer '$access_token \
-H 'Cache-Control: no-cache' \
-H 'Content-Type: application/json' \
-d '{
    "name": "Any people manager",
    "description": "Allow access to any people manager",
    "scopes": ["read"],
    "groups": ["/Managers/People Managers"]
}'
```

Or a specific client:

```
curl -X POST \
  http://localhost:8180/auth/realms/photoz/authz/protection/uma-policy/{resource_id} \
  -H 'Authorization: Bearer '$access_token \
  -H 'Cache-Control: no-cache' \
  -H 'Content-Type: application/json' \
  -d '{
      "name": "Any people manager",
      "description": "Allow access to any people manager",
      "scopes": ["read"],
      "clients": ["my-client"]
}'
```

Or even using a custom policy using JavaScript:

```
curl -X POST \
  http://localhost:8180/auth/realms/photoz/authz/protection/uma-policy/{resource_id} \
  -H 'Authorization: Bearer '$access_token \
  -H 'Cache-Control: no-cache' \
  -H 'Content-Type: application/json' \
  -d '{
      "name": "Any people manager",
```

```
        "description": "Allow access to any people manager",
        "scopes": ["read"],
        "condition": "if (isPeopleManager()) {$evaluation.grant()}"'
    }'
```

It is also possible to set any combination of these access control mechanisms.

To update an existing permission, send an HTTP PUT request as follows:

```
curl -X PUT \
  http://localhost:8180/auth/realmz/photoz/authz/protection/uma-policy/{permission_id} \
  -H 'Authorization: Bearer '$access_token \
  -H 'Content-Type: application/json' \
  -d '{
    "id": "21eb3fed-02d7-4b5a-9102-29f3f09b6de2",
    "name": "Any people manager",
    "description": "Allow access to any people manager",
    "type": "uma",
    "scopes": [
      "album:view"
    ],
    "logic": "POSITIVE",
    "decisionStrategy": "UNANIMOUS",
    "owner": "7e22131a-aa57-4f5f-b1db-6e82abcd322",
    "roles": [
      "user"
    ]
}'
```

## Removing a Permission

To remove a permission associated with a resource, send an HTTP DELETE request as follows:

```
curl -X DELETE \
    http://localhost:8180/auth/realms/photoz/authz/protection/uma-policy/{permission_id} \
    -H 'Authorization: Bearer '$access_token
```

## Querying Permission

To query the permissions associated with a resource, send an HTTP GET request as follows:

```
http://${host}:${port}/auth/realms/${realm}/authz/protection/uma-policy?resource={resource_id}
```

To query the permissions given its name, send an HTTP GET request as follows:

```
http://${host}:${port}/auth/realms/${realm}/authz/protection/uma-policy?name=Any people manager
```

To query the permissions associated with a specific scope, send an HTTP GET request as follows:

```
http://${host}:${port}/auth/realms/${realm}/authz/protection/uma-policy?scope=read
```

To query all permissions, send an HTTP GET request as follows:

```
http://${host}:${port}/auth/realms/${realm}/authz/protection/uma-policy
```

When querying the server for permissions use parameters `first` and

`max` results to limit the result.

## 8.5. Requesting Party Token

A requesting party token (RPT) is a [JSON web token \(JWT\)](#) digitally signed using [JSON web signature \(JWS\)](#). The token is built based on the OAuth2 access token previously issued by Keycloak to a specific client acting on behalf of a user or on its own behalf.

When you decode an RPT, you see a payload similar to the following:

```
{  
  "authorization": {  
    "permissions": [  
      {  
        "resource_set_id": "d2fe9843-6462-4bfc-baba-  
b5787bb6e0e7",  
        "resource_set_name": "Hello World Resource"  
      }  
    ]  
  },  
  "jti": "d6109a09-78fd-4998-bf89-95730dfd0892-  
1464906679405",  
  "exp": 1464906971,  
  "nbf": 0,  
  "iat": 1464906671,  
  "sub": "f1888f4d-5172-4359-be0c-af338505d86c",  
  "typ": "kc_ett",  
  "azp": "hello-world-authz-service"  
}
```

From this token you can obtain all permissions granted by the server from the **permissions** claim.

Also note that permissions are directly related with the resources/scopes you are protecting and completely decoupled from the access control

methods that were used to actually grant and issue these same permissions.

### 8.5.1. Introspecting a Requesting Party Token

Sometimes you might want to introspect a requesting party token (RPT) to check its validity or obtain the permissions within the token to enforce authorization decisions on the resource server side.

There are two main use cases where token introspection can help you:

- When client applications need to query the token validity to obtain a new one with the same or additional permissions
- When enforcing authorization decisions at the resource server side, especially when none of the built-in [policy enforcers](#) fits your application

### 8.5.2. Obtaining Information about an RPT

The token introspection is essentially a [OAuth2 token introspection](#)-compliant endpoint from which you can obtain information about an RPT.

```
http://${host}:${port}/auth/realms/${realm_name}/proto-
col/openid-connect/token/introspect
```

To introspect an RPT using this endpoint, you can send a request to the server as follows:

```
curl -X POST \
-H "Authorization: Basic aGVsbG8td29ybGQtYXV0aH0t-
c2Vydm1jZTpzZWNyZXQ=" \
```

```
-H "Content-Type: application/x-www-form-urlencoded" \
-d 'token_type_hint=requesting_party_to-
ken&token=${RPT}' \
"http://localhost:8080/auth/realms/hello-world-aut-
hz/protocol/openid-connect/token/introspect"
```

The request above is using HTTP BASIC and passing the client's credentials (client ID and secret) to authenticate the client attempting to introspect the token, but you can use any other client authentication method supported by Keycloak.

The introspection endpoint expects two parameters:

- **token\_type\_hint**

Use **requesting\_party\_token** as the value for this parameter, which indicates that you want to introspect an RPT.

- **token**

Use the token string as it was returned by the server during the authorization process as the value for this parameter.

As a result, the server response is:

