# Keycloak
# Reference Guide

## SSO for Web Apps and REST Services

**1.9.1.Final**

## 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
```

# Chapter 1. License

Keycloak codebase is distributed under the ASL 2.0 license. It does not distribute any thirdparty libraries that are GPL. It does ship thirdparty libraries licensed under Apache ASL 2.0 and LGPL.

# Chapter 2. Overview

Keycloak is an SSO solution for web apps, mobile and RESTful web services. It is an authentication server where users can centrally login, logout, register, and manage their user accounts. The Keycloak admin UI can manage roles and role mappings for any application secured by Keycloak. The Keycloak Server can also be used to perform social logins via the user's favorite social media site i.e. Google, Facebook, Twitter etc.

Features:

- SSO and Single Log Out for browser applications
- Social Login. Enable Google, GitHub, Facebook, Twitter social login with no code required.
- LDAP and Active Directory support.
- Optional User Registration
- Password and TOTP support (via Google Authenticator). Client cert auth coming soon.
- Forgot password support. User can have an email sent to them
- Reset password/totp. Admin can force a password reset, or set up a temporary password.
- Not-before revocation policies per realm, application, or user.
- User session management. Admin can view user sessions and what applications/clients have an access token. Sessions can be invalidated per realm or per user.
- Pluggable theme and style support for user facing screens. Login, grant pages, account mgmt, and admin console all can be styled, branded, and tailored to your application and organizational needs.
- Integrated Browser App to REST Service token propagation
- OAuth Bearer token auth for REST Services
- OAuth 2.0 Grant requests
- OpenID Connect Support.
- SAML Support.
- CORS Support
- CORS Web Origin management and validation
- Completely centrally managed user and role mapping metadata. Minimal configuration at the application side

- Admin Console for managing users, roles, role mappings, clients, user sessions and allowed CORS web origins.
- Account Management console that allows users to manage their own account, view their open sessions, reset passwords, etc.
- Deployable as a WAR, appliance, or on Openshift. Completely clusterable.
- Multitenancy support. You can host and manage multiple realms for multiple organizations. In the same auth server and even within the same deployed application.
- Identity brokering/chaining. You can make the Keycloak server a child IDP to another SAML 2.0 or OpenID Connect IDP.
- Token claim, assertion, and attribute mappings. You can map user attributes, roles, and role names however you want into a OIDC ID Token, Access Token, SAML attribute statements, etc. This feature allows you to basically tailor however you want auth responses to look.
- Supports JBoss AS7, EAP 6.x, Wildfly, Tomcat 7, Tomcat 8, Jetty 9.1.x, Jetty 9.2.x, Jetty 8.1.x, and Pure JavaScript applications. Plans to support Node.js, RAILS, GRAILS, and other non-Java deployments

## 2.1. Key Concepts in Keycloak

The core concept in Keycloak is a *Realm*. A realm secures and manages security metadata for a set of users and registered clients. Users can be created within a specific realm within the Administration console. Roles (permission types) can be defined at the realm level and you can also set up user role mappings to assign these permissions to specific users.

A *client* is a service that is secured by a realm. You will often use Client for every Application secured by Keycloak. When a user browses an application's web site, the application can redirect the user agent to the Keycloak Server and request a login. Once a user is logged in, they can visit any other client (application) managed by the realm and not have to re-enter credentials. This also hold true for logging out. Roles can also be defined at the client level and assigned to specific users. Depending on the client type, you may also be able to view and manage user sessions from the administration console.

In admin console there is switch *Consent required* specified when creating/editing client. When on, the client is not immediately granted all permissions of the user. In addition to requesting the login credentials of the user, the Keycloak Server will also display a grant page asking the user if it is ok to grant allowed permissions to the client. The granted consents are saved and every user can see his granted consents in Account Management UI and he can also revoke them for particular client. Also admin can see and revoke the grants of particular user in Keycloak Admin Console UI.

## 2.2. How Does Security Work in Keycloak?

Keycloak uses *access tokens* to secure web invocations. Access tokens contains security metadata specifying the identity of the user as well as the role mappings for that user. The format of

these tokens is a Keycloak extension to the *JSON Web Token* [http://tools.ietf.org/html/draft-ietf-oauth-json-web-token-14] specification. Each realm has a private and public key pair which it uses to digitally sign the access token using the *JSON Web Signature* [http://tools.ietf.org/html/draft-ietf-jose-json-web-signature-19] specification. Applications can verify the integrity of the digitally signed access token using the public key of the realm. The protocols used to obtain this token is defined by the *OAuth 2.0* [http://tools.ietf.org/html/rfc6749] specification.

The interesting thing about using these *smart* access tokens is that applications themselves are completely stateless as far as security metadata goes. All the information they need about the user is contained in the token and there's no need for them to store any security metadata locally other than the public key of the realm.

Signed access tokens can also be propagated by REST client requests within an `Authorization` header. This is great for distributed integration as applications can request a login from a client to obtain an access token, then invoke any aggregated REST invocations to other services using that access token. So, you have a distributed security model that is centrally managed, yet does not require a Keycloak Server hit per request, only for the initial login.

### 2.2.1. Permission Scopes

Each client is configured with a set of permission scopes. These are a set of roles that a client is allowed to ask permission for. Access tokens are always granted at the request of a specific client. This also holds true for SSO. As you visit different sites, the application will redirect back to the Keycloak Server via the OAuth 2.0 protocol to obtain an access token specific to that application (client). The role mappings contained within the token are the intersection between the set of user role mappings and the permission scope of the client. So, access tokens are tailor made for each client and contain only the information required for by them.

# Chapter 3. Installation and Configuration of Keycloak Server

## 3.1. Installation

Keycloak Server has three downloadable distributions. To run the Keycloak server you need to have Java 8 already installed.

- `keycloak-1.9.1.Final.[zip|tar.gz]` - Standalone server
- `keycloak-overlay-1.9.1.Final.[zip|tar.gz]` - Installer for WildFly or JBoss EAP
- `keycloak-demo-1.9.1.Final.[zip|tar.gz]` - Development bundle including WildFly, Keycloak, examples and documentation

### 3.1.1. Install Standalone Server

For production and for non-JavaEE developers we recommend using the standalone Keycloak server. All you need to do is to download `keycloak-1.9.1.Final.zip` or `keycloak-1.9.1.Final.tar.gz`, unpackage and start to have a Keycloak server up and running.

To install first download either the zip or tar.gz and extract. Then start by running either:

```
keycloak-1.9.1.Final/bin/standalone.sh
```

or:

```
keycloak-1.9.1.Final/bin/standalone.bat
```

### 3.1.2. Install on existing WildFly 10.0.0.Final or JBoss EAP 7.0.0.Beta

Keycloak can be installed into an existing installations of WildFly 10.0.0.Final or JBoss EAP 7.0.0.Beta. To do this download `keycloak-overlay-1.9.1.Final.zip` or `keycloak-overlay-1.9.1.Final.tar.gz`. Once downloaded extract into the root directory of your installation.

To add Keycloak to existing standalone.xml server config run:

```

```

```
bin/jboss-cli.sh --file=bin/keycloak-install.cli
```

To add Keycloak to existing standalone-ha.xml server config run:

```
bin/jboss-cli.sh --file=bin/keycloak-install-ha.cli
```

If you want to add Keycloak to a different server config edit keycloak-install.cli or keycloak-install-ha.cli and change the name of the server config.

### 3.1.3. Install Development Bundle

The demo bundle contains everything you need to get started with Keycloak including documentation and examples. To install it first download `keycloak-demo-1.9.1.Final.zip` or `keycloak-demo-1.9.1.Final.tar.gz`. Once downloaded extract it inside `keycloak-demo-1.9.1.Final` you'll find `keycloak` which contains a full WildFly 10.0.0.Final server with Keycloak Server and Adapters included. You'll also find `docs` and `examples` which contains everything you need to get started developing applications that use Keycloak.

To start WildFly with Keycloak run:

```
keycloak-1.9.1.Final/bin/standalone.sh
```

or:

```
keycloak-1.9.1.Final/bin/standalone.bat
```

## 3.2. Configuring the Server

Although the Keycloak Server is designed to run out of the box, there's some things you'll need to configure before you go into production. Specifically:

- Configuring Keycloak to use a production database
- Setting up SSL/HTTPS
- Enforcing HTTPS connections

### 3.2.1. Admin User

To access the admin console to configure Keycloak you need an account to login. There is no built in user, instead you have to first create an admin account. This can done either by opening *http://*

*localhost:8080/auth* (creating a user through the browser can only be done through localhost) or you can use the add-user script from the command-line.

The `add-user` script creates a temporary file with the details of the user, which are imported at startup. To add a user with this script run:

```
bin/add-user.[sh|bat] -r master -u <username> -p <password>
```

Then restart the server.

For `keycloak-overlay`, please make sure to use:

```
bin/add-user-keycloak.[sh|bat] -r master -u <username> -p <password>
```

### 3.2.2. Relational Database Configuration

You might want to use a better relational database for Keycloak like PostgreSQL or MySQL. You might also want to tweak the configuration settings of the datasource. Please see the *Wildfly* [https://docs.jboss.org/author/display/WFLY8/DataSource+configuration] documentation on how to do this.

Keycloak runs on a Hibernate/JPA backend which is configured in the `standalone/configuration/keycloak-server.json`. By default the setting is like this:

```
"connectionsJpa": {
    "default": {
        "dataSource": "java:jboss/datasources/KeycloakDS",
        "databaseSchema": "update"
    }
},
```

Possible configuration options are:

dataSource
: JNDI name of the dataSource

jta
: boolean property to specify if datasource is JTA capable

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

databaseSchema
: Specify if schema should be updated or validated. Valid values are "update" and "validate" ("update is default).

showSql
: Specify whether Hibernate should show all SQL commands in the console (false by default)

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

schema
: Specify the database schema to use

#### 3.2.2.1. Tested databases

Here is list of RDBMS databases and corresponding JDBC drivers, which were tested with Keycloak. Note that Hibernate dialect is usually set automatically according to your database, but in some cases, you must manually set the proper dialect, as the default dialect may not work correctly. You can setup dialect by adding property `driverDialect` to the `keycloak-server.json` into `connectionsJpa` section (see above).

**Table 3.1. Tested databases**

| Database | JDBC driver | Hibernate Dialect |
|---|---|---|
| H2 1.3.161 | H2 1.3.161 | auto |
| MySQL 5.5 | MySQL Connector/J 5.1.25 | auto |
| PostgreSQL 9.2 | JDBC4 Postgresql Driver, Version 9.3-1100 | auto |
| PostgresPlus 9.4 | edb-jdbc17-4.0 | auto |
| Oracle 11g R1 | Oracle JDBC Driver v11.1.0.7 | auto |
| Microsoft SQL Server 2012 | Microsoft SQL Server JDBC Driver 4.0.2206.100 | org.hibernate.dialect.SQLServer2008Dialect |
| IBM DB2 10.5 | JDBC 4.0 Driver (db2jcc4.jar) 4.19.26 | auto |

### 3.2.3. MongoDB based model

Keycloak provides *MongoDB* [http://www.mongodb.com] based model implementation, which means that your identity data will be saved in MongoDB instead of traditional RDBMS. To configure Keycloak to use Mongo open `standalone/configuration/keycloak-server.json` in your favourite editor, then change:

```
"eventsStore": {
    "provider": "jpa",
    "jpa": {
        "exclude-events": [ "REFRESH_TOKEN" ]
    }
},

"realm": {
    "provider": "jpa"
},

"user": {
    "provider": "${keycloak.user.provider:jpa}"
},
```

to:

```
"eventsStore": {
    "provider": "mongo",
    "mongo": {
        "exclude-events": [ "REFRESH_TOKEN" ]
    }
},

"realm": {
    "provider": "mongo"
},

"user": {
    "provider": "mongo"
},
```

And at the end of the file add the snippet like this where you can configure details about your Mongo database:

```
"connectionsMongo": {
    "default": {
        "host": "127.0.0.1",
        "port": "27017",
        "db": "keycloak",
        "connectionsPerHost": 100,
        "databaseSchema": "update"
```

```
    }
}
```

All configuration options are optional. Default values for host and port are localhost and 27017. Default name of database is `keycloak` . You can also specify properties `user` and `password` if you want authenticate against your MongoDB. If user and password are not specified, Keycloak will connect unauthenticated to your MongoDB.

Finally there is set of optional configuration options, which can be used to specify connection-pooling capabilities of Mongo client. Supported int options are: `connectionsPerHost`, `threadsAllowedToBlockForConnectionMultiplier`, `maxWaitTime`, `connectTimeout` `socketTimeout`. Supported boolean options are: `socketKeepAlive`, `autoConnectRetry`. Supported long option is `maxAutoConnectRetryTime`. See *Mongo documentation* [http://api.mongodb.org/java/2.11.4/com/mongodb/MongoClientOptions.html] for details about those options and their default values.

Alternatively, you can configure MongoDB using a MongoDB *connection URI* [http://docs.mongodb.org/manual/reference/connection-string/]. In this case, you define all information concerning the connection and authentication within the URI, as described in the MongoDB documentation. Please note that the database specified within the URI is only used for authentication. To change the database used by keycloak you have to set `db` property as before. Therefore, a configuration like the following

```
"connectionsMongo": {
    "default": {
        "uri": "mongodb://user:password@127.0.0.1/authentication",
        "db": "keycloak"
    }
}
```

will authenticate the user against the authentication database, but store all keycloak related data in the keycloak database.

#### 3.2.3.1. MongoDB Replica Sets

In order to use a mongo replica set for Keycloak, one has to use URI based configuration, which supports the definition of replica sets out of the box: `mongodb://host1:27017,host2:27017,host3:27017/`.

### 3.2.4. Outgoing Server HTTP Requests

Keycloak server needs to invoke on remote HTTP endpoints to do things like backchannel logouts and other management functions. Keycloak maintains a HTTP client connection pool which

has various configuration settings you can specify before boot time. This is configured in the `standalone/configuration/keycloak-server.json`. By default the setting is like this:

```
"connectionsHttpClient": {
    "default": {}
},
```

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
: *Not supported yet, but we will support in future versions.* Password for the client's key. This is *REQUIRED* if `client-keystore` is set.

### 3.2.5. Securing Outgoing Server HTTP Requests

When Keycloak connects out to remote HTTP endpoints over secure https connection, it has to validate the other server's certificate in order to ensure it is connecting to a trusted server. That is necessary in order to prevent man-in-the-middle attacks.

How certificates are validated is configured in the `standalone/configuration/keycloak-server.json`. By default truststore provider is not configured, and any https connections fall back to standard java truststore configuration as described in *Java's JSSE Reference Guide* [https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html] - using `javax.net.ssl.trustStore system property`, otherwise `cacerts` file that comes with java is used.

Truststore is used when connecting securely to identity brokers, LDAP identity providers, when sending emails, and for backchannel communication with client applications. Some of these facilities may - in case when no trusted certificate is found in your configured truststore - fallback to using the JSSE provided truststore. The default JavaMail API implementation used to send out emails behaves in this way, for example.

You can add your truststore configuration by using the following template:

```
"truststore": {
    "file": {
        "file": "path to your .jks file containing public certificates",
        "password": "password",
        "hostname-verification-policy": "WILDCARD",
        "disabled": false
    }
}
```

Possible configuration options are:

file
: The value is the file 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. 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.

You can use *keytool* to create a new truststore file and add trusted host certificates to it:

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

### 3.2.6. SSL/HTTPS Requirement/Modes

**Warning**

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.

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 try to access Keycloak from a non-IP adress you will get an error.

Keycloak has 3 SSL/HTTPS modes which you can set up in the admin console under the Settings->Login page and the `Require SSL` select box. Each adapter config should mirror this server-side setting. See adapter config section for more details.

external
: 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 try to access Keycloak from a non-IP adress you will get an error.

none
: Keycloak does not require SSL.

all
: Keycloak requires SSL for all IP addresses.

### 3.2.7. SSL/HTTPS Setup

First enable SSL on Keycloak or on a reverse proxy in front of Keycloak. Then configure the Keycloak Server to enforce HTTPS connections.

### 3.2.7.1. Enable SSL on Keycloak

The following things need to be done

- Generate a self signed or third-party signed certificate and import it into a Java keystore using `keytool`.
- Enable Wildfly to use this certificate and turn on SSL/HTTPS.

#### 3.2.7.1.1. 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 Keycloak Server to.

##### 3.2.7.1.1.1. 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 on. Generate one is very easy to do with 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-----
MIIC2jCCAcICAQAwZTELMAkGA1UEBhMCVVMxCzAJBgNVBAgTAk1BMREwDwYDVQQHEwhXZXN0Zm9y
ZDEQMA4GA1UEChMHUmVkIEhhdDEQMA4GA1UECxMHUmVkIEhhdDESMBAGA1UEAxMJbG9jYWxob3N0
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAr7kck2TaavlEOGbcpi9c0rncY4HhdzmY
Ax2nZfq1eZEaIPqI5aTxwQZzzLDK9qbeAd8Ji79HzSqnRDxNYaZu7mAYhFKHgixsolE3o5Yfzbw1
29Rvy+eUVe+WZxv5oo9wolVVpdSINIMEL2LaFhtX/c1dqiqYVpfnvFshZQaIg2nL8juzZcBjj4as
H98gIS7khql/dkZKsw9NLvyxgJvp7PaXurX29fNf3ihG+oFrL22oFyV54BWWxXCKU/GPn61EGZGw
Ft2qSIGLdctpMD1aJR2bcnlhEjZKDksjQZoQ5YMXaAGkcYkG6QkgrocDE2YXDbi7GIdf9MegVJ35
2DQMpwIDAQABoDAwLgYJKoZIhvcNAQkOMSEwHzAdBgNVHQ4EFgQUQwlZJBA+fjiDdiVzaO9vrE/i
n2swDQYJKoZIhvcNAQELBQADggEBAC5FRvMkhal3q86tHPBYWBuTtmcSjs4qUm6V6f63frhveWHf
PzRrI1xH272XUIeBk0gtzWo0nNZnf0mMCtUBbHhhDcG82xolikfqibZijoQZCiGiedVjHJFtniDQ
9bMDUOXEMQ7gHZg5q6mJfNG9MbMpQaUVEEFvfGEQQxbiFK7hRWU8S23/d80e8nExgQxdJWJ6vd0X
MzzFK6j4Dj55bJVuM7GFmfdNC52pNOD5vYe47Aqh8oajHX9XTycVtPXl45rrWAH33ftbrS8SrZ2S
vqIFQeuLL3BaHwpl3t7j2lMWcK1p80laAxEASib/fAwrRHpLHBXRcq6uALUOZl4Alt8=
-----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 import your new CA generated certificate to your keystore:

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

#### 3.2.7.1.2. Installing the keystore to WildFly

Now that you have a Java keystore with the appropriate certificates, you need to configure your Wildfly installation to use it. First step is to move the keystore file to a directory you can reference in

configuration. I like to put it in `standalone/configuration`. Then you need to edit `standalone/configuration/standalone.xml` to enable SSL/HTTPS.

To the `security-realms` element add:

```
<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>
```

Find the element `<server name="default-server">` (it's a child element of `<subsystem xmlns="urn:jboss:domain:undertow:1.0">`) and add:

```
<https-listener name="https" socket-binding="https" security-
realm="UndertowRealm"/>
```

Check the *Wildfly Undertow* [https://docs.jboss.org/author/display/WFLY8/Undertow+(web)+subsystem+configuration] documentation for more information on fine tuning the socket connections.

### 3.2.7.2. Enable SSL on a Reverse Proxy

Follow the documentation for your web server to enable SSL and configure reverse proxy for Keycloak. It is important that you make sure the web server sets the `X-Forwarded-For` and `X-Forwarded-Proto` headers on the requests made to Keycloak. Next you need to enable `proxy-address-forwarding` on the Keycloak http connector. Assuming that your reverse proxy doesn't use port 8443 for SSL you also need to configure what port http traffic is redirected to.

#### 3.2.7.2.1. Configure WildFly

Open `standalone/configuration/standalone.xml` in your favorite editor.

First add `proxy-address-forwarding` and `redirect-socket` to the `http-listener` element:

```
<subsystem xmlns="urn:jboss:domain:undertow:1.1">
    ...
    <http-listener name="default" socket-binding="http"
        proxy-address-forwarding="true" redirect-socket="proxy-https"/>
    ...
```

```
</subsystem>
```

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>
```

Check the *WildFly* [https://docs.jboss.org/author/display/WFLY8/Undertow+(web)+subsystem+configuration] documentation for more information.

## 3.3. Keycloak server in Domain Mode

In domain mode, you start the server with the "domain" command instead of the "standalone" command. In this case, the Keycloak subsystem is defined in domain/configuration/domain.xml instead of standalone/configuration.standalone.xml. Inside domain.xml, you will see more than one profile. The Keycloak subsystem is defined for all initial profiles.

THe server is also added to server profiles. By default two servers are started in the main-server-group which uses the full profile.

You need to make sure `domain/servers/<SERVER NAME>/configuration` is identical for all servers in a group.

To deploy custom providers and themes you should deploys these as modules and make sure the modules are available to all servers in the group. See *Providers* and *Themes* sections for more information on how to do this.

## 3.4. Installing Keycloak Server as Root Context

The Keycloak server can be installed as the default web application. In doing so, the server can be referenced at `http://mydomain.com/` instead of `http://mydomain.com/auth`.

To do this, add the `default-web-module` attribute in the Undertow subystem in standalone.xml.

```
<subsystem xmlns="urn:jboss:domain:undertow:2.0">
    <server name="default-server">
      <host name="default-host" alias="localhost" default-web-module="keycloak-
server.war">
            <location name="/" handler="welcome-content"/>
```

```
        </host>
```

`keycloak-server.war` is the runtime name of the Keycloak server application. Note that the WAR file does not exist as a file. If its name changes (ie. `keycloak-server.war`) in the future, find its new name from the Keycloak log entry with `runtime-name:`.

### Note

If you have run your server before altering the root context, your database will contain references to the old /auth context. Your clients may also have incorrect references. To fix this on the server side, you will need to *export your database to json, make corrections, and then import.* Client-side keycloak.json files will need to be updated manually as well.

# Chapter 4. Providers and SPIs

Keycloak is designed to cover most use-cases without requiring custom code, but we also want it to be customizable. To achive this Keycloak has a number of SPIs which you can implement your own providers for.

## 4.1. Implementing a SPI

To implement an SPI you need to implement it's ProviderFactory and Provider interfaces. You also need to create a provider-configuration file. For example to implement the Event Listener SPI you need to implement EventListenerProviderFactory and EventListenerProvider and also provide the file `META-INF/services/org.keycloak.events.EventListenerProviderFactory`

For example to implement the Event Listener SPI you start by implementing EventListenerProviderFactory:

```
package org.acme.provider;

import ...

public class MyEventListenerProviderFactory implements
 EventListenerProviderFactory {

    private List<Event> events;

    public String getId() {
        return "my-event-listener";
    }

    public void init(Config.Scope config) {
        int max = config.getInt("max");
        events = new MaxList(max);
    }

    public EventListenerProvider create(KeycloakSession session) {
        return new MyEventListenerProvider(events);
    }

    public void close() {
        events = null;
    }

}
```

The example uses an imagined MaxList which has a maximum size and is concurrency safe. When the maximum size is reached and new entries are added the oldest entry is removed. Keycloak creates a single instance of EventListenerProviderFactory which makes it possible to store state for multiple requests. EventListenerProvider instances are created by calling create on the factory for each requests so these should be light-weight.

Next you would implement EventListenerProvider:

```
package org.acme.provider;

import ...

public class MyEventListenerProvider implements EventListenerProvider {

    private List<Event> events;

    public MyEventListenerProvider(List<Event> events) {
        this.events = events;
    }

    @Override
    public void onEvent(Event event) {
        events.add(event);
    }

    @Override
    public void close() {

    }

}
```

The file `META-INF/services/org.keycloak.events.EventListenerProviderFactory` should contain the full name of your ProviderFactory implementation:

```
org.acme.provider.MyEventListenerProviderFactory
```

### 4.1.1. Show info from you SPI implementation in Keycloak admin console

Sometimes it is useful to show additional info about your Provider to a Keycloak administrator. You can show provider build time informations (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 MyEventListenerProviderFactory from previous example:

```
package org.acme.provider;

import ...

public class MyEventListenerProviderFactory implements
 EventListenerProviderFactory, ServerInfoAwareProviderFactory {

    private List<Event> events;
    private int max;

...

    @Override
    public void init(Config.Scope config) {
        max = config.getInt("max");
        events = new MaxList(max);
    }

...

    @Override
    public Map<String, String> getOperationalInfo() {
        Map<String, String> ret = new LinkedHashMap<>();
        ret.put("version", "1.0");
        ret.put("listSizeMax", max + "");
        ret.put("listSizeCurrent", events.size() + "");
        return ret;
    }

}
```

## 4.2. Registering provider implementations

Keycloak can load provider implementations from JBoss Modules or directly from the file-system. Using Modules is recommended as you can control exactly what classes are available to your provider. Any providers loaded from the file-system uses a classloader with the Keycloak classloader as its parent.

### 4.2.1. 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.keycloak.examples.event-sysout --resources=target/event-listener-sysout-example.jar --dependencies=org.keycloak.keycloak-core,org.keycloak.keycloak-server-spi,org.keycloak.keycloak-events-api"
```

Or to manually create it start by creating the folder `KEYCLOAK_HOME/modules/org/keycloak/examples/event-sysout/main`. Then copy `event-listener-sysout-example.jar` to this folder and create `module.xml` with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.1" name="org.keycloak.examples.event-sysout">
    <resources>
        <resource-root path="event-listener-sysout-example.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 keycloak-server.json and adding it to the providers:

```
{
    "providers": [
        ...
        "module:org.keycloak.examples.event-sysout"
    ]
}
```

### 4.2.2. Register a provider using file-system

To register your provider simply copy the JAR including the ProviderFactory and Provider classes and the provider configuration file to server's root `providers` directory.

You can also define multiple provider class-path if you want to create isolated class-loaders. To do this edit keycloak-server.json and add more classpath entries to the providers array. For example:

```
{
    "providers": [
        "classpath:provider1.jar;lib-v1.jar",
        "classpath:provider2.jar;lib-v2.jar"
    ]
}
```

The above example will create two separate class-loaders for providers. The classpath entries follow the same syntax as Java classpath, with ';' separating multiple-entries. Wildcard is also supported allowing loading all jars (files with .jar or .JAR extension) in a folder, for example:

```
{
    "providers": [
        "classpath:/home/user/providers/*"
    ]
}
```

### 4.2.3. Configuring a provider

You can pass configuration options to your provider by setting them in `keycloak-server.json`. For example to set the max value for `my-event-listener` add:

```
{
    "eventsListener": {
        "my-event-listener": {
            "max": 100
        }
    }
}
```

### 4.2.4. Disabling a provider

You can disable a provider by setting the enabled field for the provider to false in `keycloak-server.json`. For example to disable the Infinispan user cache provider add:

```
{
"userCache": {
    "infinispan" : {
        "enabled": false
```

```
    }
}
}
```

## 4.3. Available SPIs

Here's a list of the available SPIs and a brief description. For more details on each SPI refer to individual sections.

Account
: Provides the account manage console pages. The default implementation uses FreeMarker templates.

Connections Infinispan
: Loads and configures Infinispan connections. The default implementation can load connections from the Infinispan subsystem, or alternatively can be manually configured in keycloak-server.json.

Connections Jpa
: Loads and configures Jpa connections. The default implementation can load datasources from WildFly/EAP, or alternatively can be manually configured in keycloak-server.json.

Connections Jpa Updater
: Updates database schema. The default implementation uses Liquibase.

Connections Mongo
: Loads and configures MongoDB connections. The default implementation is configured in keycloak-server.json.

Email
: Formats and sends email. The default implementation uses FreeMarker templates and JavaMail.

Events Listener
: Listen to user related events for example user login success and failures. Keycloak provides two implementations out of box. One that logs events to the server log and another that can send email notifications to users on certain events.

Events Store
: Store user related events so they can be viewed through the admin console and account management console. Keycloak provides implementations for Relational Databases and MongoDB.

Export
: Exports the Keycloak database. Keycloak provides implementations that export to JSON files either as a single file, multiple files in a directory or a encrypted ZIP archive.

Import
: Imports an exported Keycloak database. Keycloak provides implementations that import from JSON files either as a single file, multiple files in a directory or a encrypted ZIP archive.

Login
: Provides the login pages. The default implementation uses FreeMarker templates.

Login Protocol
: Provides protocols. Keycloak provides implementations of OpenID Connect and SAML 2.0.

Realm
: Provides realm and application meta-data. Keycloak provides implementations for Relational Databases and MongoDB.

Realm Cache
: Caches realm and application meta-data to improve performance. Keycloak provides a basic in-memory cache and a Infinispan cache.

Theme
: Allows creating themes to customize look and feel. Keycloak provides implementations that can load themes from the file-system or classpath.

Timer
: Executes scheduled tasks. Keycloak provides a basic implementation based on java.util.Timer.

User
: Provides users and role-mappings. Keycloak provides implementations for Relational Databases and MongoDB.

User Cache
: Caches users and role-mappings to improve performance. Keycloak provides a basic in-memory cache and a Infinispan cache.

User Federation
: Support syncing users from an external source. Keycloak provides implementations for LDAP and Active Directory.

User Sessions
: Provides users session information. Keycloak provides implementations for basic in-memory, Infinispan, Relational Databases and MongoDB

# Chapter 5. Running Keycloak Server on OpenShift

Keycloak provides a OpenShift cartridge to make it easy to get it running on OpenShift. If you don't already have an account or don't know how to create applications go to *https://www.openshift.com/* first. You can create the Keycloak instance either with the web tool or the command line tool, both approaches are described below.

**Warning**

It's important that immediately after creating a Keycloak instance you open the `Administration Console` and login to reset the password. If this is not done anyone can easily gain admin rights to your Keycloak instance.

## 5.1. Create Keycloak instance with the web tool

Open *https://openshift.redhat.com/app/console/applications* and click on `Add Application`. Scroll down to the bottom of the page to find the `Code Anything` section. Insert `http://cartreflect-claytondev.rhcloud.com/github/keycloak/openshift-keycloak-cartridge` into the `URL to a cartridge definition` field and click on `Next`. Fill in the following form and click on `Create Application`.

Click on `Continue to the application overview page`. Under the list of applications you should find your Keycloak instance and the status should be `Started`. Click on it to open the Keycloak servers homepage.

## 5.2. Create Keycloak instance with the command-line tool

Run the following command from a terminal:

```
rhc app create <APPLICATION NAME> http://cartreflect-claytondev.rhcloud.com/github/keycloak/openshift-keycloak-cartridge
```

Replace `<APPLICATION NAME>` with the name you want (for example keycloak).

Once the instance is created the rhc tool outputs details about it. Open the returned `URL` in a browser to open the Keycloak servers homepage.

## 5.3. Next steps

The Keycloak servers homepage shows the Keycloak logo and `Welcome to Keycloak`. There is also a link to the `Administration Console`. Open that and log in using username `admin` and password `admin`. On the first login you are required to change the password.

**Tip**

On OpenShift Keycloak has been configured to only accept requests over https. If you try to use http you will be redirected to https.

# Chapter 6. Master Admin Access Control

You can create and manage multiple realms by logging into the `master` Keycloak admin console at `/{keycloak-root}/admin/index.html`

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 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 have only access to certain operations in specific realms.

## 6.1. Global Roles

There are two realm roles in the `master` realm. These are:

- `admin` - This is the super-user role and grants permissions to all operations on all realms
- `create-realm` - This grants the user permission to create new realms. A user that creates a realm is granted all permissions to the newly created realm.

To add these roles to a user select the `master` realm, then click on `Users`. Find the user you want to grant permissions to, open the user and click on `Role Mappings`. Under `Realm Roles` assign any of the above roles to the user by selecting it and clicking on the right-arrow.

## 6.2. Realm Specific Roles

Each realm in Keycloak is represented by an application in the `master` realm. The name of the application is `<realm name>-realm`. This allows assigning access to users for individual realms. The roles available are:

- `view-realm` - View the realm configuration
- `view-users` - View users (including details for specific user) in the realm
- `view-applications` - View applications in the realm
- `view-clients` - View clients in the realm
- `view-events` - View events in the realm
- `manage-realm` - Modify the realm configuration (and delete the realm)
- `manage-users` - Create, modify and delete users in the realm

- `manage-applications` - Create, modify and delete applications in the realm
- `create-clients` - Create clients in the realm
- `manage-clients` - Create, modify and delete clients in the realm
- `manage-events` - Enable/disable events, clear logged events and manage event listeners

Manage roles includes permissions to view (for example a user with manage-realm role can also view the realm configuration).

To add these roles to a user select the `master` realm, then click on `Users`. Find the user you want to grant permissions to, open the user and click on `Role Mappings`. Under `Application Roles` select the application that represents the realm you're adding permissions to (`<realm name>-realm`), then assign any of the above roles to the user by selecting it and clicking on the right-arrow.

# Chapter 7. Per Realm Admin Access Control

Administering your realm through the `master` realm as discussed in *Chapter 6, Master Admin Access Control* may not always be ideal or feasible. For example, maybe you have more than one admin application that manages various admin aspects of your organization and you want to unify all these different "admin consoles" under one realm so you can do SSO between them. Keycloak allows you to grant realm admin privileges to users within that realm. These realm admins can participate in SSO for that realm and visit a keycloak admin console instance that is dedicated solely for that realm by going to the url: `/{keycloak-root}/admin/{realm}/console`

## 7.1. Realm Roles

Each realm has a built-in application called `realm-management`. This application defines roles that define permissions that can be granted to manage the realm.

- `realm-admin` - This is a composite role that grants all admin privileges for managing security for that realm.

These are more fine-grain roles you can assign to the user.

- `view-realm` - View the realm configuration
- `view-users` - View users (including details for specific user) in the realm
- `view-applications` - View applications in the realm
- `view-clients` - View clients in the realm
- `view-events` - View events in the realm
- `manage-realm` - Modify the realm configuration (and delete the realm)
- `manage-users` - Create, modify and delete users in the realm
- `manage-applications` - Create, modify and delete applications in the realm
- `manage-clients` - Create, modify and delete clients in the realm
- `manage-events` - Enable/disable events, clear logged events and manage event listeners

Manage roles includes permissions to view (for example a user with manage-realm role can also view the realm configuration).

To add these roles to a user select the realm you want. Then click on `Users`. Find the user you want to grant permissions to, open the user and click on `Role Mappings`. Under `Application`

`Roles` select `realm-management`, then assign any of the above roles to the user by selecting it and clicking on the right-arrow.

# Chapter 8. Adapters

Keycloak can secure a wide variety of application types. This section defines which application types are supported and how to configure and install them so that you can use Keycloak to secure your applications.

These client adapters use an extension of the OpenID Connect protocol (a derivate of OAuth 2.0). This extension provides support for clustering, backchannel logout, and other non-standard adminstrative functions. The Keycloak project also provides a separate, standalone, generic, SAML client adapter. But that is describe in a separate document and has a different download.

## 8.1. General Adapter Config

Each adapter supported by Keycloak can be configured by a simple JSON text 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",
  "bearer-only" : false,
  "enable-basic-auth" : false,
  "expose-token" : 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"
}
```

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}`. Also, you can obtain a template for this config file from the admin console. Go to the realm and select the application you want a template for. Go to the `Installation` tab and this will provide you with a template that includes the public key of the realm.

Here is a description of each item:

realm
: Name of the realm representing the users of your distributed applications and services. This is *REQUIRED.*

resource
: Username of the application. Each application has a username that is used when the application connects with the Keycloak server to turn an access code into an access token (part of the OAuth 2.0 protocol). This is *REQUIRED.*

realm-public-key
: PEM format of public key. You can obtain this from the administration console. This is *REQUIRED.*

auth-server-url
: The base URL of the Keycloak Server. All other Keycloak pages and REST services 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 from the adapter is over HTTPS. 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'.

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. 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.

bearer-only
: This tells the adapter to only do bearer token authentication. That is, it will not do OAuth 2.0 redirects, but only accept bearer tokens through the `Authorization` header. This is *OPTIONAL*. 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 only `password` is supported. This is *REQUIRED*.

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. While convenient, this setting is not recommended as you will not be verifying the host name of the Keycloak Server. 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 is not a recommended. This seting may be useful in test environments This is *OPTIONAL*. The default value is `false`.

truststore
: This setting is for Java adapters. The value is the file path to a Java keystore 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 trustore 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 *OPTIONAL* if `ssl-required` is `none` or `disable-trust-manager` is `true`.

truststore-password
: Password for the truststore keystore. This is *REQUIRED* if `truststore` is set.

client-keystore
: *Not supported yet, but we will support in future versions.* This setting is for Java adapters. This is the file path to a Java 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
: *Not supported yet, but we will support in future versions.* Password for the client keystore. This is *REQUIRED* if `client-keystore` is set.

client-key-password
: *Not supported yet, but we will support in future versions.* Password for the client's key. This is *REQUIRED* if `client-keystore` is set.

auth-server-url-for-backend-requests
: Alternative location of auth-server-url used just for backend requests. It must be absolute URI. Useful especially in cluster (see *Relative URI Optimization*) or if you would like to use *https* for browser requests but stick with *http* for backend requests etc.

always-refresh-token
: If *true*, Keycloak will refresh token in every request. More info in *Refresh token in each request* .

register-node-at-startup
: If *true*, then adapter will send registration request to Keycloak. It's *false* by default and useful just in cluster (See *Registration of application nodes to Keycloak*)

register-node-period
: Period for re-registration adapter to Keycloak. Useful in cluster. See *Registration of application nodes to Keycloak* 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 *Stateless token store* for details.

principal-attribute
: OpenID Connection 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 (Tomcat 8, Jetty9, Undertow/Wildfly). Change this to true if you want to turn this off This is *OPTIONAL*. The default value is *false*.

## 8.2. JBoss/Wildfly Adapter

To be able to secure WAR apps deployed on JBoss AS 7.1.1, JBoss EAP 6.x, or Wildfly, you must install and configure the Keycloak Subsystem. You then have two options to secure your WARs. You can provide a keycloak config file in your WAR and change the auth-method to KEYCLOAK within web.xml. Alternatively, you don't have to crack open your WARs at all and can apply Keycloak via the Keycloak Subsystem configuration in standalone.xml. Both methods are described in this section.

### 8.2.1. Adapter Installation

Adapters are no longer included with the appliance or war distribution. Each adapter is a separate download on the Keycloak download site. They are also available as a maven artifact.

Install on Wildfly 9 or 10:

```
$ cd $WILDFLY_HOME
$ unzip keycloak-wildfly-adapter-dist.zip
```

Install on Wildfly 8:

```
$ cd $WILDFLY_HOME
$ unzip keycloak-wf8-adapter-dist.zip
```

Install on JBoss EAP 6.x:

```
$ cd $JBOSS_HOME
$ unzip keycloak-eap6-adapter-dist.zip
```

Install on JBoss AS 7.1.1:

```
$ cd $JBOSS_HOME
```

```
$ unzip keycloak-as7-adapter-dist.zip
```

This zip file creates new JBoss Modules specific to the Wildfly Keycloak Adapter within your Wildfly distro.

After adding the Keycloak modules, you must then enable the Keycloak 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:

```
$ cd $JBOSS_HOME/bin
$ jboss-cli.sh -c --file=adapter-install.cli
```

The script will add the extension, subsystem, and optional security-domain as described below.

For more recent versions of WildFly there's also a offline CLI script that can be used to install the adapter while the server is not running:

```
$ cd $JBOSS_HOME/bin
$ jboss-cli.sh -c --file=adapter-install-offline.cli
```

```
<server xmlns="urn:jboss:domain:1.4">

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

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

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.

## 8.2.2. Required 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.json` adapter config file within the `WEB-INF` directory of your WAR. The format of this config file is describe in the *general 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 pulled from one of the examples that comes distributed with Keycloak.

```
<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-name>
            <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>
```

### 8.2.3. Securing WARs via Keycloak Subsystem

You do not have to crack open a WAR to secure it with Keycloak. Alternatively, 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 XML in your server's `domain.xml` or `standalone.xml` subsystem configuration section.

```
<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>
        <realm-public-key>MIGfMA0GCSqGSIb3DQEBAQUAA</realm-public-key>
        <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 *general adapter configuration*. The exception is the `credential` element.

To make it easier for you, you can go to the Keycloak Adminstration Console and go to the Application/Installation tab of the application this WAR is aligned with. It provides an example XML file you can cut and paste.

There is an additional convenience format for this XML if you have multiple WARs you are deployment that are secured by the same domain. This format allows you to define common configuration items in one place under the `realm` element.

```
<subsystem xmlns="urn:jboss:domain:keycloak:1.1">
    <realm name="demo">
        <realm-public-key>MIGfMA0GCSqGSIb3DQEBA</realm-public-key>
        <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>
```

## 8.3. Tomcat 6, 7 and 8 Adapters

To be able to secure WAR apps deployed on Tomcat 6, 7 and 8 you must install the Keycloak Tomcat 6, 7 or 8 adapter into your Tomcat installation. You then have to provide some extra configuration in each WAR you deploy to Tomcat. Let's go over these steps.

### 8.3.1. Adapter Installation

Adapters are no longer included with the appliance or war distribution. Each adapter is a separate download on the Keycloak download site. They are also available as a maven artifact.

You must unzip the adapter distro into Tomcat's `lib/` directory. Including adapter's jars within your WEB-INF/lib directory will not work! The Keycloak adapter is implemented as a Valve and valve code must reside in Tomcat's main lib/ directory.

```
$ cd $TOMCAT_HOME/lib
$ unzip keycloak-tomcat6-adapter-dist.zip
    or
$ unzip keycloak-tomcat7-adapter-dist.zip
    or
$ unzip keycloak-tomcat8-adapter-dist.zip
```

### 8.3.2. Required 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 `META-INF/context.xml` file in your WAR package. This is a Tomcat specific config file and you must define a Keycloak specific Valve.

```
<Context path="/your-context-path">
   <Valve className="org.keycloak.adapters.tomcat.KeycloakAuthenticatorValve"/>
</Context>
```

Next you must create a `keycloak.json` adapter config file within the `WEB-INF` directory of your WAR. The format of this config file is describe in the *general adapter configuration* section.

Finally you must specify both a `login-config` and 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>customer-portal</module-name>

    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Customers</web-resource-name>
            <url-pattern>/*</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>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>
```

## 8.4. Jetty 9.x Adapters

Keycloak has a separate adapter for Jetty 9.1.x and Jetty 9.2.x that you will have to install into your Jetty installation. You then have to provide some extra configuration in each WAR you deploy to Jetty. Let's go over these steps.

### 8.4.1. Adapter Installation

Adapters are no longer included with the appliance or war distribution.Each adapter is a separate download on the Keycloak download site. They are also available as a maven artifact.

You must unzip the Jetty 9.x distro into Jetty 9.x's root directory. Including adapter's jars within your WEB-INF/lib directory will not work!

```
$ cd $JETTY_HOME
$ unzip keycloak-jetty92-adapter-dist.zip
```

Next, you will have to enable the keycloak module for your jetty.base.

```
$ cd your-base
$ java -jar $JETTY_HOME/start.jar --add-to-startd=keycloak
```

### 8.4.2. Required 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 `WEB-INF/jetty-web.xml` file in your WAR package. This is a Jetty specific config file and you must define a Keycloak specific authenticator within it.

```
<?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.WebAppContext">
    <Get name="securityHandler">
        <Set name="authenticator">
           <New class="org.keycloak.adapters.jetty.KeycloakJettyAuthenticator">
            </New>
        </Set>
    </Get>
</Configure>
```

Next you must create a `keycloak.json` adapter config file within the `WEB-INF` directory of your WAR. The format of this config file is describe in the *general adapter configuration* section.

**Warning**

The Jetty 9.1.x adapter will not be able to find the `keycloak.json` file. You will have to define all adapter settings within the `jetty-web.xml` file as described below.

Instead of using keycloak.json, you can define everything within the `jetty-web.xml`. You'll just have to figure out how the json settings match to the `org.keycloak.representations.adapters.config.AdapterConfig` class.

```
<?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.WebAppContext">
  <Get name="securityHandler">
    <Set name="authenticator">
        <New class="org.keycloak.adapters.jetty.KeycloakJettyAuthenticator">
            <Set name="adapterConfig">
                                                                        <New
 class="org.keycloak.representations.adapters.config.AdapterConfig">
                    <Set name="realm">tomcat</Set>
                    <Set name="resource">customer-portal</Set>
                    <Set name="authServerUrl">http://localhost:8081/auth</Set>
                    <Set name="sslRequired">external</Set>
                    <Set name="credentials">
                        <Map>
                            <Entry>
                                <Item>secret</Item>
                                <Item>password</Item>
                            </Entry>
                        </Map>
                    </Set>
                    <Set name="realmKey">MIGfMA0GCSqGSIb3DQEBAQUAA4</Set>
                </New>
            </Set>
        </New>
    </Set>
  </Get>
</Configure>
```

You do not have to crack open your WAR to secure it with keycloak. Instead create the jetty-web.xml file in your webapps directory with the name of yourwar.xml. Jetty should pick it up. In this mode, you'll have to declare keycloak.json configuration directly within the xml file.

Finally you must specify both a `login-config` and 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>customer-portal</module-name>
```

```
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Customers</web-resource-name>
            <url-pattern>/*</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>BASIC</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>
```

## 8.5. Jetty 8.1.x Adapter

Keycloak has a separate adapter for Jetty 8.1.x that you will have to install into your Jetty installation. You then have to provide some extra configuration in each WAR you deploy to Jetty. Let's go over these steps.

### 8.5.1. Adapter Installation

Adapters are no longer included with the appliance or war distribution.Each adapter is a separate download on the Keycloak download site. They are also available as a maven artifact.