```
{
  "permissions": [
    {
      "resource_id": "90ccc6fc-b296-4cd1-881e-
089e1ee15957",
      "resource_name": "Hello World Resource"
    }
  ],
  "exp": 1465314139,
```

```
"nbf": 0,  
"iat": 1465313839,  
"aud": "hello-world-authz-service",  
"active": true  
}
```

If the RPT is not active, this response is returned instead:

```
{  
  "active": false  
}
```

### 8.5.3. Do I Need to Invoke the Server Every Time I Want to Introspect an RPT?

No. Just like a regular access token issued by a KeyCloak server, RPTs also use the [JSON web token \(JWT\)](#) specification as the default format.

If you want to validate these tokens without a call to the remote introspection endpoint, you can decode the RPT and query for its validity locally. Once you decode the token, you can also use the permissions within the token to enforce authorization decisions.

This is essentially what the [policy enforcers](#) do. Be sure to:

- Validate the signature of the RPT (based on the realm's public key)
- Query for token validity based on its *exp*, *iat*, and *aud* claims

## 8.6. Authorization Client Java API

Depending on your requirements, a resource server should be able to manage resources remotely or even check for permissions programmatically. If you are using Java, you can access the KeyCloak Authorization

Services using the Authorization Client API.

It is targeted for resource servers that want to access the different endpoints provided by the server such as the Token Endpoint, Resource, and Permission management endpoints.

### 8.6.1. Maven Dependency

```
<dependencies>
    <dependency>
        <groupId>org.keycloak</groupId>
        <artifactId>keycloak-authz-client</artifactId>
        <version>${KEYCLOAK_VERSION}</version>
    </dependency>
</dependencies>
```

### 8.6.2. Configuration

The client configuration is defined in a `keycloak.json` file as follows:

```
{
    "realm": "hello-world-authz",
    "auth-server-url" : "http://localhost:8080/auth",
    "resource" : "hello-world-authz-service",
    "credentials": {
        "secret": "secret"
    }
}
```

- **realm** (required)

The name of the realm.

- **auth-server-url** (required)

The base URL of the KeyCloak server. All other KeyCloak pages

and REST service endpoints are derived from this. It is usually in the form <https://host:port/auth>.

- **resource** (required)

The client-id of the application. Each application has a client-id that is used to identify the application.

- **credentials** (required)

Specifies the credentials of the application. This is an object notation where the key is the credential type and the value is the value of the credential type.

The configuration file is usually located in your application's classpath, the default location from where the client is going to try to find a `keycloak.json` file.

### 8.6.3. Creating the Authorization Client

Considering you have a `keycloak.json` file in your classpath, you can create a new `AuthzClient` instance as follows:

```
// create a new instance based on the configuration defined in a keycloak.json located in your classpath
AuthzClient authzClient = AuthzClient.create();
```

### 8.6.4. Obtaining User Entitlements

Here is an example illustrating how to obtain user entitlements:

```
// create a new instance based on the configuration defined in keycloak-authz.json
AuthzClient authzClient = AuthzClient.create();
```

```

// create an authorization request
AuthorizationRequest request = new AuthorizationRequest();

// send the entitlement request to the server in order to
// obtain an RPT with all permissions granted to the user
AuthorizationResponse response = authzClient.authorization("alice", "alice").authorize(request);
String rpt = response.getToken();

System.out.println("You got an RPT: " + rpt);

// now you can use the RPT to access protected resources on
the resource server

```

Here is an example illustrating how to obtain user entitlements for a set of one or more resources:

```

// create a new instance based on the configuration defined
in keycloak-authz.json
AuthzClient authzClient = AuthzClient.create();

// create an authorization request
AuthorizationRequest request = new AuthorizationRequest();

// add permissions to the request based on the resources
and scopes you want to check access
request.addPermission("Default Resource");

// send the entitlement request to the server in order to
// obtain an RPT with permissions for a single resource
AuthorizationResponse response = authzClient.authorization("alice", "alice").authorize(request);
String rpt = response.getToken();

System.out.println("You got an RPT: " + rpt);

// now you can use the RPT to access protected resources on
the resource server

```

## 8.6.5. Creating a Resource Using the Protection API

```

// create a new instance based on the configuration defined
in keycloak-authz.json
AuthzClient authzClient = AuthzClient.create();

// create a new resource representation with the information we want
ResourceRepresentation newResource = new ResourceRepresentation();

newResource.setName("New Resource");
newResource.setType("urn:hello-world-authz:resources:example");

newResource.addScope(new ScopeRepresentation("urn:hello-
world-authz:scopes:view"));

ProtectedResource resourceClient = authzClient.protection().resource();
ResourceRepresentation existingResource = resourceClient.findByName(newResource.getName());

if (existingResource != null) {
    resourceClient.delete(existingResource.getId());
}

// create the resource on the server
ResourceRepresentation response = resourceClient.create(ne-
wResource);
String resourceId = response.getId();

// query the resource using its newly generated id
ResourceRepresentation resource = resourceClient.findById(
    resourceId);

System.out.println(resource);

```

### 8.6.6. Introspecting an RPT

```

// create a new instance based on the configuration defined
in keycloak-authz.json
AuthzClient authzClient = AuthzClient.create();

```

```
// send the authorization request to the server in order to
// obtain an RPT with all permissions granted to the user
AuthorizationResponse response = authzClient.authorization-
on("alice", "alice").authorize();
String rpt = response.getToken();

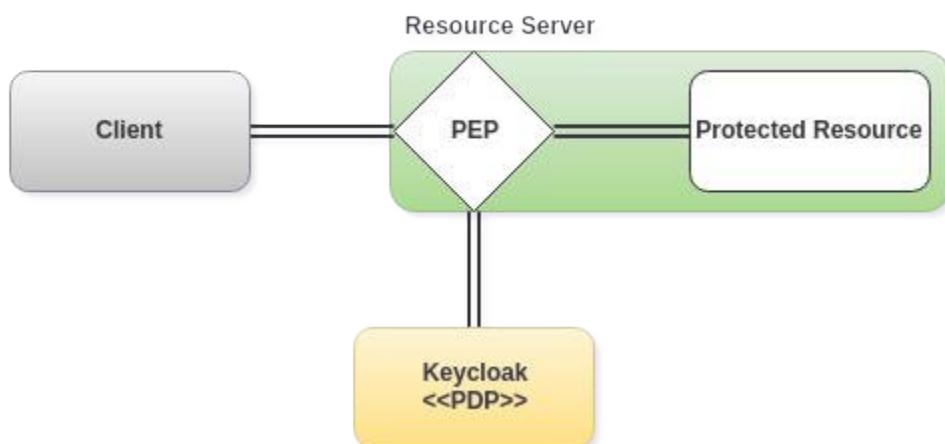
// introspect the token
TokenIntrospectionResponse requestingPartyToken = authzCli-
ent.protection().introspectRequestingPartyToken(rpt);

System.out.println("Token status is: " + requestingPartyTo-
ken.getActive());
System.out.println("Permissions granted by the server: ");

for (Permission granted : requestingPartyToken.getPermissi-
ons()) {
    System.out.println(granted);
}
```

## 9. Policy Enforcers

Policy Enforcement Point (PEP) is a design pattern and as such you can implement it in different ways. KeyCloak provides all the necessary means to implement PEPs for different platforms, environments, and programming languages. KeyCloak Authorization Services presents a RESTful API, and leverages OAuth2 authorization capabilities for fine-grained authorization using a centralized authorization server.



A PEP is responsible for enforcing access decisions from the KeyCloak server where these decisions are taken by evaluating the policies associated with a protected resource. It acts as a filter or interceptor in your application in order to check whether or not a particular request to a protected resource can be fulfilled based on the permissions granted by these decisions.

Permissions are enforced depending on the protocol you are using. When using UMA, the policy enforcer always expects an RPT as a bearer token in order to decide whether or not a request can be served. That means clients should first obtain an RPT from KeyCloak before sending

requests to the resource server.

However, if you are not using UMA, you can also send regular access tokens to the resource server. In this case, the policy enforcer will try to obtain permissions directly from the server.

If you are using any of the KeyCloak OIDC adapters, you can easily enable the policy enforcer by adding the following property to your **keycloak.json** file:

### *keycloak.json*

```
{  
  "policy-enforcer": {}  
}
```

When you enable the policy enforcer all requests sent to your application are intercepted and access to protected resources will be granted depending on the permissions granted by KeyCloak to the identity making the request.

Policy enforcement is strongly linked to your application's paths and the [resources](#) you created for a resource server using the KeyCloak Administration Console. By default, when you create a resource server, KeyCloak creates a [default configuration](#) for your resource server so you can enable policy enforcement quickly.

## 9.1. Configuration

To enable policy enforcement for your application, add the following property to your **keycloak.json** file:

## *keycloak.json*

```
{  
  "policy-enforcer": {}  
}
```

Or a little more verbose if you want to manually define the resources being protected:

```
{  
  "policy-enforcer": {  
    "user-managed-access" : {},  
    "enforcement-mode" : "ENFORCING"  
    "paths": [  
      {  
        "path" : "/someUri/*",  
        "methods" : [  
          {  
            "method": "GET",  
            "scopes" : ["urn:app.com:scopes:view"]  
          },  
          {  
            "method": "POST",  
            "scopes" : ["urn:app.com:scopes:create"]  
          }  
        ]  
      },  
      {  
        "name" : "Some Resource",  
        "path" : "/usingPattern/{id}",  
        "methods" : [  
          {  
            "method": "DELETE",  
            "scopes" : ["urn:app.com:scopes:delete"]  
          }  
        ]  
      },  
      {  
        "path" : "/exactMatch"  
      },  
      {
```