You must unzip the Jetty 8.1.x distro into Jetty 8.1.x's root directory. Including adapter's jars within your WEB-INF/lib directory will not work!

```
$ cd $JETTY_HOME
$ unzip keycloak-jetty81-adapter-dist.zip
```

Next, you will have to enable the keycloak option. Edit start.ini and add keycloak to the options

```
#=============================================================
# Start classpath OPTIONS.
# These control what classes are on the classpath
# for a full listing do
#   java -jar start.jar --list-options
#-------------------------------------------------------------
OPTIONS=Server,jsp,jmx,resources,websocket,ext,plus,annotations,keycloak
```

### 8.5.2. Required Per WAR Configuration

Enabling Keycloak for your WARs is the same as the Jetty 9.x adapter. Our 8.1.x adapter supports both keycloak.json and the jboss-web.xml advanced configuration. See *Required Per WAR Configuration*

## 8.6. Java Servlet Filter Adapter

If you want to use Keycloak 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 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.

**Warning**

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.

```
<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.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>
```

If you notice above, there are two url-patterns. `/protected/*` are just the files we want protected. `/keycloak/*` url-pattern will handle callback from the keycloak server. 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`. There is an example of this in the distribution.

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.

To use this filter, include this maven artifact in your WAR poms

```
          <dependency>
               <groupId>org.keycloak</groupId>
               <artifactId>keycloak-servlet-filter-adapter</artifactId>
               <version>&project.version;</version>
           </dependency>
```

## 8.7. JBoss Fuse and Apache Karaf Adapter

Currently Keycloak supports securing your web applications running inside *JBoss Fuse* [http://www.jboss.org/products/fuse/overview/] or *Apache Karaf* [http://karaf.apache.org/] . It leverages *Jetty 8 adapter* as both JBoss Fuse 6.1 and Apache Karaf 3 are bundled with *Jetty 8.1 server* [http://eclipse.org/jetty/] under the covers and Jetty is used for running various kinds of web applications.

What is supported for Fuse/Karaf is:

- Security for classic WAR applications deployed on Fuse/Karaf with *Pax Web War Extender* [https://ops4j1.jira.com/wiki/display/ops4j/Pax+Web+Extender+-+War].
- Security for servlets deployed on Fuse/Karaf as OSGI services with *Pax Web Whiteboard Extender* [https://ops4j1.jira.com/wiki/display/ops4j/Pax+Web+Extender+-+Whiteboard].

- Security for *Apache Camel* [http://camel.apache.org/] Jetty endpoints running with *Camel Jetty* [http://camel.apache.org/jetty.html] component.
- Security for *Apache CXF* [http://cxf.apache.org/] endpoints running on their own separate *Jetty engine* [http://cxf.apache.org/docs/jetty-configuration.html].
- Security for *Apache CXF* [http://cxf.apache.org/] endpoints running on default engine provided by CXF servlet.
- Security for SSH and JMX admin access.
- Security for *Hawt.io admin console* [http://hawt.io/] .

The best place to start is look at Fuse demo bundled as part of Keycloak examples in directory `examples/fuse` .

## 8.8. Javascript Adapter

The Keycloak Server comes with a Javascript library you can use to secure HTML/Javascript applications. This library is referenceable directly from the keycloak server. You can also download the adapter from Keycloak's download site if you want a static copy. It works in the same way as other application adapters except that your browser is driving the OAuth redirect protocol rather than the server.

The disadvantage of using this approach is that you have a non-confidential, public client. This makes it more important that you register valid redirect URLs and make sure your domain name is secured.

To use this adapter, you must first configure an application (or client) through the `Keycloak Admin Console`. You should select `public` for the `Client Type` field. As public clients can't be verified with a client secret, you are required to configure one or more valid redirect uris. Once you've configured the application, click on the `Installation` tab and download the `keycloak.json` file. This file should be hosted on your web-server at the same root as your HTML pages. Alternatively, you can manually configure the adapter and specify the URL for this file.

Next, you have to initialize the adapter in your application. An example is shown below.

```
<head>
    <script src="http://<keycloak server>/auth/js/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>
```

To specify the location of the keycloak.json file:

```
var keycloak = Keycloak('http://localhost:8080/myapp/keycloak.json'));
```

Or finally to manually configure the adapter:

```
var keycloak = Keycloak({
    url: 'http://keycloak-server/auth',
    realm: 'myrealm',
    clientId: 'myapp'
});
```

You can also pass `login-required` or `check-sso` to the init function. Login required will cause a redirect to the login form on the server, while check-sso will simply redirect to the auth server to check if the user is already logged in to the realm. For example:

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

After you login, your application will be able to make REST calls using bearer token authentication. Here's an example pulled from the `customer-portal-js` example that comes with the distribution.

```
<script>
    var loadData = function () {
        document.getElementById('username').innerText = keycloak.username;

        var url = 'http://localhost:8080/database/customers';

        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) {
                    var users = JSON.parse(req.responseText);
```

```
                    var html = '';
                    for (var i = 0; i < users.length; i++) {
                        html += '<p>' + users[i] + '</p>';
                    }
                    document.getElementById('customers').innerHTML = html;
                    console.log('finished loading data');
                }
            }
        }

        req.send();
    };

    var loadFailure = function () {
          document.getElementById('customers').innerHTML = '<b>Failed to load
 data.  Check console log</b>';

    };

    var reloadData = function () {
        keycloak.updateToken().success(loadData).error(loadFailure);
    }
</script>

<button onclick="reloadData()">Submit</button>
```

The `loadData()` method builds an HTTP request setting the `Authorization` header to a bearer token. The `keycloak.token` points to the access token the browser obtained when it logged you in. The `loadFailure()` method is invoked on a failure. The `reloadData()` function calls `keycloak.updateToken()` passing in the `loadData()` and `loadFailure()` callbacks. The `keycloak.updateToken()` method checks to see if the access token hasn't expired. If it hasn't, and your oauth login returned a refresh token, this method will refresh the access token. Finally, if successful, it will invoke the success callback, which in this case is the `loadData()` method.

To refresh the token when it is expired, call the `updateToken` method. This method returns a promise object, which can be used to invoke a function on success or failure. This method can be used to wrap functions that should only be called with a valid token. For example, the following method will refresh the token if it expires within 30 seconds, and then invoke the specified function. If the token is valid for more than 30 seconds it will just call the specified function.

```
keycloak.updateToken(30).success(function() {
    // send request with valid token
}).error(function() {
    alert('failed to refresh token');
);
```

### 8.8.1. Session status iframe

By default, the JavaScript adapter creates a non-visible 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 from a special status cookie. This feature can be disabled by setting `checkLoginIframe: false` in the options passed to the `init` method.

### 8.8.2. Implicit and Hybrid Flow

By default, the JavaScript adapter uses *OpenID Connect standard (Authorization code) flow* [http://openid.net/specs/openid-connect-core-1_0.html#CodeFlowAuth], which means that after authentication, the Keycloak server redirects the user back to your application, where the JavaScript adapter will exchange the `code` for an access token and a refresh token.

However, Keycloak also supports *OpenID Connect Implicit flow* [http://openid.net/specs/openid-connect-core-1_0.html#ImplicitFlowAuth] where an access token is sent immediately after successful authentication with Keycloak (there is no additional request for exchange code). This could have better performance than standard flow, as there is no additional request to exchange the code for tokens. However, sending the access token in the URL fragment could pose a security issue in some environments (access logs might expose tokens located in the URL).

To enable implicit flow, you need to enable the `Implicit Flow Enabled` flag for the client in the Keycloak admin console. You also need to pass the parameter `flow` with value `implicit` to `init` method. An example is below:

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

Note that with implicit flow, you are not given a refresh token after authentication. This makes it harder for your application to periodically update the access token in background (without browser redirection). It's recommended that you implement an `onTokenExpired` callback method on the keycloak object, so you are notified after the token is expired (For example you can call keycloak.login, which will redirect browser to Keycloak login screen and it will immediately redirect you back if the SSO session is still valid and the user is still logged. However, make sure to save the application state before performing a redirect.)

Keycloak also has support for *OpenID Connect Hybrid flow* [http://openid.net/specs/openid-connect-core-1_0.html#HybridFlowAuth]. 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 security risks might still apply. However, one advantage over the implicit flow is that a refresh token is made available to the application (after the code-to-token request is finished).

For hybrid flow, you need to pass the parameter `flow` with value `hybrid` to `init` method.

### 8.8.3. Older browsers

The JavaScript adapter depends on Base64 (window.btoa and window.atob) and HTML5 History API. If you need to support browsers that don't provide those (for example IE9) you'll need to add polyfillers. Example polyfill libraries:

- Base64 - *https://github.com/davidchambers/Base64.js*
- HTML5 History - *https://github.com/devote/HTML5-History-API*

### 8.8.4. JavaScript Adapter reference

#### 8.8.4.1. Constructor

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

#### 8.8.4.2. Properties

- authenticated - true if the user is authenticated
- token - the base64 encoded token that can be sent in the `Authorization` header in requests to services
- tokenParsed - the parsed token
- subject - the user id
- idToken - the id token if claims is enabled for the application, null otherwise
- idTokenParsed - the parsed id token
- realmAccess - the realm roles associated with the token
- resourceAccess - the resource roles assocaited with the token
- refreshToken - the base64 encoded token that can be used to retrieve a new token
- refreshTokenParsed - the parsed refresh token
- timeSkew - estimated skew between local time and Keycloak server in seconds

- responseMode - responseMode passed during initialization. See below for details. Default value is `fragment`
- flow - OpenID Connect flow passed during initialization. See *Implicit flow* for details.
- responseType - responseType used for send to Keycloak server at login request. This is determined based on the `flow` value used during initialization, but you have possibility to override it by directly set this value

### 8.8.4.3. Methods

#### init(options)

Called to initialize the adapter.

Options is an Object, where:

- onLoad - specifies an action to do on load, can be either '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 `fragment` . 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`. See *Implicit flow* for details.

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 - can be set to 'none' to check if the user is logged in already (if not logged in, a login form is not displayed)
- loginHint - used to pre-fill the username/email field on the login form
- action - if value is 'register' then user is redirected to registration page, otherwise to login page
- locale - specifies the desired locale for the UI

### createLoginUrl(options)

Returns the url 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 - can be set to 'none' to check if the user is logged in already (if not logged in, a login form is not displayed)

### logout(options)

Redirects to logout

Options is an Object, where:

- redirectUri - specifies the uri to redirect to after logout

### createLogoutUrl(options)

Returns logout out

Options is an Object, where:

- redirectUri - specifies the uri to redirect to after logout

### register(options)

Redirects to registration form. It's a shortcut for doing login with option action = 'register'

Options are same as login method but 'action' is overwritten to 'register'

### createRegisterUrl(options)

Returns the url to registration page. It's a shortcut for doing createRegisterUrl with option action = 'register'

Options are same as createLoginUrl method but 'action' is overwritten to 'register'

### accountManagement()

Redirects to account management

### createAccountUrl()

Returns the url to account management

### 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 on success or error.

### isTokenExpired(minValidity)

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

### updateToken(minValidity)

If the token expires within minValidity seconds (minValidity is optional, if not specified 0 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 has expired, for example if updating token fails. Invoking this results in onAuthLogout callback listener being invoked.

```
keycloak.updateToken(5).error(function() {
    keycloak.clearToken();
});
```

#### 8.8.4.4. Callback Events

The adapter supports setting callback listeners for certain events. For example:

```
keycloak.onAuthSuccess = function() { alert('authenticated'); }
```

- 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 access token expired. When this happens you can for example refresh token, or if refresh not available (ie. with implicit flow) you can redirect to login screen

## 8.9. 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.

### 8.9.1. 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 adapter JAR to your project. Depending on what container you are using with Spring Boot, you also need to add the appropriate Keycloak container adapter. If you are using Maven, add the following to your pom.xml (using Tomcat as an example):

```
<dependency>
    <groupId>org.keycloak</groupId>
```

```
    <artifactId>keycloak-spring-boot-adapter</artifactId>
    <version>&project.version;</version>
</dependency>
<dependency>
    <groupId>org.keycloak</groupId>
    <artifactId>keycloak-tomcat8-adapter</artifactId>
    <version>&project.version;</version>
</dependency>
```

### 8.9.2. 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.realmKey     =     MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCLCWYuxXmsmfV
+Xc9Ik8QET8lD4wuHrJAXbbutS2O/eMjQQLNK7QDX/k/
XhOkhxP0YBEypqeXeGaeQJjCxDhFjJXQuewUEMlmSja3IpoJ9/hFn4Cns4m7NGO
+rtvnfnwgVfsEOS5EmZhRddp+40KBPPJfTH6Vgu6KjQwuFPj6DTwIDAQAB
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 also need to specify the J2EE security config that would normally go in the `web.xml`. Here's an example configuration:

```
keycloak.securityConstraints[0].securityCollections[0].name = insecure stuff
keycloak.securityConstraints[0].securityCollections[0].authRoles[0] = admin
keycloak.securityConstraints[0].securityCollections[0].authRoles[0] = user
keycloak.securityConstraints[0].securityCollections[0].patterns[0] = /insecure

keycloak.securityConstraints[0].securityCollections[1].name = admin stuff
keycloak.securityConstraints[0].securityCollections[1].authRoles[0] = admin
keycloak.securityConstraints[0].securityCollections[1].patterns[0] = /admin
```

## 8.10. Spring Security Adapter

To secure an application with Spring Security and Keycloak, add this adapter as a dependency to your project. You then have to provide some extra beans in your Spring Security configuration file and add the Keycloak security filter to your pipeline.

Unlike the other Keycloak Adapters, you should not configure your security in web.xml. However, keycloak.json is still required.

### 8.10.1. Adapter Installation

Add Keycloak Spring Security adapter as a dependency to your Maven POM or Gradle build.

```
<dependency>
    <groupId>org.keycloak</groupId>
    <artifactId>keycloak-spring-security-adapter</artifactId>
    <version>&project.version;</version>
</dependency>
```

### 8.10.2. Spring Security Configuration

The Keycloak Spring Security adapter takes advantage of Spring Security's flexible security configuration syntax.

#### 8.10.2.1. Java Configuration

Keycloak provides a KeycloakWebSecurityConfigurerAdapter as a convenient base class for creating a *WebSecurityConfigurer* [http://docs.spring.io/spring-security/site/docs/4.0.x/apidocs/org/springframework/security/config/annotation/web/WebSecurityConfigurer.html] instance. The implementation allows customization by overriding methods. While its use is not required, it greatly simplifies your security context configuration.

```
@Configuration
@EnableWebSecurity
@ComponentScan(basePackageClasses = KeycloakSecurityComponents.class)
```

```
public class SecurityConfig extends KeycloakWebSecurityConfigurerAdapter
{
    /**
       * Registers the KeycloakAuthenticationProvider with the authentication
 manager.
     */
    @Autowired
       public void configureGlobal(AuthenticationManagerBuilder auth) throws
 Exception {
        auth.authenticationProvider(keycloakAuthenticationProvider());
    }

    /**
     * Defines the session authentication strategy.
     */
    @Bean
    @Override
    protected SessionAuthenticationStrategy sessionAuthenticationStrategy() {
                      return   new   RegisterSessionAuthenticationStrategy(new
 SessionRegistryImpl());
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception
    {
        super.configure(http);
        http
                .authorizeRequests()
                .antMatchers("/customers*").hasRole("USER")
                .antMatchers("/admin*").hasRole("ADMIN")
                .anyRequest().permitAll();
    }
}
```

You must provide a session authentication strategy bean which should be of type `RegisterSessionAuthenticationStrategy` for public or confidential applications and `NullAuthenticatedSessionStrategy` for bearer-only applications.

Spring Security's `SessionFixationProtectionStrategy` is currently not supported because it changes the session identifier after login via Keycloak. If the session identifier changes, universal log out will not work because Keycloak is unaware of the new session identifier.

### 8.10.2.2. XML Configuration

While Spring Security's XML namespace simplifies configuration, customizing the configuration can be a bit verbose.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:security="http://www.springframework.org/schema/security"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd
       http://www.springframework.org/schema/security
       http://www.springframework.org/schema/security/spring-security.xsd">

                                          <context:component-scan          base-
package="org.keycloak.adapters.springsecurity" />

    <security:authentication-manager alias="authenticationManager">
      <security:authentication-provider ref="keycloakAuthenticationProvider" />
    </security:authentication-manager>

                                       <bean         id="adapterDeploymentContext"
 class="org.keycloak.adapters.springsecurity.AdapterDeploymentContextFactoryBean">
        <constructor-arg value="/WEB-INF/keycloak.json" />
    </bean>

                                   <bean       id="keycloakAuthenticationEntryPoint"
class="org.keycloak.adapters.springsecurity.authentication.KeycloakAuthenticationEntryPoint"
>
                                     <bean        id="keycloakAuthenticationProvider"
  class="org.keycloak.adapters.springsecurity.authentication.KeycloakAuthenticationProvider"
>
                                       <bean        id="keycloakPreAuthActionsFilter"
          class="org.keycloak.adapters.springsecurity.filter.KeycloakPreAuthActionsFilter"
>
                             <bean      id="keycloakAuthenticationProcessingFilter"
 class="org.keycloak.adapters.springsecurity.filter.KeycloakAuthenticationProcessingFilter">
                                   <constructor-arg    name="authenticationManager"
 ref="authenticationManager" />
    </bean>

                                         <bean         id="keycloakLogoutHandler"
 class="org.keycloak.adapters.springsecurity.authentication.KeycloakLogoutHandler">
        <constructor-arg ref="adapterDeploymentContext" />
    </bean>
```

```
                                                <bean           id="logoutFilter"
 class="org.springframework.security.web.authentication.logout.LogoutFilter">
        <constructor-arg name="logoutSuccessUrl" value="/" />
        <constructor-arg name="handlers">
            <list>
                <ref bean="keycloakLogoutHandler" />
                                                                                <bean
class="org.springframework.security.web.authentication.logout.SecurityContextLogoutHandler"
>
            </list>
        </constructor-arg>
        <property name="logoutRequestMatcher">
                                                                                <bean
 class="org.springframework.security.web.util.matcher.AntPathRequestMatcher">
                <constructor-arg name="pattern" value="/sso/logout**" />
                <constructor-arg name="httpMethod" value="GET" />
            </bean>
        </property>
    </bean>

                      <security:http     auto-config="false"     entry-point-
ref="keycloakAuthenticationEntryPoint">
                   <security:custom-filter   ref="keycloakPreAuthActionsFilter"
 before="LOGOUT_FILTER" />
          <security:custom-filter ref="keycloakAuthenticationProcessingFilter"
 before="FORM_LOGIN_FILTER" />
        <security:intercept-url pattern="/customers**" access="ROLE_USER" />
        <security:intercept-url pattern="/admin**" access="ROLE_ADMIN" />
        <security:custom-filter ref="logoutFilter" position="LOGOUT_FILTER" />
    </security:http>

</beans>
```

### 8.10.3. Multi Tenancy

The Keycloak Spring Security adapter also supports multi tenancy. Instead of injecting `AdapterDeploymentContextFactoryBean` with the path to `keycloak.json` you can inject an implementation of the `KeycloakConfigResolver` interface. More details on how to implement the `KeycloakConfigResolver` can be found in *Section 8.14, "Multi Tenancy"*.

### 8.10.4. Naming Security Roles

Spring Security, when using role-based authentication, requires that role names start with `ROLE_`. For example, an administrator role must be declared in Keycloak as `ROLE_ADMIN` or similar, not simply `ADMIN`.

The class `org.keycloak.adapters.springsecurity.authentication.KeycloakAuthenticationProvider` supports an optional `org.springframework.security.core.authority.mapping.GrantedAuthoritiesMapper` which can be used to map roles coming from Keycloak to roles recognized by Spring Security. Use, for example, `org.springframework.security.core.authority.mapping.SimpleAuthorityMapper` to insert the `ROLE_` prefix and convert the role name to upper case. The class is part of Spring Security Core module.

## 8.10.5. Client to Client Support

To simplify communication between clients, Keycloak provides an extension of Spring's `RestTemplate` that handles bearer token authentication for you. To enable this feature your security configuration must add the `KeycloakRestTemplate` bean. Note that it must be scoped as a prototype to function correctly.

For Java configuration:

```
@Configuration
@EnableWebSecurity
@ComponentScan(basePackageClasses = KeycloakSecurityComponents.class)
public class SecurityConfig extends KeycloakWebSecurityConfigurerAdapter {

    ...

    @Autowired
    public KeycloakClientRequestFactory keycloakClientRequestFactory;

    @Bean
    @Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
    public KeycloakRestTemplate keycloakRestTemplate() {
        return new KeycloakRestTemplate(keycloakClientRequestFactory);
    }

    ...
}
```

For XML configuration:

```
```

```
<bean                                                    id="keycloakRestTemplate"

 scope="prototype">
    <constructor-arg name="factory" ref="keycloakClientRequestFactory" />
</bean>
```

Your application code can then use `KeycloakRestTemplate` any time it needs to make a call to another client. For example:

```
@Service
public class RemoteProductService implements ProductService {

    @Autowired
    private KeycloakRestTemplate template;

    private String endpoint;

    @Override
    public List<String> getProducts() {
           ResponseEntity<String[]> response = template.getForEntity(endpoint,
 String[].class);
        return Arrays.asList(response.getBody());
    }
}
```

## 8.10.6. Spring Boot Configuration

Spring Boot attempts to eagerly register filter beans with the web application context. Therefore, when running the Keycloak Spring Security adapter in a Spring Boot environment, it may be necessary to add two `FilterRegistrationBean`s to your security configuration to prevent the Keycloak filters from being registered twice.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends KeycloakWebSecurityConfigurerAdapter
```

```
{
    ...

    @Bean
                                        public          FilterRegistrationBean
 keycloakAuthenticationProcessingFilterRegistrationBean(
            KeycloakAuthenticationProcessingFilter filter) {
                            FilterRegistrationBean   registrationBean   =   new
 FilterRegistrationBean(filter);
        registrationBean.setEnabled(false);
        return registrationBean;
    }

    @Bean
    public FilterRegistrationBean keycloakPreAuthActionsFilterRegistrationBean(
            KeycloakPreAuthActionsFilter filter) {
                            FilterRegistrationBean   registrationBean   =   new
 FilterRegistrationBean(filter);
        registrationBean.setEnabled(false);
        return registrationBean;
    }

    ...
}
```

## 8.11. Installed Applications

Keycloak provides two special redirect uris for installed applications.

### 8.11.1. http://localhost

This returns the code to a web server on the client as a query parameter. Any port number is allowed. This makes it possible to start a web server for the installed application on any free port number without requiring changes in the `Admin Console`.

### 8.11.2. 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.

## 8.12. 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 the url `http://auth-server/auth/realms/{realm-name}/tokens/logout?redirect_uri=encodedRedirectUri`. This will log you out if you have an SSO session with your browser.

## 8.13. 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 may throw 400, 401, 403, and 500 errors.

```
<error-page>
    <error-code>404</error-code>
    <location>/ErrorHandler</location>
</error-page>
```

Keycloak 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.OIDCAuthenticationError`. 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 OIDCAuthenticationError implements AuthenticationError {
    public static enum Reason {
        NO_BEARER_TOKEN,
        NO_REDIRECT_URI,
        INVALID_STATE_COOKIE,
        OAUTH_ERROR,
        SSL_REQUIRED,
        CODE_TO_TOKEN_FAILURE,
        INVALID_TOKEN,
        STALE_TOKEN,
        NO_AUTHORIZATION_HEADER
    }

    private Reason reason;
    private String description;

    public OIDCAuthenticationError(Reason reason, String description) {
```

```
        this.reason = reason;
        this.description = description;
    }

    public Reason getReason() {
        return reason;
    }

    public String getDescription() {
        return description;
    }
}
```

## 8.14. Multi Tenancy

Multi Tenancy, in our context, means that one single target application (WAR) can be secured by a single (or clustered) Keycloak server, authenticating its users against different realms. In practice, this means that one application needs to use different `keycloak.json` files. For this case, there are two possible solutions:

- The same WAR file deployed under two different names, each with its own Keycloak configuration (probably via the Keycloak Subsystem). This scenario is suitable when the number of realms is known in advance or when there's a dynamic provision of application instances. One example would be a service provider that dynamically creates servers/deployments for their clients, like a PaaS.
- A WAR file deployed once (possibly in a cluster), that decides which realm to authenticate against based on the request parameters. This scenario is suitable when there are an undefined number of realms. One example would be a SaaS provider that have only one deployment (perhaps in a cluster) serving several companies, differentiating between clients based on the hostname (`client1.acme.com`, `client2.acme.com`) or path (`/app/client1/`, `/app/client2/`) or even via a special HTTP Header.

This chapter of the reference guide focus on this second scenario.

Keycloak provides an extension point for applications that need to evaluate the realm on a request basis. During the authentication and authorization phase of the incoming request, Keycloak queries the application via this extension point and expects the application to return a complete representation of the realm. With this, Keycloak then proceeds the authentication and authorization process, accepting or refusing the request based on the incoming credentials and on the returned realm. For this scenario, an application needs to:

- Add a context parameter to the `web.xml`, named `keycloak.config.resolver`. The value of this property should be the fully qualified name of the class extending `org.keycloak.adapters.KeycloakConfigResolver`.

- A concrete implementation of `org.keycloak.adapters.KeycloakConfigResolver`. Keycloak will call the `resolve(org.keycloak.adapters.spi.HttpFacade.Request)` method and expects a complete `org.keycloak.adapters.KeycloakDeployment` in response. Note that Keycloak will call this for every request, so, take the usual performance precautions.

An implementation of this feature can be found in the examples.

## 8.15. JAAS plugin

It's generally not needed to use JAAS for most of the applications, especially if they are HTTP based, but directly choose one of our adapters. However some applications and systems may still rely on pure legacy JAAS solution. Keycloak provides couple of login modules to help with such use cases. Some login modules provided by Keycloak are:

org.keycloak.adapters.jaas.DirectAccessGrantsLoginModule
: This login module allows to authenticate with username/password from Keycloak database. It's using *Direct Access Grants* Keycloak endpoint to validate on Keycloak side if provided username/password is valid. It's useful especially for non-web based systems, which need to rely on JAAS and want to use Keycloak credentials, but can't use classic browser based authentication flow due to their non-web nature. Example of such application could be messaging application or SSH system.

org.keycloak.adapters.jaas.BearerTokenLoginModule
: This login module allows to authenticate with Keycloak access token passed to it through CallbackHandler as password. It may be useful for example in case, when you have Keycloak access token from classic web based authentication flow and your web application then needs to talk to external non-web based system, which rely on JAAS. For example to JMS/messaging system.

Both login modules have configuration property `keycloak-config-file` where you need to provide location of keycloak.json configuration file. It could be either provided from filesystem or from classpath (in that case you may need value like `classpath:/folder-on-classpath/keycloak.json` ).

Second property `role-principal-class` allows to specify alternative class for Role principals attached to JAAS Subject. Default value for Role principal is `org.keycloak.adapters.jaas.RolePrincipal` . Note that class should have constructor with single String argument.

# Chapter 9. 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 Keycloak's client registration service.

The Client Registration Service provides built-in support for Keycloak Client Representations, OpenID Connect Client Meta Data and SAML Entity Descriptors. It's also possible to plugin custom client registration providers if required. The Client Registration Service endpoint is `<KEYCLOAK URL>/realms/<realm>/clients-registrations/<provider>`.

The built-in supported `providers` are:

- `default` Keycloak Representations
- `install` Keycloak Adapter Configuration
- `openid-connect` OpenID Connect Dynamic Client Registration
- `saml2-entity-descriptor` SAML Entity Descriptors

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

## 9.1. Authentication

To invoke the Client Registration Services you need a token. The token can be a standard bearer token, a initial access token or a registration access token.

### 9.1.1. Bearer Token

The bearertoken can be issued on behalf of a user or a Service Account. The following permissions are required to invoke the endpoints (see *Admin Permissions* 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 clients

If you are using a regular bearer token to create clients we recommend using a token from on behalf of a Service Account with only the `create-client` role. See the *Service Account* section for more details.

### 9.1.2. Initial Access Token

The best approach to create 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 `Initial Access Tokens` in the tabs displayed in the page.

You will now be able to see any existing initial access tokens. If you have access you can delete tokens that are no longer required. You can only retrieve the value of the token when you are creating it. To create a new token click on `Create`. You can now optionally add how long the token should be valid, also how many clients can be created using the token. After you click on `Save` the token value is displayed. It is important that you copy/paste this token now as you won't be able to retrieve it later. If you forget to copy/paste it, then delete the token and create another one. The token value is used as a standard bearer token when invoking the Client Registration Services, by adding it to the Authorization header in the request. For example:

```
Authorization:                                                    bearer
 eyJhbGciOiJSUzI1NiJ9.eyJqdGkiOiJmMjJmNzQyYy04ZjNlLTQ2M....
```

### 9.1.3. 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 loose 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`.

## 9.2. Keycloak Representations

The `default` client registration provider can be used to create, retrieve, update and delete a client. It uses Keycloaks 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 do a HTTP POST to: `<KEYCLOAK URL>/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 then do a HTTP GET to: `<KEYCLOAK URL>/realms/<realm>/clients-registrations/default/<client id>`. It will also return a new registration access token.

To update the Client Representation then do a HTTP PUT to with the updated Client Representation to: `<KEYCLOAK URL>/realms/<realm>/clients-registrations/default/<client id>`. It will also return a new registration access token.

To delete the Client Representation then do a HTTP DELETE to: `<KEYCLOAK URL>/realms/<realm>/clients-registrations/default/<client id>`

## 9.3. 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-secret)
```

To retrieve the Adapter Configuration then do a HTTP GET to: `<KEYCLOAK URL>//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.

## 9.4. OpenID Connect Dynamic Client Registration

Keycloak implements *OpenID Connect Dynamic Client Registration* [https://openid.net/specs/openid-connect-registration-1_0.html], which extends *OAuth 2.0 Dynamic Client Registration Protocol* [https://tools.ietf.org/html/rfc7591] and *OAuth 2.0 Dynamic Client Registration Management Protocol* [https://tools.ietf.org/html/rfc7592].

The endpoint to use these specifications to register clients in Keycloak is: `<KEYCLOAK URL>/realms/<realm>/clients-registrations/oidc[/<client id>]`.

This endpoints can also be found in the OpenID Connect Discovery endpoint for the realm: `<KEYCLOAK URL>/realms/<realm>/.well-known/openid-configuration`.

## 9.5. 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 do a HTTP POST with the SAML Entity Descriptor to: `<KEYCLOAK URL>/realms/<realm>/clients-registrations/saml2-entity-descriptor`.

## 9.6. Client Registration Java API

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

For full instructions on using the Client Registration refer to the JavaDocs. Below is an example of creating a client:

```
String                            initialAccessToken                            =
 "eyJhbGciOiJSUzI1NiJ9.eyJqdGkiOiJmMjJmNzQyYy04ZjNlLTQ2M....";

ClientRepresentation client = new ClientRepresentation();
client.setClientId(CLIENT_ID);

ClientRegistration reg = ClientRegistration.create().url("http://keycloak/auth/
realms/myrealm/clients").build();
reg.auth(Auth.token(initialAccessToken));

client = reg.create(client);

String registrationAccessToken = client.getRegistrationAccessToken();
```

# Chapter 10. Identity Broker

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 to create a trust relationship with an external identity provider in order to use its identities to access internal services exposed by service providers.