```
        "name" : "Admin Resources",
        "path" : "/usingWildCards/*"
    }
]
}
}
```

Here is a description of each configuration option:

- **policy-enforcer**

Specifies the configuration options that define how policies are actually enforced and optionally the paths you want to protect. If not specified, the policy enforcer queries the server for all resources associated with the resource server being protected. In this case, you need to ensure the resources are properly configured with a [URIS](#) property that matches the paths you want to protect.

- **user-managed-access**

Specifies that the adapter uses the UMA protocol. If specified, the adapter queries the server for permission tickets and returns them to clients according to the UMA specification. If not specified, the policy enforcer will be able to enforce permissions based on regular access tokens or RPTs. In this case, before denying access to the resource when the token lacks permission, the policy enforcer will try to obtain permissions directly from the server.

- **enforcement-mode**

Specifies how policies are enforced.

- **ENFORCING**

(default mode) Requests are denied by default even when there is no policy associated with a given resource.

- **PERMISSIVE**

Requests are allowed even when there is no policy associated with a given resource.

- **DISABLED**

Completely disables the evaluation of policies and allows access to any resource. When `enforcement-mode` is `DISABLED` applications are still able to obtain all permissions granted by Keycloak through the [Authorization Context](#)

- **on-deny-redirect-to**

Defines a URL where a client request is redirected when an "access denied" message is obtained from the server. By default, the adapter responds with a 403 HTTP status code.

- **path-cache**

Defines how the policy enforcer should track associations between paths in your application and resources defined in Keycloak. The cache is needed to avoid unnecessary requests to a Keycloak server by caching associations between paths and protected resources.

- **lifespan**

Defines the time in milliseconds when the entry should be expired. If not provided, default value is **3000**. A value less than or equal to 0 can be set to completely disable the cache.

- **max-entries**

Defines the limit of entries that should be kept in the cache. If not provided, default value is **1000**.

- **paths**

Specifies the paths to protect. This configuration is optional. If not defined, the policy enforcer will discover all paths by fetching the resources you defined to your application in Keycloak, where these resources are defined with **URIS** representing some paths in your application.

- **name**

The name of a resource on the server that is to be associated with a given path. When used in conjunction with a **path**, the policy enforcer ignores the resource's **URIS** property and uses the path you provided instead.

- **path**

(required) A URI relative to the application's context path. If this option is specified, the policy enforcer queries the server for a resource with a **URI** with the same value. Currently a very basic logic for path matching is supported. Examples of valid paths are:

- Wildcards: `/*`
- Suffix: `/* .html`
- Sub-paths: `/path/*`
- Path parameters: `/resource/{id}`

- Exact match: /resource
- Patterns: /{version}/resource, /api/{version}/resource, /api/{version}/resource/\*

- **methods**

The HTTP methods (for example, GET, POST, PATCH) to protect and how they are associated with the scopes for a given resource in the server.

- **method**

The name of the HTTP method.

- **scopes**

An array of strings with the scopes associated with the method. When you associate scopes with a specific method, the client trying to access a protected resource (or path) must provide an RPT that grants permission to all scopes specified in the list. For example, if you define a method *POST* with a scope *create*, the RPT must contain a permission granting access to the *create* scope when performing a POST to the path.

- **scopes-enforcement-mode**

A string referencing the enforcement mode for the scopes associated with a method. Values can be **ALL** or **ANY**. If **ALL**, all defined scopes must be granted in order to access the resource using that method. If **ANY**, at least one scope should be granted in order to gain access to the resource using that method. By default, enforcement mode

is set to **ALL**.

- **enforcement-mode**

Specifies how policies are enforced.

- **ENFORCING**

(default mode) Requests are denied by default even when there is no policy associated with a given resource.

- **DISABLED**

- **claim-information-point**

Defines a set of one or more claims that must be resolved and pushed to the Keycloak server in order to make these claims available to policies. See [Claim Information Point](#) for more details.

- **lazy-load-paths**

Specifies how the adapter should fetch the server for resources associated with paths in your application. If **true**, the policy enforcer is going to fetch resources on-demand accordingly with the path being requested. This configuration is specially useful when you don't want to fetch all resources from the server during deployment (in case you have provided no `paths`) or in case you have defined only a sub set of `paths` and want to fetch others on-demand.

- **http-method-as-scope**

Specifies how scopes should be mapped to HTTP methods. If set to **true**, the policy enforcer will use the HTTP method from the

current request to check whether or not access should be granted.

When enabled, make sure your resources in Keycloak are associated with scopes representing each HTTP method you are protecting.

- **claim-information-point**

Defines a set of one or more **global** claims that must be resolved and pushed to the Keycloak server in order to make these claims available to policies. See [Claim Information Point](#) for more details.

## 9.2. Claim Information Point

A Claim Information Point (CIP) is responsible for resolving claims and pushing these claims to the Keycloak server in order to provide more information about the access context to policies. They can be defined as a configuration option to the policy-enforcer in order to resolve claims from different sources, such as:

- HTTP Request (parameters, headers, body, etc)
- External HTTP Service
- Static values defined in configuration
- Any other source by implementing the Claim Information Provider SPI

When pushing claims to the Keycloak server, policies can base decisions not only on who a user is but also by taking context and contents into account, based on who, what, why, when, where, and which for a given transaction. It is all about Contextual-based Authorization and

how to use runtime information in order to support fine-grained authorization decisions.

### 9.2.1. Obtaining information from the HTTP Request

Here are several examples showing how you can extract claims from an HTTP request:

#### keycloak.json

```
"policy-enforcer": {
    "paths": [
        {
            "path": "/protected/resource",
            "claim-information-point": {
                "claims": {
                    "claim-from-request-parameter": "{request.parameter['a']}",
                    "claim-from-header": "{request.header['b']}",
                    "claim-from-cookie": "{request.cookie['c']}",
                    "claim-from-remoteAddr": "{request.remoteAddress}",
                    "claim-from-method": "{request.method}",
                    "claim-from-uri": "{request.uri}",
                    "claim-from-relativePath": "{request.relativePath}",
                    "claim-from-secure": "{request.secure}",
                    "claim-from-json-body-object": "{request.body['/a/b/c']}",
                    "claim-from-json-body-array": "{request.body['/d/1']}",
                    "claim-from-body": "{request.body}",
                    "claim-from-static-value": "static value",
                    "claim-from-multiple-static-value": ["static", "value"],
                    "param-replace-multiple-placeholder": "Test
{keycloak.access_token['/custom_claim/0']} and {request.parameter['a']} "
                }
            }
        }
    ]
}
```

```
    ]
}
```

### 9.2.2. Obtaining information from an External HTTP Service

Here are several examples showing how you can extract claims from an external HTTP Service:

*keycloak.json*

```
"policy-enforcer": {
  "paths": [
    {
      "path": "/protected/resource",
      "claim-information-point": {
        "http": {
          "claims": {
            "claim-a": "/a",
            "claim-d": "/d",
            "claim-d0": "/d/0",
            "claim-d-all": ["/d/0", "/d/1"]
          },
          "url": "http://mycompany/claim-provider",
          "method": "POST",
          "headers": {
            "Content-Type": "application/x-www-form-urlencoded",
            "header-b": ["header-b-value1", "header-b-value2"],
            "Authorization": "Bearer {keycloak.access_token}"
          },
          "parameters": {
            "param-a": ["param-a-value1", "param-a-value2"],
            "param-subject": "{keycloak.access_token['/sub']}",
            "param-user-name": "{keycloak.access_token['/preferred_username']}",
            "param-other-claims": "{keycloak.access_token['/custom_claim']}"
          }
        }
      }
    }
  ]
}
```

```
        }
    }
}
]
```

### 9.2.3. Static Claims

*keycloak.json*

```
"policy-enforcer": {
  "paths": [
    {
      "path": "/protected/resource",
      "claim-information-point": {
        "claims": {
          "claim-from-static-value": "static value",
          "claim-from-multiple-static-value": ["static",
"value"],
        }
      }
    }
  ]
}
```

### 9.2.4. Claim Information Provider SPI

The Claim Information Provider SPI can be used by developers to support different claim information points in case none of the built-ins providers are enough to address their requirements.

For example, to implement a new CIP provider you need to implement `org.keycloak.adapters.authorization.ClaimInformationPointProviderFactory` and `ClaimInformationPointProvider` and also provide the file `META-INF/services/org.keycloak.adapters.authorization.ClaimInformationPointProvider-Factory` in your application's classpath.

Example of `org.keycloak.adapters.authorization.ClaimInformationPointProviderFactory`:

```
public class MyClaimInformationPointProviderFactory implements ClaimInformationPointProviderFactory<MyClaimInformationPointProvider> {

    @Override
    public String getName() {
        return "my-claims";
    }

    @Override
    public void init(PolicyEnforcer policyEnforcer) {

    }

    @Override
    public MyClaimInformationPointProvider create(Map<String, Object> config) {
        return new MyClaimInformationPointProvider(config);
    }
}
```

Every CIP provider must be associated with a name, as defined above in the `MyClaimInformationPointProviderFactory.getName` method. The name will be used to map the configuration from the `claim-information-point` section in the `policy-enforcer` configuration to the implementation.

When processing requests, the policy enforcer will call the `MyClaimInformationPointProviderFactory.create` method in order to obtain an instance of `MyClaimInformationPointProvider`. When called, any configuration defined for this particular CIP provider (via `claim-information-point`) is passed as a map.

Example of `ClaimInformationPointProvider`:

```
public class MyClaimInformationPointProvider implements
ClaimInformationPointProvider {

    private final Map<String, Object> config;

    public ClaimsInformationPointProvider(Map<String, Object> config) {
        this.config = config;
    }

    @Override
    public Map<String, List<String>> resolve(HttpFacade
httpFacade) {
        Map<String, List<String>> claims = new HashMap<>();
        // put whatever claim you want into the map

        return claims;
    }
}
```

## 9.3. Obtaining the Authorization Context

When policy enforcement is enabled, the permissions obtained from the server are available through `org.keycloak.AuthorizationContext`. This class provides several methods you can use to obtain permissions and ascertain whether a permission was granted for a particular resource or scope.

### Obtaining the Authorization Context in a Servlet Container

```
HttpServletRequest request = ... // obtain javax.
servlet.http.HttpServletRequest
KeycloakSecurityContext keycloakSecurityContext =
(KeycloakSecurityContext) request
```

```
        .getAttribute(KeycloakSecurityContext.class.  
getName());  
    AuthorizationContext authzContext =  
        keycloakSecurityContext.getAuthorizationContext();
```

For more details about how you can obtain a `KeycloakSecurityContext` consult the adapter configuration. The example above should be sufficient to obtain the context when running an application using any of the servlet containers supported by Keycloak.

The authorization context helps give you more control over the decisions made and returned by the server. For example, you can use it to build a dynamic menu where items are hidden or shown depending on the permissions associated with a resource or scope.

```
if (authzContext.hasResourcePermission("Project Resource"))  
{  
    // user can access the Project Resource  
}  
  
if (authzContext.hasResourcePermission("Admin Resource")) {  
    // user can access administration resources  
}  
  
if (authzContext.hasScopePermission("urn:project.com:project:create")) {  
    // user can create new projects  
}
```

The `AuthorizationContext` represents one of the main capabilities of Keycloak Authorization Services. From the examples above, you can see that the protected resource is not directly associated with the poli-

cies that govern them.

Consider some similar code using role-based access control (RBAC):

```
if (User.hasRole('user')) {  
    // user can access the Project Resource  
}  
  
if (User.hasRole('admin')) {  
    // user can access administration resources  
}  
  
if (User.hasRole('project-manager')) {  
    // user can create new projects  
}
```

Although both examples address the same requirements, they do so in different ways. In RBAC, roles only *implicitly* define access for their resources. With KeyCloak you gain the capability to create more manageable code that focuses directly on your resources whether you are using RBAC, attribute-based access control (ABAC), or any other BAC variant. Either you have the permission for a given resource or scope, or you don't.

Now, suppose your security requirements have changed and in addition to project managers, PMOs can also create new projects.

Security requirements change, but with KeyCloak there is no need to change your application code to address the new requirements. Once your application is based on the resource and scope identifier, you need only change the configuration of the permissions or policies associated with a particular resource in the authorization server. In this case, the permissions and policies associated with the `Project Resource`

and/or the scope `urn:project.com:project:create` would be changed.

## 9.4. Using the AuthorizationContext to obtain an Authorization Client Instance

The `AuthorizationContext` can also be used to obtain a reference to the [Authorization Client API](#) configured to your application:

```
ClientAuthorizationContext clientContext = ClientAuthorizationContext.class.cast(authzContext);
AuthzClient authzClient = clientContext.getClient();
```

In some cases, resource servers protected by the policy enforcer need to access the APIs provided by the authorization server. With an `AuthzClient` instance in hands, resource servers can interact with the server in order to create resources or check for specific permissions programmatically.

## 9.5. JavaScript Integration

The Keycloak Server comes with a JavaScript library you can use to interact with a resource server protected by a policy enforcer. This library is based on the Keycloak JavaScript adapter, which can be integrated to allow your client to obtain permissions from a Keycloak Server.

You can obtain this library from a running a Keycloak Server instance by including the following `script` tag in your web page:

```
<script src="http://.../auth/js/keycloak-authz.js">
</script>
```

Once you do that, you can create a `KeycloakAuthorization` instance as follows:

```
var keycloak = ... // obtain a Keycloak instance from keycloak.js library
var authorization = new KeycloakAuthorization(keycloak);
```

The `keycloak-authz.js` library provides two main features:

- Obtain permissions from the server using a permission ticket, if you are accessing a UMA protected resource server.
- Obtain permissions from the server by sending the resources and scopes the application wants to access.

In both cases, the library allows you to easily interact with both resource server and KeyCloak Authorization Services to obtain tokens with permissions your client can use as bearer tokens to access the protected resources on a resource server.

### 9.5.1. Handling Authorization Responses from a UMA-Protected Resource Server

If a resource server is protected by a policy enforcer, it responds to client requests based on the permissions carried along with a bearer token. Typically, when you try to access a resource server with a bearer token that is lacking permissions to access a protected resource, the resource server responds with a **401** status code and a `WWW-Authenticate` header.

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: UMA realm="${realm}",
```

```
as_uri="https://${host}:${port}/auth/realms/${realm}",
ticket="016f84e8-f9b9-11e0-bd6f-0021cc6004de"
```

See [UMA Authorization Process](#) for more information.

What your client needs to do is extract the permission ticket from the `WWW-Authenticate` header returned by the resource server and use the library to send an authorization request as follows:

```
// prepare a authorization request with the permission ticket
var authorizationRequest = {};
authorizationRequest.ticket = ticket;

// send the authorization request, if successful retry the request
Identity.authorization.authorize(authorizationRequest).then(function (rpt) {
    // onGrant
}, function () {
    // onDeny
}, function () {
    // onError
});
```

The `authorize` function is completely asynchronous and supports a few callback functions to receive notifications from the server:

- `onGrant` : The first argument of the function. If authorization was successful and the server returned an RPT with the requested permissions, the callback receives the RPT.
- `onDeny` : The second argument of the function. Only called if the server has denied the authorization request.

- `onError` : The third argument of the function. Only called if the server responds unexpectedly.

Most applications should use the `onGrant` callback to retry a request after a 401 response. Subsequent requests should include the RPT as a bearer token for retries.

### 9.5.2. Obtaining Entitlements

The `keycloak-authz.js` library provides an `entitlement` function that you can use to obtain an RPT from the server by providing the resources and scopes your client wants to access.

*Example about how to obtain an RPT with permissions for all resources and scopes the user can access*

```
authorization.entitlement('my-resource-server-id')
  .then(function (rpt) {
    // onGrant callback function.
    // If authorization was successful you'll receive an RPT
    // with the necessary permissions to access the resource server
  });

```

*Example about how to obtain an RPT with permissions for specific resources and scopes*

```
authorization.entitlement('my-resource-server', {
  "permissions": [
    {
      "id" : "Some Resource"
    }
  ]
}).then(function (rpt) {
  // onGrant
});

```

---

When using the `entitlement` function, you must provide the `client_id` of the resource server you want to access.

The `entitlement` function is completely asynchronous and supports a few callback functions to receive notifications from the server:

- `onGrant` : The first argument of the function. If authorization was successful and the server returned an RPT with the requested permissions, the callback receives the RPT.
- `onDeny` : The second argument of the function. Only called if the server has denied the authorization request.
- `onError` : The third argument of the function. Only called if the server responds unexpectedly.

### 9.5.3. Authorization Request

Both `authorize` and `entitlement` functions accept an authorization request object. This object can be set with the following properties:

- **permissions**

An array of objects representing the resource and scopes. For instance:

```
var authorizationRequest = {
    "permissions": [
        {
            "id" : "Some Resource",
            "scopes" : ["view", "edit"]
        }
    ]
}
```

- **metadata**

An object where its properties define how the authorization request should be processed by the server.

- **response\_include\_resource\_name**

A boolean value indicating to the server if resource names should be included in the RPT's permissions. If false, only the resource identifier is included.

- **response\_permissions\_limit**

An integer N that defines a limit for the amount of permissions an RPT can have. When used together with `rpt` parameter, only the last N requested permissions will be kept in the RPT

- **submit\_request**

A boolean value indicating whether the server should create permission requests to the resources and scopes referenced by a permission ticket. This parameter will only take effect when used together with the `ticket` parameter as part of a UMA authorization process.

#### 9.5.4. Obtaining the RPT

If you have already obtained an RPT using any of the authorization functions provided by the library, you can always obtain the RPT as follows from the authorization object (assuming that it has been initialized by one of the techniques shown earlier):

```
var rpt = authorization.rpt;
```

## 9.6. Setting Up TLS/HTTPS

When the server is using HTTPS, ensure your adapter is configured as follows:

*keycloak.json*

```
{  
  "truststore": "path_to_your_trust_store",  
  "truststore-password": "trust_store_password"  
}
```

The configuration above enables TLS/HTTPS to the Authorization Client, making possible to access a KeyCloak Server remotely using the HTTPS scheme.

It is strongly recommended that you enable TLS/HTTPS when accessing the KeyCloak Server endpoints.

Last updated 2019-06-13 12:48:56 MESZ

## Table of Contents

*{upgradingguide!}*

---

# {upgradingguide!}

Last updated 2019-06-13 12:48:57 MESZ

## Table of Contents

*{releasenotes\_name}*

---

{releasenotes\_name}

# {project\_name\_full} 6.0.0

## WildFly 16 Upgrade

Keycloak server was upgraded to use WildFly 16 under the covers.

### SmallRye Health and Metrics extensions

Keycloak now comes enabled with the SmallRye Health and Metrics extensions which provides standard health and metrics endpoints. We will add some documentation as well as Keycloak specific metrics soon.