From an user perspective, an identity broker provides an user-centric and centralized way to manage identities across different security domains or realms, where 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 in order to authenticate and communicate authentication and authorization information to their users. It can be a social provider such as Facebook, Google or Twitter, a business partner which you want to allow its users to access your services or a cloud-based identity service that you want to integrate with. Usually, identity providers are based on the following protocols:

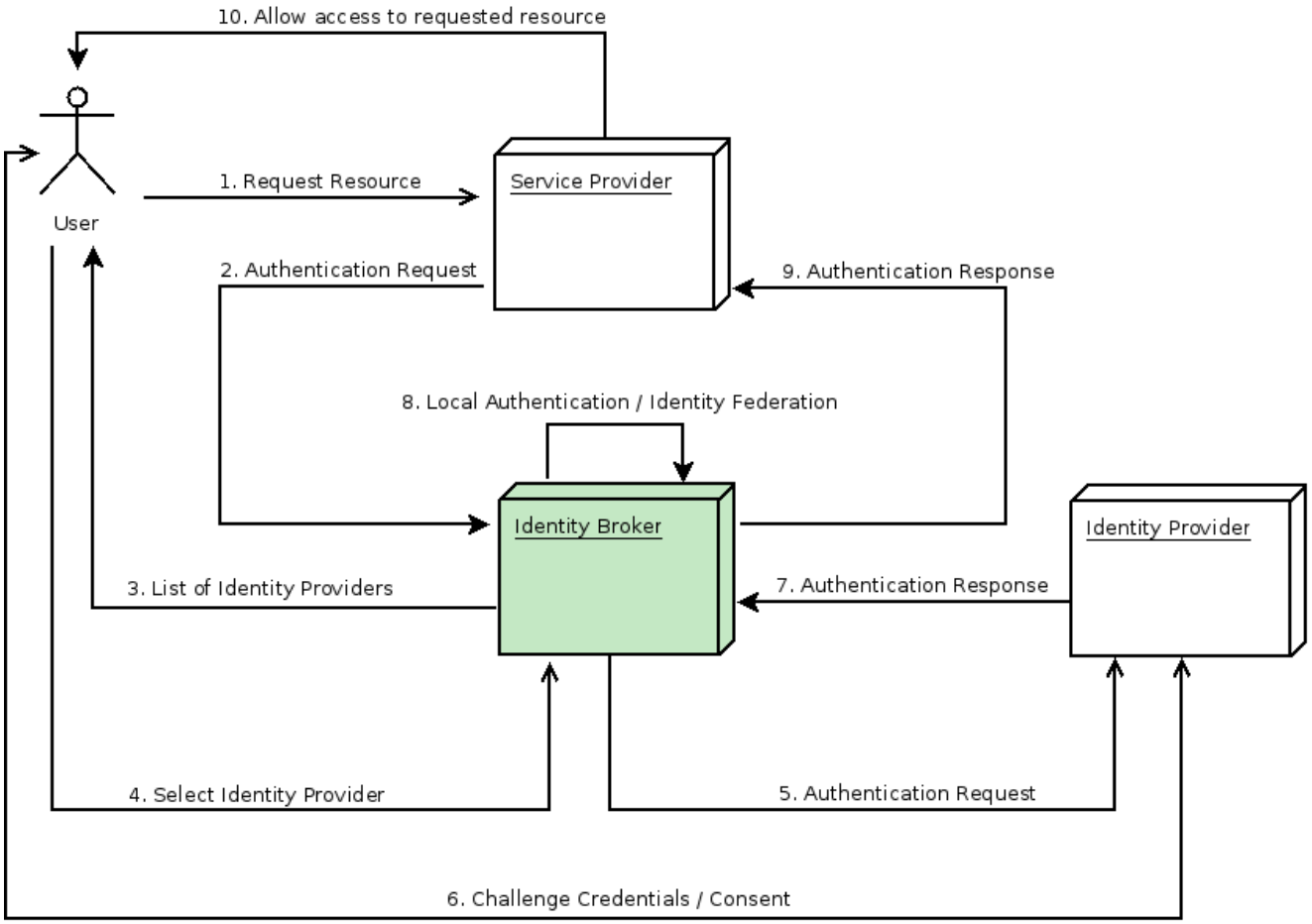
- `SAML v2.0`
- `OpenID Connect v1.0`
- `oAuth v2.0`

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`

## 10.1. 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 of that, they are presented with a list of identity providers from where they can pick one and authenticate. You can also configure a hard-coded default broker. In this case the user will not be given a choice, but instead be redirected directly the the parent broker. The following diagram demonstrates the steps involved when using Keycloak to broker an external identity provider:




1. User is not authenticated and requests a protected resource in a service provider.
2. The service provider redirects the user to Keycloak to authenticate.
3. At this point the user is presented to the login page where there is a list of identity providers supported by 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 redirect to the login page or just to a consent page in 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 in 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 create an user or just skip that if the user already exists. If it is a new user, Keycloak may

ask informations about the user to the identity provider (or just read that from a security token) and create the user locally. 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 his existing account. A process that we call *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 present a list of identity providers, the service provider can automatically select a specific one to authenticate an user. Or you can tell Keycloak to force the user to provide additional information before federate his identity.

**Note**

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, despite 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 service providers, what means that service providers 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 !

## 10.2. 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, follow these steps:

1. In the admin console, select a realm.
2. On the left side menu, click on `Settings`.
3. Select the `Identity Provider` tab on the settings page.
4. You should now be able to see a table that lists all registered identity providers. To add a new identity provider, click the select box on the top right of this table and select which type of identity provider you want to create.

Identity providers are organized in two main categories:

Social
: Social providers allows you to enable social authentication to your realm. Keycloak makes it easy to let users log in to your application using an existing account with a social network. Currently Facebook, Google, Twitter, GitHub, LinkedIn, Microsoft and StackOverflow are supported with more planned for the future.

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 the identity provider you are creating, you'll see the following configuration options avaiable:

### Table 10.1. Common Configuration

| Configuration | Description |
|---|---|
| `Alias` | The alias is an unique identifier for an identity provider. It is used to reference internally an identity provider. Some protocols require a *redirect uri or callback url* in order to communicate with an identity provider. For instance, OpenID Connect. In this case, the alias is used to build the redirect uri. Every single identity provider must have an alias. For example, facebook, google, idp.acme.com, etc. |
| `Name` | You can give a friendly name to an identity provider. The name will be used, for example, to display a link or a button in the login page. |
| `Enabled` | Allows you to enable or disable an identity provider. When disabled, the identity provider will not be available from the login page and can not be used by any other means. |
| `Authenticate By Default` | If enabled, Keycloak will automatically redirect to this identity provider even before displaying login screen. In other words, steps 3 and 4 from *the base flow* are skipped. |
| `Store Tokens` | Any external tokens provided by the parent IDP will be stored. This options is useful if you |

| Configuration | Description |
| --- | --- |
| | are using social authentication and need to access the token in order to invoke the API of a social provider on behalf of the user. |
| `Stored Tokens Readable` | Automatically assigns a `broker.read-token` role that allows the user to access any stored external tokens via the broker service. |
| `Trust email` | Allows you to trust email address returned from the social provider. If enabled then email address returned by the provider is marked as 'verified' in the Keycloak user profile. This means that email verification step is skipped even if "Verify email" feature is enabled in realm settings. |
| `GUI order` | Allows you to define order of the provider when shown on login page. You can put number into this field, providers with lower numbers are shown first. |
| `First Login Flow` | Alias of authentication flow, which is triggered during first login with this identity provider. Term `First Login` means that there is not yet existing Keycloak account linked with the authenticated identity provider account. More details in *First Login section*. |
| `Post Login Flow` | Alias of authentication flow, which is triggered after each login with this identity provider. Useful if you want additional verification of each user authenticated with this identity provider (for example OTP). Leave this empty if you don't want any additional authenticators to be triggered after login with this identity provider. Also note, that authenticator implementations must assume that user is already set in ClientSession as identity provider already set it. |

On the next sections, we'll see how to configure each type of identity provider individually.

## 10.3. Social Identity Providers

Forcing users to register to your realm when they want to access applications is hard. So is trying to remember yet another username and password combination. Social identity providers makes it easy for users to register on your realm and quickly sign in using a social network.

Keycloak provides built-in support for the most common social networks out there, such as Google, Facebook, Twitter, Github, LinkedId, Microsoft and StackOverflow.

### 10.3.1. Google

To enable login with Google you first have to create a project and a client in the *Google Developer Console* [https://cloud.google.com/console/project]. Then you need to copy the client id and secret into the Keycloak Admin Console.

Let's see first how to create a project with Google.

1. Log in to the *Google Developer Console* [https://cloud.google.com/console/project]. 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).
2. Once the project has been created click on `APIs & auth` in sidebar on the left. To retrieve user profiles the `Google+ API` has to be enabled. Scroll down to find it in the list. If its status is `OFF`, click on `OFF` to enable it (it should move to the top of the list and the status should be `ON`).
3. Now click on the `Consent screen` link on the sidebar menu on the left. You must specify a project name and choose an email for the consent screen. Otherwise users will get a login error. There's other things you can configure here like what the consent screen looks like. Feel free to play around with this.
4. Now click `Credentials` in the sidebar on the left. Then click `Create New Client ID`. Select `Web application` as `Application type`. Empty the `Authorized Javascript origins` textarea. Click the `Create Client ID` button.
5. Copy `Client ID` and `Client secret`.

Now that you have the client id and secret, you can proceed with the creation of a Google Identity Provider in Keycloak. As follows:

1. Select the `Google` identity provider from the drop-down box on the top right corner of the identity providers table in Keycloak's Admin Console. You should be presented with a specific page to configure the selected provided.
2. Copy the client id and secret to their corresponding fields in the Keycloak Admin Console. Click `Save`.

Once you create the identity provider in Keycloak, you must update your Google project with the redirect url that was generated to your identity provider.

1. Open the Google Developer Console and select your project. Click `Credentials` in the sidebar on the left. In `Authorized redirect URI` insert the redirect uri created by Keycloak. The redirect uri usually have the following format: `http://{host}:{port}/auth/realms/{realm}/broker/{provider_alias}`.

**Note**

You can always get the redirect url for a specific identity provider from the table presented when you click on the 'Identity Provider' tab in *Realm > Settings*.

That is it! This is pretty much what you need to do in order to setup this identity provider.

The table below lists some additional configuration options you may use when configuring this provider.

**Table 10.2. Configuration Options**

| Configuration | Description |
|---|---|
| `Default Scopes` | 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` |

## 10.3.2. Facebook

To enable login with Facebook you first have to create an application in the *Facebook Developer Console* [https://developers.facebook.com/]. Then you need to copy the client id and secret into the Keycloak Admin Console.

Let's see first how to create an application with Facebook.

1. Log in to the *Facebook Developer Console* [https://developers.facebook.com/]. Click `Apps` in the menu and select `Create a New App`. Use any value for `Display Name` and `Category` you want, then click the `Create App` button. Wait for the project to be created (this may take a while). If after creating the app you are not redirected to the app settings, click on `Apps` in the menu and select the app you created.

2. Once the app has been created click on `Settings` in sidebar on the left. You must specify a contact email. Save your changes. Then click on `Advanced`. Under `Security` make sure `Client OAuth Login` is enabled. Scroll down and click on the `Save Changes` button.

3. Click `Status & Review` and select `YES` for `Do you want to make this app and all its live features available to the general public?`. You will not be able to set this until you have provided a contact email in the general settings of this application.

4. Click `Basic`. Copy `App ID` and `App Secret` (click `show`) from the *Facebook Developer Console* [https://developers.facebook.com/].

Now that you have the client id and secret, you can proceed with the creation of a Facebook Identity Provider in Keycloak. As follows:

1. Select the `Facebook` identity provider from the drop-down box on the top right corner of the identity providers table in Keycloak's Admin Console. You should be presented with a specific page to configure the selected provided.
2. Copy the client id and secret to their corresponding fields in the Keycloak Admin Console. Click `Save`.

Once you create the identity provider in Keycloak, you must update your Facebook application with the redirect url that was generated to your identity provider.

1. Open the Facebook Developer Console and select your application. Click on `Advanced`. Under `Security` make sure `Client OAuth Login` is enabled. In `Valid OAuth redirect URIs` insert the redirect uri created by Keycloak. The redirect uri usually have the following format: `http://{host}:{port}/auth/realms/{realm}/broker/{provider_alias}`.

**Note**

You can always get the redirect url for a specific identity provider from the table presented when you click on the 'Identity Provider' tab in *Realm > Settings*.

That is it! This pretty much what you need to do in order to setup this identity provider.

The table below lists some additional configuration options you may use when configuring this provider.

### Table 10.3. Configuration Options

| Configuration | Description |
|---|---|
| `Default Scopes` | 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` |

## 10.3.3. Twitter

To enable login with Twtter you first have to create an application in the *Twitter Developer Console* [https://dev.twitter.com/apps]. Then you need to copy the consumer key and secret into the Keycloak Admin Console.

Let's see first how to create an application with Twitter.

1. Log in to the *Twitter Developer Console* [https://dev.twitter.com/apps]. Click the `Create a new application` button. Use any value for `Name`, `Description` and `Website` you want. Insert the social callback url in `Callback URL`. Then click `Create your Twitter application`.
2. Now click on `Settings` and tick the box `Allow this application to be used to Sign in with Twitter`, then click on `Update this Twitter application's settings`.
3. Now click `API Keys` tab. Copy `API key` and `API secret` from the *Twitter Developer Console* [https://dev.twitter.com/apps].

**Note**

Twitter doesn't allow `localhost` in the redirect URI. To test on a local server replace `localhost` with `127.0.0.1`.

Now that you have the client id and secret, you can proceed with the creation of a Twitter Identity Provider in Keycloak. As follows:

1. Select the `Twitter` identity provider from the drop-down box on the top right corner of the identity providers table in Keycloak's Admin Console. You should be presented with a specific page to configure the selected provided.
2. Copy the client id and secret to their corresponding fields in the Keycloak Admin Console. Click `Save`.

That is it! This pretty much what you need to do in order to setup this identity provider.

The table below lists some additional configuration options you may use when configuring this provider.

**Table 10.4. Configuration Options**

| Configuration | Description |
|---|---|
| `Default Scopes` | Not supported by Twitter. |

## 10.3.4. Github

To enable login with GitHub you first have to create an application in *GitHub Settings* [https://github.com/settings/applications]. Then you need to copy the client id and secret into the Keycloak Admin Console.

Let's see first how to create an application with GitHub.

1. Log in to *GitHub Settings* [https://github.com/settings/applications]. Click the `Register new application` button. Use any value for `Application name`, `Homepage URL` and `Application Description` you want. Click the `Register application` button.

2. Copy `Client ID` and `Client Secret` from the *GitHub Settings* [https://github.com/settings/applications].

Now that you have the client id and secret, you can proceed with the creation of a Github Identity Provider in Keycloak. As follows:

1. Select the `Github` identity provider from the drop-down box on the top right corner of the identity providers table in Keycloak's Admin Console. You should be presented with a specific page to configure the selected provided.
2. Copy the client id and secret to their corresponding fields in the Keycloak Admin Console. Click `Save`.

Once you create the identity provider in Keycloak, you must update your GitHub application with the redirect url that was generated to your identity provider.