## PS256 support

Thanks to [tnorimat](#) Keycloak now has support for signing and verifying tokens with PS256.

## MP-JWT Client Scope

New built-in client scope to make it easy to issue tokens following the Eclipse MicroProfile specification.

---

## {project\_name\_full} 5.0.0

### WildFly 15 Upgrade

Keycloak server was upgraded to use WildFly 15 under the covers.

---

## {project\_name\_full} 4.8.0.Final

### OpenShift Integration

It is now possible to fully secure OpenShift 3.11 with KeyCloak, including the ability to automatically expose Service Accounts as OAuth clients as clients to KeyCloak.

This is currently a technology preview feature.

### Support for DB2 removed

DB2 support has been deprecated for a while. With this release we have removed all support for DB2.

---

## {project\_name\_full} 4.7.0.Final

### Enhanced Remember Me

Introduced the ability to specify different session idle and max timeouts for remember me sessions. This enables remember me sessions to live longer than regular sessions.

### Pagination support for Groups

Large numbers of groups have previously caused issues in the admin console. This is now resolved by the introduction of pagination of groups.

### Improve startup time with large number of offline sessions

In the past, starting the server could take a long time if there were many offline sessions. This startup time has now been significantly reduced.

---

{project\_name\_full} 4.6.0.Final

---

# {project\_name\_full} 4.5.0.Final

## Signature SPI

The Signature SPI makes it possible to plug-in additional signature algorithms. This enables additional signatures and also enables changing how signatures are generated. For example, using this allows using an HSM device to sign tokens.

Thanks to [tnorimat](#) for contributing a significant part of this work.

## New Signature Algorithms

Alongside the Signature SPI there is now also support for additional signature algorithms.

Keycloak now has support for RS256, RS384, RS512, ES256, ES384, ES512, HS256, HS384 and HS512.

Elliptic Curve Digital Signature Algorithm (ES256/384/512) are very interesting as they provide similar security properties as RSA signatures, but use significantly less CPU.

HMAC (HS256/384/512) are also very useful when you do not want your application to verify the signature itself. Since these are symmetric signatures only Keycloak is able to verify the signature, which requires the application to use the token introspection endpoint to verify tokens.

Thanks to [tnorimat](#) for contributing a significant part of this work.

## Better Audience Support for OpenID Connect clients

It is now possible to specify the audiences in the tokens issued for OpenID Connect clients. There is also support for verification of audience on the adapter side.

## Minor improvements

- Added LocaleSelector SPI, which allows to change the way how the locale will be resolved for a particular request. Thanks to [knutz3n](#)
- Added an authenticator to automatically link Identity Provider identity to an existing account after first Idp authentication. Thanks to [slominskir](#)

---

# {project\_name\_full} 4.4.0.Final

## Authorization Services support in Node.js

Having authorization services support in Node.js makes it very easy to do fine-grained central authorization with the Node.js adapter.

## Minor improvements

- Update design for the welcome page
- Allow passing current locale to OAuth2 IdPs. Thanks to [knutz3n](#)
- Support Content-Security-Policy-Report-Only security header.  
Thanks to [knutz3n](#)
- Script based ProtocolMapper for SAML. Thanks to [AlistairDoswald](#)

---

## {project\_name\_full} 4.3.0.Final

### Hostname SPI

The hostname SPI introduces a more flexible way to configure the hostname for Keycloak. There are two built-in providers. The first is request, which uses the request headers to determine the hostname. The second is fixed, which allows configuring a fixed hostname. The latter makes sure that only valid hostnames can be used and also allows internal applications to invoke Keycloak through an alternative URL.

For more details refer to the threat mitigation section in the [{adminguide!}](#).

### X509 Client Authenticator

The newly added Client Authenticator uses X509 Client Certificates and Mutual TLS to secure a connection from the client. In addition to that the Keycloak Server validates Subject DN field of the client's certificate.

### Performance improvements to Authorization Services

For this release, we improved policy evaluation performance across the board, increasing reliability and throughput. The main changes we did were related with trying to optimize the policy evaluation path by avoiding unnecessary flows and collect decisions as soon as they happen. We also introduced a policy decision cache on a per request basis, avo-

iding redundant decisions from policies previously evaluated.

We are also working on other layers of cache which should give a much better experience. See [KEYCLOAK-7952](#).

## Choosing the response mode when obtaining permissions from the server

In previous versions, permissions were always returned from the server using standard OAuth2 response, containing the access and refresh tokens. In this release, clients can use a `response_mode` parameter to specify how the server should respond to an authorization request. This parameter accepts two values:

- `decision`

Indicating that responses should only contain a flag indicating whether or not permissions were granted by the server. Otherwise a `403` HTTP status code is returned.

- `permissions`

Indicating that a response should contain every single permission granted by the server using a JSON format.

## NodeJS Policy Enforcer

The [keycloak-nodejs-connect](#), an adapter for NodeJS, now supports constructs to protect resources based on decisions taken from the server. The new construct allows users to protect their resources using fine-grained permissions as follows:

```
app.get('/protected/resource', keycloak.enforcer('resour-
```

```
ce:view'), function (req, res) {  
  res.json({message: 'access granted'});  
});
```

## Support hosted domain for Google logins

Login with Google now supports the `hd` parameter to restrict Google logins to a specific hosted domain at Google. When this is specified in the identity provider any login from a different domain is rejected.

Thanks to [brushmate](#) for the contribution.

## Escape unsafe tags in HTML output

Most HTML output is already escaped for HTML tags, but there are some places where HTML tags are permitted. These are only where admin access is needed to update the value. Even though it would require admin access to update such fields we have added an extra layer of defence and are now escaping unsafe elements like `<script>`.

---

## {project\_name\_full} 4.2.0.Final

### Browser tab support for Cordova

We now have support for using browser tab and universal links in the JavaScript adapter for Cordova. This enables SSO between multiple applications as well as increases security.

Thanks to [gtudan](#) for the contribution.

### SAML adapter multitenancy support

The SAML adapter can support multi-tenancy now just like the built in adapter for OpenID Connect.

### An option to create claims with dots (.) in them

In previous versions, it was not possible to create claims in the token using a claim name containing a dot (.) character. Now it is possible to escape the dot character in the configuration, so a claim name with the dot character can be used.

---

## {project\_name\_full} 4.1.0.Final

### Making Spring Boot 2 the default starter

Starting with release 4.1, the Spring Boot starter will be based on the Spring Boot 2 adapter. If you are using an older Spring Boot version, the keycloak-legacy-spring-boot-starter is available.

---

## {project\_name\_full} 4.0.0.Final

### Client Scopes and support for OAuth 2 scope parameter

We added support for Client Scopes, which replaces Client Templates. Client Scopes are a more flexible approach and also provides better support for the OAuth `scope` parameter.

There are changes related to Client Scopes to the consent screen. The list on the consent screen is now linked to client scopes instead of protocol mappers and roles.

See the documentation and migration guide for more details.

### OAuth 2 Certificate Bound Access Tokens

We now have a partial implementation of the specification [OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens](#). More accurately we have support for the Certificate Bound Access Tokens. If your confidential client is able to use 2-way SSL, KeyCloak will be able to add the hash of the client certificate into the tokens issued for the client. At this moment, it's just the KeyCloak itself, which verifies the token hashes (for example during `refresh token` requests). We plan to add support to adapters as well. We also plan to add support for Mutual TLS Client Authentication.

Thanks to [tnorimat](#) for the contribution.

# Authorization Services

## UMA 2.0 Support

UMA 2.0 is now supported for Authorization Services. Check the documentation for more details if you are coming from previous versions of KeyCloak.

## User-Managed Access through the KeyCloak Account Service

Now end-users are able to manage their resources and the permissions associated with them through the KeyCloak Account Service. From there, resource owners can now check their resources, share resources with another users as well approve requests from other users.

## Asynchronous Authorization Flow

When using UMA, client applications can now choose whether or not an authorization request should start an authorization flow to ask for the resource owner approval. This functionality allows applications to ask for resource owner approval when trying to access one of his resources on behalf of another user.

## User-Managed Permission API

Resource servers are now capable of associating additional policies to resources owned by a particular user. The new API provides operations to manage these permissions using different policy types such as role, group, user, client or a condition using JavaScript.

## Pushed Claims

Clients applications are now able to send arbitrary claims to KeyCloak along with an authorization request in order to evaluate permissions based on these claims. This is a very handy addition when access

should be granted (or denied) in the scope of a specific transaction or based on information about the runtime.

## Resource Attributes

It is now possible to associate attributes with resources protected by Keycloak and use these same attributes to evaluate permissions from your policies.

## Policy enforcer now accepts regular access tokens

In some situations, you may want to just send regular access tokens to a resource server but still be able to enforce policies on these resources.

One of the main changes introduced by this release is that you are no longer required to exchange access tokens with RPTs in order to access resources protected by a resource server (when not using UMA). Depending on how the policy enforcer is configured on the resource server side, you can just send regular access tokens as a bearer token and permissions will still be enforced.

## Policy enforcer can now load resources from the server on-demand

Until now, when deploying an application configured with a `policy-enforcer`, the policy enforcer would either load all protected paths from the server or just map these paths from the adapter configuration. Users can now decide to load paths on-demand from the server and avoid map these resources in the adapter configuration. Depending on how many protected resources you have this functionality can also improve the time to deploy an application.

## Policy enforcer now supports configuring the resource cache

In order to avoid unnecessary hits to the server, the policy enforcer caches the mapping between protected resources and their corresponding paths in your application. Users can now configure the behaviour of the cache or even completely disable it.

## Claim Information Points

The `policy-enforcer` definition on the adapters (`keycloak.json`) was also updated to support the concept of pushed claims. There you have the concept of a `claim-information-point` which can be set to push claims from different sources such as the HTTP request or even from an external HTTP service.

## Improvements to the Evaluation API

The Evaluation API used to implement policies in KeyCloak, especially JavaScript and Drools policies, provides now methods to:

- Access information from the current realm such as check for user roles, groups and attributes
- Push back arbitrary claims to the resource server in order to provide additional information on how a specific permissions should be enforced

## Authorization Services

### UMA 2.0

UMA 2.0 is now supported for Authorization Services, including support for users to manage user access through the account management console. There are also other additions and improvements to authorization services.

## Pushed Claims

Clients can now push additional claims and have them used by policies when evaluating permissions.

## Resource Attributes

It is now possible to define attributes on resources in order to have them used by policies when evaluating permissions.

## Themes and Theme Resources

It is now possible to hot-deploy themes to Keycloak through a regular provider deployment. We have also added support for theme resources, which allows adding additional templates and resources without creating a theme. This is useful for custom authenticators that require additional pages to be added to the authentication flow.

We have also added support to override the theme for specific clients. If that is not adequate for your needs, then there is also a new Theme Selector SPI that allows you to implement custom logic to select the theme.

## Instagram Identity Provider

We have added support to login with Instagram. Thanks to [hguerrero](#) for the contribution.

## Search by User ID in Admin Console

To search for a user by id in the admin console you previously had to edit the URL. It is now possible to search directly in the user search field.

# Adapters

## Spring Boot 2

We now have support for Spring Boot 2.

## Fuse 7

We now have support for Fuse 7.

## JavaScript - Native Promise Support

The JavaScript adapter now supports native promises. It retains support for the old style promises as well. Both can be used interchangeably.

## JavaScript - Cordova Options

It is now possible to pass Cordova-specific options to login and other methods in the JavaScript adapter. Thanks to [loorent](#) for the contribution.

Last updated 2019-06-13 12:48:56 MESZ