1. Open the GitHub Settings and select your application. In `Authorization callback URL` insert the redirect uri created by Keycloak. The redirect uri usually have the following format: `http://{host}:{port}/auth/realms/{realm}/broker/{provider_alias}`.

**Note**

You can always get the redirect url for a specific identity provider from the table presented when you click on the 'Identity Provider' tab in *Realm > Settings*.

That is it! This pretty much what you need to do in order to setup this identity provider.

The table below lists some additional configuration options you may use when configuring this provider.

**Table 10.5. Configuration Options**

| Configuration | Description |
|---|---|
| `Default Scopes` | 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://developer.github.com/v3/oauth/#scopes*. By default, Keycloak uses the following scopes: `user:email` |

## 10.3.5. LinkedIn

To enable login with LinkedIn you first have to create an application in *LinkedIn Developer Network* [https://www.linkedin.com/secure/developer]. Then you need to copy the client id and secret into the Keycloak Admin Console.

Let's see first how to create an application with LinkedIn.

1. Log in to *LinkedIn Developer Network* [https://www.linkedin.com/secure/developer]. Click the `Add New Application` link. Use any value for `Application Name`, `Website URL`, `Description`, `Developer Contact Email` and `Phone` you want. Select `r_basicprofile` and `r_emailaddress` in the `Default Scope` section. Click the `Add Application` button.
2. Copy `Consumer Key / API Key` and `Consumer Secret / Secret Key` from the shown page.

Now that you have the client id and secret, you can proceed with the creation of a LinkedIn Identity Provider in Keycloak. As follows:

1. Select the `LinkedIn` identity provider from the drop-down box on the top right corner of the identity providers table in Keycloak's Admin Console. You should be presented with a specific page to configure the selected provided.
2. Copy the client id and secret to their corresponding fields in the Keycloak Admin Console. Click `Save`.

Once you create the identity provider in Keycloak, you must update your LinkedIn application with the redirect url that was generated to your identity provider.

1. Open the LinkedIn Developer Network and select your application. In `OAuth 2.0 Redirect URLs` insert the redirect uri created by Keycloak. The redirect uri usually have the following format: `http://{host}:{port}/auth/realms/{realm}/broker/{provider_alias}/endpoint`.

**Note**

You can always get the redirect url for a specific identity provider from the table presented when you click on the 'Identity Provider' tab in *Realm > Settings*.

That is it! This pretty much what you need to do in order to setup this identity provider.

The table below lists some additional configuration options you may use when configuring this provider.

### Table 10.6. Configuration Options

| Configuration | Description |
|---|---|
| `Default Scopes` | 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 application configuration in *LinkedIn Developer Network* [https://www.linkedin.com/secure/developer]. |

| Configuration | Description |
| --- | --- |
| | By default, Keycloak uses the following scopes: `r_basicprofile r_emailaddress` |

## 10.3.6. Microsoft

To enable login with Microsoft account you first have to register an OAuth application on *Microsoft account Developer Center* [https://account.live.com/developers/applications/index]. Then you need to copy the client id and secret into the Keycloak Admin Console.

Let's see first how to create an application with Microsoft.

1. Go to *create new application on Microsoft account Developer Center* [https://account.live.com/developers/applications/create] url and login here. Use any value for `Application Name`, `Application Logo` and `URLs` you want. In `API Settings` set `Target Domain` to the domain where your Keycloak instance runs.
2. Copy `Client Id` and `Client Secret` from `App Settings` page.

Now that you have the client id and secret you can proceed with the creation of a Microsoft Identity Provider in Keycloak. As follows:

1. Select the `Microsoft` identity provider from the drop-down box on the top right corner of the identity providers table in Keycloak's Admin Console. You should be presented with a specific page to configure the selected provided.
2. Copy the client id and client secret to their corresponding fields in the Keycloak Admin Console. Click `Save`.

Once you create the identity provider in Keycloak, you must update your Microsoft application with the redirect url that was generated to your identity provider.

1. Open the Microsoft account Developer Center and select `API Settings` of your application. In `Redirect URLs` insert the redirect uri created by Keycloak. The redirect uri usually have the following format: `http://{host}:{port}/auth/realms/{realm}/broker/microsoft/endpoint`.

### Note

You can always get the redirect url for a specific identity provider from the table presented when you click on the 'Identity Provider' tab in *Realm > Settings*.

That is it! This pretty much what you need to do in order to setup this identity provider.

The table below lists some additional configuration options you may use when configuring this provider.

### Table 10.7. Configuration Options

| Configuration | Description |
| --- | --- |
| `Default Scopes` | 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://msdn.microsoft.com/en-us/library/hh243646.aspx*. By default, Keycloak uses the following scopes: `wl.basic,wl.emails` |

## 10.3.7. StackOverflow

To enable login with StackOverflow you first have to register an OAuth application on *StackApps* [https://stackapps.com/]. Then you need to copy the client id, secret and key into the Keycloak Admin Console.

Let's see first how to create an application with StackOverflow.

1. Go to *registering your application on Stack Apps* [http://stackapps.com/apps/oauth/register] url and login here. Use any value for `Application Name`, `Application Website` and `Description` you want. Set `OAuth Domain` to the domain where your Keycloak instance runs. Click the `Register Your Application` button.
2. Copy `Client Id`, `Client Secret` and `Key` from the shown page.

Now that you have the client id, secret and key, you can proceed with the creation of a StackOverflow Identity Provider in Keycloak. As follows:

1. Select the `StackOverflow` identity provider from the drop-down box on the top right corner of the identity providers table in Keycloak's Admin Console. You should be presented with a specific page to configure the selected provided.
2. Copy the client id, client secret and key to their corresponding fields in the Keycloak Admin Console. Click `Save`.

That is it! This pretty much what you need to do in order to setup this identity provider.

The table below lists some additional configuration options you may use when configuring this provider.

### Table 10.8. Configuration Options

| Configuration | Description |
| --- | --- |
| `Default Scopes` | Allows you to manually specify the scopes that users must authorize when authenticating with this provider. For |

| Configuration | Description |
| --- | --- |
| | a complete list of scopes, please take a look at application configuration in *StackExchange API Authentication* [https://api.stackexchange.com/docs/authentication#scope] documentation. Keycloak uses the empty scope by default. |

## 10.4. SAML v2.0 Identity Providers

Keycloak can broker identity providers based on the SAML v2.0 protocol.

In order to configure a SAML identity provider, follow these steps:

1. Select the `SAML v2.0` identity provider from the drop-down box on the top right corner of the identity providers table in Keycloak's Admin Console. You should be presented with a specific page to configure the selected provider.

When configuring a SAML identity provider you are presented with different configuration options in order to properly communicate and integrate with the external identity provider. In this case, you must keep in mind that Keycloak will act as an Service Provider that issues authentication requests(AuthnRequest) to the external identity provider.

### Table 10.9. Configuration Options

| Configuration | Description |
| --- | --- |
| `Import IdP SAML Metadata` | When creating a new identity provider, you may just upload the SAML Metadata for the brokered IdP (IDPSSODescriptor). In this case, Keycloak will read all the necessary configuration from the metadata and automatically configure the identity provider for you. |
| `Single Sign-On Service Url` | Allows you to specify the URL that will be used to send SAML authentication requests. |
| `Single Logout Service Url` | Allows you to specify the URL that will be used to send SAML logout requests. |
| `Backchannel Logout` | If set to true, logout to the external IDP will be done in a background HTTP request. If set to false, then the browser will be redirected to the external IDP to perform the logout. |
| `NameID Policy Format` | Allows you to specify a NameID Policy that will be sent in the SAML authentication request. |

| Configuration | Description |
| --- | --- |
| `Validating X509 Certificate` | Allows you to specify the certificate in PEM format that will be used to validate signatures for all messages received from the brokered identity provider. |
| `Want AuthnRequests Signed` | Allows you to specify whether the brokered identity provider is expecting signed SAML authentication requests or not. |
| `Force Authentication` | Allows you to tell the brokered identity provider that user must be authenticated even if he was previously authenticated (re-authentication) in the same session. |
| `Validate Signature` | Enable or disable signature validation of any message returned by the brokered identity provider. |
| `HTTP-POST Binding Response` | Allows you to specify if responses from the brokered identity providers are returned using the HTTP-POST or HTTP-Redirect protocol bindings. If enabled, only responses using HTTP-POST binding are accepted. |
| `HTTP-POST Binding for AuthnReques` | Allows you to specify whether SAML authentication requests must be sent using the HTTP-POST or HTTP-Redirect protocol bindings. If enabled, it will send requests using HTTP-POST binding. |

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.

Once you create a SAML provider, there is an `EXPORT` button that appears when viewing that provider. Clicking this button will export a SAML entity descriptor which you can use to

### 10.4.1. SP Descriptor

The SAML SP Descriptor XML file for the broker is available publically by going to this URL

```
http[s]://{host:port}/auth/realms/{realm-name}/broker/{broker-alias}/endpoint/descriptor
```

This URL is useful if you need to import this information into an IDP that needs or is more user friendly to load from a remote URL.

## 10.5. OpenID Connect v1.0 Identity Providers

Keycloak can broker identity providers based on the OpenID Connect v1.0 protocol.

In order to configure a OIDC identity provider, follow these steps:

1. Select the `OpenID Connect v1.0` identity provider from the drop-down box on the top right corner of the identity providers table in Keycloak's Admin Console. You should be presented with a specific page to configure the selected provider.

When configuring an OIDC identity provider you are presented with different configuration options in order to send authentication requests to the external identity provider. In this case, the brokered identity provider must support the Authorization Code Flow (as defined by the specification) in order to authenticate the user and authorize access for the scopes specified in the configuration.

### Table 10.10. Configuration Options

| Configuration | Description |
|---|---|
| `Authorization Url` | The authorization url. |
| `Token Url` | The token url. |
| `Logout Url` | The IDP logout url. |
| `Backchannel Logout` | If set to true, logout to the external IDP will be done in a background HTTP request. If set to false, then the browser will be redirected to the external IDP to perform the logout. |
| `User Info Url` | The user info url. This is usually an url from where Keycloak will obtain user information in order to create a local account. |
| `Client ID` | The client id is usually generated when configuring an application or a project on the brokered identity provider. |
| `Client Secret` | The client secret is usually generated when configuring an application or a project on the brokered identity provider. |
| `Issuer` | Allows you to specify the expected value for the "issuer" claim when validating the ID token. |
| `Default Scopes` | Allows you to specify additional scopes when asking for user authentication and consent. By default, scope `openid` is always appended to the list of the scopes. |
| `Prompt` | Allows you to specify how the brokered identity provider must prompt user for |

| Configuration | Description |
| --- | --- |
| | authentication. You must check which values are supported by the brokered identity provider before choosing a value. |

You can also import all this configuration data by providing a URL or file that points to OpenID Provider Metadata (see OIDC Discovery specification)

## 10.6. Retrieving Tokens from Identity Providers

Keycloak allows you to store tokens and responses from identity providers during the authentication process. For that, you can use the `Store Token` configuration option, as mentioned before.

It also allows you to retrieve these tokens and responses once the user is authenticated in order to use their information or use them to invoke external resources protected by these tokens. The latter case is usually related with social providers, where you usually need to use their tokens to invoke methods on their 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}
```

In this case, given that you are accessing an protected service in Keycloak, you need to send the access token issued by Keycloak during the user authentication.

By default, the Keycloak access token issued for the application can't be automatically used for retrieve thirdparty token. A user will have to have the `broker.read-token` role. The client will also have to have that role in its scope. In the broker configuration page you can automatically assign this role to newly imported users by turning on the `Stored Tokens Readable` switch.

**Note**

If your application is not at the same origin as the authentication server, make sure you have properly configured CORS.

## 10.7. Automatically Select and Identity Provider

Each Identity provider has option `Authenticate By Default`, which allows that Identity provider is automatically selected during authentication. User won't even see Keycloak login page, but is automatically redirected to the identity provider.

Applications can also automatically select an identity provider in order to authenticate an user. Selection per application is preferred over `Authenticate By Default` option if you need more control on when exactly is Identity provider automatically selected.

Keycloak supports a specific HTTP query parameter that you can use as a hint to tell the server which identity provider should be used to authenticate the user.

For that, you can append the `kc_idp_hint` as a query parameter to your application url, as follows:

```
GET /myapplication.com?kc_idp_hint=facebook HTTP/1.1
Host: localhost:8080
```

In this case, is expected that your realm has an identity provider with an alias `facebook`.

If you are using `keycloak.js` adapter, you can also achieve the same behavior:

```
var keycloak = new Keycloak('keycloak.json');

keycloak.createLoginUrl({
    idpHint: 'facebook'
});
```

## 10.8. Mapping/Importing SAML and OIDC Metadata

You can import SAML assertion data, OpenID Connect ID Token claims, and Keycloak access token claims into new users that are imported from a brokered IDP. After you configure a broker, you'll see a `Mappers` button appear. Click on that and you'll get to the list of mappers that are assigned to this broker. 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. You can assign a role mapping to a user if a claim or external role exists. There's a bunch of options here so just mouse over the tool tips to see what each mapper can do for you.

## 10.9. Mapping/Importing User profile data from Social Identity Provider

You can import user profile data provided by social identity providers like Google, GitHub, LinkedIn, Microsoft, Stackoverflow and Facebook into new Keycloak user created from given social accounts. After you configure a broker, you'll see a `Mappers` button appear. Click on that and you'll get to the list of mappers that are assigned to this broker. There is a `Create` button on this page. Clicking on this create button allows you to create a broker mapper. "Attribute Importer" mapper allows you to define path in JSON user profile data provided by the provider to get value from. You can use dot notation for nesting and square brackets to access fields in array by index.

For example 'contact.address[0].country'. Then you can define name of Keycloak's user profile attribute this value is stored into.

To investigate structure of user profile JSON data provided by social providers you can enable `DEBUG` level for logger `org.keycloak.social.user_profile_dump` and login using given provider. Then you can find user profile JSON structure in Keycloak log file.

## 10.10. User Session Data

After a user logs in from the external IDP, there's 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.

BROKER_PROVIDER_ID
: This is the IDP alias of the broker used to perform the login.

## 10.11. First Login Flow

When Keycloak successfully authenticates user through identity provider (step 8 in *Overview* chapter), there can be two situations:

1. There is already Keycloak user account linked with the authenticated identity provider account. In this case, Keycloak will just authenticate as the existing user and redirect back to application (step 9 in *Overview* chapter).
2. There is not yet existing Keycloak user account linked with the identity provider account. This situation is more tricky. Usually you just want to register new account into Keycloak database, but what if there is existing Keycloak account with same email like the identity provider account? Automatically link identity provider account with existing Keycloak account is not very good option as there are possible security flaws related to that...

Because we had various requirements what to do in second case, we changed the behaviour to be flexible and configurable through *Authentication Flows SPI*. In admin console in Identity provider settings, there is option `First Login Flow`, which allows you to choose, which workflow will be used after "first login" with this identity provider account. 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 etc.

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 either re-configure existing flow (For example disable some authenticators, mark some of them as `required`, configure some authenticators etc). Or you can even create new authentication flow and/or write your own Authenticator implementations and use it in your flow. See *Authentication Flows SPI* for more details on how to do it.

For `First Broker Login` case, it might be useful if your Authenticator is subclass of `org.keycloak.authentication.authenticators.broker.AbstractIdpAuthenticator` so

you have access to all details about authenticated Identity provider account. But it's not a requirement.

### 10.11.1. Default First Login Flow

Let's describe the default behaviour provided by `First Broker Login` flow. There are those authenticators:

Review Profile
: This authenticator might display the profile info page, where user can review his profile retrieved from identity provider. The authenticator is configurable. You can set `Update Profile On First Login` option. When `On`, users will be always presented with the profile page asking for additional information in order to federate 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 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 existing Keycloak account with same email or username like the account from identity provider. If it's not, then authenticator just creates new Keyclok account and link it with identity provider and 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 existing Keycloak account and user needs to link his identity provider account through Account management.

  This authenticator also has config option `Require Password Update After Registration` . When enabled, user is required to update password after account is created.

Confirm Link Existing Account
: User will see the info page, that there is existing Keycloak account with same email. He can either review his profile again and use different email or username (flow is restarted and goes back to `Review Profile` authenticator). Or he can confirm that he wants to link identity provider account with his 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 realm has SMTP setup configured. It will send mail to user, where he can confirm that he wants to link identity provider with his 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 non-available (SMTP not configured for realm). It will display login screen where user needs to authenticate with his

password to link his Keycloak account with Identity provider. User can also re-authenticate with some different identity provider, which is already linked to his keycloak account. You can also force users to use OTP, otherwise it's optional and used only if OTP is already set for user account.

## 10.12. Examples

Keycloak provides some useful examples about how to use it as an identity broker. Take a look at `{KEYCLOAK_HOME}/examples/broker` for more details.

Each example application has its own README file where you can find additional information about how to configure Keycloak and run it.

# Chapter 11. 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.

## 11.1. Theme types

A theme can support several types to customize different aspects of Keycloak. The types currently available are:

- Account - Account management
- Admin - Admin console
- Email - Emails
- Login - Login forms
- Welcome - Welcome pages

## 11.2. Configure theme

All theme types, except welcome, is configured through `Keycloak Admin Console`. To change the theme used for a realm open the `Keycloak Admin Console`, select your realm from the drop-down box in the top left corner. Under `Settings` click on `Theme`.

To set the theme for the `master` Keycloak admin console set the admin console theme for the `master` realm. To set the theme for per realm admin access control set the admin console theme for the corresponding realm.

To change the welcome theme you need to edit `standalone/configuration/keycloak-server.json` and add `welcomeTheme` to the theme element, for example:

```
"theme": {
    ...
    "welcomeTheme": "custom-theme"
}
```

## 11.3. Default themes

Keycloak comes bundled with default themes in the server's root `themes` directory. You should not edit the bundled themes directly. Instead create a new theme that extends a bundled theme.

## 11.4. Creating a theme

A theme consists of:

- *FreeMarker* [http://freemarker.org] templates
- Stylesheets
- Scripts
- Images
- Message bundles
- Theme properties

A theme can extend another theme. When extending a theme you can override individual files (templates, stylesheets, etc.). The recommended way to create a theme is to extend the base theme. The base theme provides templates and a default message bundle. If you decide to override templates bear in mind that you may need to update your templates when upgrading to a new release to include any changes made to the original template.

Before creating a theme it's a good idea to disable caching as this makes it possible to edit theme resources without restarting the server. To do this open `../standalone/configuration/keycloak-server.json` for `theme` set `staticMaxAge` to `-1` and `cacheTemplates` and `cacheThemes` to `false`. For example:

```
[
"theme": {
    "default": "keycloak",
    "staticMaxAge": -1,
    "cacheTemplates": false,
    "cacheThemes": false,
    "folder": {
      "dir": "${jboss.home.dir}/themes"
    }
},
```

Remember to re-enable caching in production as it will significantly impact performance.

To create a new theme create a directory for the theme in the server's root `themes`. The name of the directory should be the name of the theme. For example to create a theme called `example-theme` create the directory `themes/example-theme`. Inside the theme directory you then need to create a directory for each of the types your theme is going to provide. For example to add the login type to the `example-theme` theme create the directory `themes/example-theme/login`.

For each type create a file `theme.properties` which allows setting some configuration for the theme, for example what theme it overrides and if it should import any themes. For the above example we want to override the base theme and import common resources from the Keycloak theme. To do this create the file `themes/example-theme/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 `example-theme` and click `Save`. Then open the login page for the realm. You can do this either by login through your application or by opening `http://localhost:8080/realms/<realm name>/account`.

To see the effect of changing the parent theme, set `parent=keycloak` in `theme.properties` and refresh the login page. To follow the rest of the documentation set it back to `parent=base` before continuing.

### 11.4.1. Stylesheets

A theme can have one or more stylesheets, to add a stylesheet create a file inside `resources/css` (for example `resources/css/styles.css`) inside your theme folder. Then registering it in `theme.properties` by adding:

```
styles=css/styles.css
```

The `styles` property supports a space separated list so you can add as many as you want. For example:

```
styles=css/styles.css css/more-styles.css
```

For the example-theme above add `example-theme/login/resources/css/styles.css` with the following content:

```
[
#kc-form {
    background-color: #000;
    color: #fff;
    padding: 20px;
}
```

Then edit `example-theme/login/theme.properties` and add

```
styles=css/styles.css
```

. Refresh the login page to see your changes. It's not pretty, but you can see how easily you can modify the styles for your theme.

### 11.4.2. Scripts

A theme can have one or more scripts, to add a script create a file inside `resources/js` (for example `resources/js/script.js`) inside your theme folder. Then registering it in `theme.properties` by adding:

```
scripts=js/script.js
```

The `scripts` property supports a space separated list so you can add as many as you want. For example:

```
scripts=js/script.js js/more-script.js
```

### 11.4.3. Images

To make images available to the theme add them to `resources/img`. They can then be used through stylesheets. For example:

```
body {
    background-image: url('../img/image.jpg');
}
```

Or in templates, for example:

```
<img src="${url.resourcesPath}/img/image.jpg">
```

### 11.4.4. Messages

Text in the templates are loaded from message bundles. A theme that extends another theme will inherit all messages from the parents message bundle, but can override individual messages. For example to replace `Username` on the login form with `Your Username` create the file `messages/messages.properties` inside your theme folder and add the following content:

```
username=Your Username
```

For the admin console, there is a second resource bundle named `admin-messages.properties`. This resource bundle is converted to JSON and shipped to the console to be processed by angular-translate. It is found in the same directory as messages.properties and can be overridden in the same way as described above.

### 11.4.5. Modifying HTML

Keycloak uses *Freemarker Templates* [http://freemarker.org] in order to generate HTML. These templates are defined in `.ftl` files and can be overriden from the base theme. Check out the Freemarker website on how to form a template file. To override the login template for the `example-theme` copy `themes/base/login/login.ftl` to `themes/example-theme/login` and open it in an editor. After the first line (<#import ...>) add `<h1>HELLO WORLD!</h1>` then refresh the page.

## 11.5. Deploying themes

Themes can be deployed to Keycloak by copying the theme directory to `themes` or it can be deployed as a module. For a single server or during development just copying the theme is fine, but in a cluster or domain it's recommended to deploy as a module.

To deploy a theme as a module you need to create an jar (it's basically just a zip with jar extension) with the theme resources and a file `META/keycloak-themes.json` that describes the themes contained in the archive. For example `example-theme.jar` with the contents:

- META-INF/keycloak-themes.json
- theme/example-theme/login/theme.properties
- theme/example-theme/login/login.ftl
- theme/example-theme/login/resources/css/styles.css

The contents of META-INF/keycloak-themes.json in this case would be:

```
[
{
    "themes": [{
        "name" : "example-theme",
        "types": [ "login" ]
    }]
}
```

As you can see a single jar can contain multiple themes and each theme can support one or more types.

The deploy the jar as a module to Keycloak you can either manually create the module or use `jboss-cli`. It's simplest to use `jboss-cli` as it creates the required directories and module descriptor for you. To deploy the above jar `jboss-cli` run:

```
[
                KEYCLOAK_HOME/bin/jboss-cli.sh    --command="module    add    --
name=org.example.exampletheme --resources=example-theme.jar"
```

If you're on windows run

```
KEYCLOAK_HOME/bin/jboss-cli.bat
```

.

This command creates `modules/org/example/exampletheme/main` containing `example-theme.jar` and `module.xml`.

Once you've created the module you need to register it with Keycloak do this by editing `../standalone/configuration/keycloak-server.json` and adding the module to `theme/module/modules`. For example:

```
[
"theme": {
    ...
    "module": {
        "modules": [ "org.example.exampletheme" ]
    }
}
```

If a theme is deployed to `themes` and as a module the first is used.

## 11.6. SPIs

For full control of login forms and account management Keycloak provides a number of SPIs.

### 11.6.1. Account SPI

The Account SPI allows implementing the account management pages using whatever web framework or templating engine you want. To create an Account provider implement `org.keycloak.account.AccountProviderFactory` and `org.keycloak.account.AccountProvider`.

Once you have deployed your account provider to Keycloak you need to configure `keycloak-server.json` to specify which provider should be used:

```
"account": {
    "provider": "custom-provider"
}
```

### 11.6.2. Login SPI

The Login SPI allows implementing the login forms using whatever web framework or templating engine you want. To create a Login forms provider implement `org.keycloak.login.LoginFormsProviderFactory` and `org.keycloak.login.LoginFormsProvider` in `forms/login-api`.

Once you have deployed your account provider to Keycloak you need to configure `keycloak-server.json` to specify which provider should be used:

```
"login": {
    "provider": "custom-provider"
}
```

# Chapter 12. Recaptcha Support on Registration

To safeguard registration against bots, Keycloak has integration with Google Recaptcha. To enable this you need to first go to *Google Recaptcha* [https://developers.google.com/recaptcha/] 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, go to the Keycloak Admin Console. Go to Authentication->Flows page. Select the 'registration' flow. Set the 'Recaptcha' requirement to 'Required'. Click on the 'Configure' button and enter in the Recaptcha site key and secret.

Finally, you have to change Keycloak's default security headers. In the Admin Console, go to Settings->Security Defenses of your realm. Add a space and `https://www.google.com` to the values of both the `X-Frame-Options` and `Content-Security-Policy` headers. i.e.

```
frame-src 'self' https://www.google.com
```

That's it! You may want to edit register.ftl in your login theme to muck around with the placement and styling of the recaptcha button. Up to you.

# Chapter 13. Email

Keycloak sends emails to users to verify their email address. Emails are also used to allow users to safely restore their username and passwords.

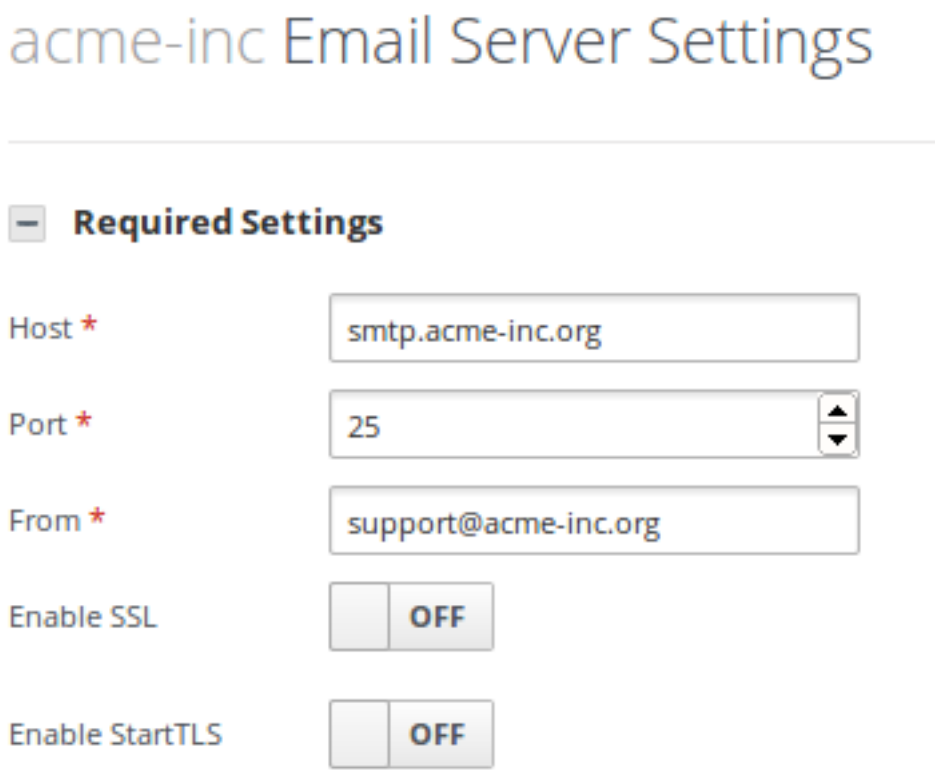
## 13.1. Email Server Config

To enable Keycloak to send emails you need to provide Keycloak with your SMTP server settings. If you don't have a SMTP server you can use one of many hosted solutions (such as Sendgrid or smtp2go).

To configure your SMTP server, open the `Keycloak Admin Console`, select your realm from the drop-down box in the top left corner. Then click on `Email` in the menu at the top.

You are required to fill in the `Host` and `Port` for your SMTP server (the default port for SMTP is 25). You also have to specify the sender email address (`From`). The other options are optional.

The screenshot below shows a simple example where the SMTP server doesn't use SSL or TLS and doesn't require authentication.




### 13.1.1. Enable SSL or TLS

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).

### 13.1.2. Authentication

If your SMTP server requires authentication click on `Enable Authentication` and insert the `Username` and `Password`.

# Chapter 14. Client Access Types

When you create a Client in admin console you may be wondering what the "Access Types" are.

confidential
: Confidential access type is for clients that need to perform a browser login and that you want to require a client secret when they turn an access code into an access token, (see *Access Token Request* [http://tools.ietf.org/html/rfc6749#section-4.1.3] in the OAuth 2.0 spec for more details). The advantages of this is that it is a little extra security. Since Keycloak requires you to register valid redirect-uris, I'm not exactly sure what this little extra security is though. :) The disadvantages of this access type is that confidential access type is pointless for pure Javascript clients as anybody could easily figure out your client's secret!

public
: Public access type is for clients that need to perform a browser login and that you feel that the added extra security of confidential access type is not needed. FYI, Pure javascript clients are by nature public.

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.

direct access only
: You would also see a "Direct Access Only" switch when creating the Client. This switch is for clients that only use the *Direct Access Grant* protocol to obtain access tokens.

# Chapter 15. Roles

In Keycloak, roles can be defined globally at the realm level, or individually per application. Each role has a name which must be unique at the level it is defined in, i.e. you can have only one "admin" role at the realm level. You may have that a role named "admin" within an Application too, but "admin" must be unique for that application.

The description of a role is displayed in the OAuth Grant page when Keycloak is processing a browser OAuth Grant request. Look for more features being added here in the future like internationalization and other fine grain options.

## 15.1. Composite Roles

Any realm or application 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. I guess another term for it could be Role Group. When a composite role is mapped to the user, the user gains the permission of that role, plus any other role the composite is associated with. This association is dynamic. So, if you add or remove an associated role from the composite, then all users that are mapped to the composite role will automatically have those permissions added or removed. Composites can also be used to define Client scopes.

Composite roles can be associated with any type of role Realm or Application. In the admin console simple flip the composite switch in the Role detail, and you will get a screen that will allow you to associate roles with the composite.

# Chapter 16. Groups

Groups in Keycloak allow you to manage a common set of attributes and role mappings for a large set of users. Users can be members of zero or more groups. Users inherit the attributes and role mappings assign to each group. As an admin this makes it easy for you to manage permissions for a user in one place.

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 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.

## 16.1. 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 are used to assign permissions and access control.

Keycloak Roles have the concept of a Composite Role. A role can be associated with one or more additional roles. This is called a Composite Role. If a user has a role mapping to the Composite Role, they inherit all the roles associated with the composite. So what's the difference from a Keycloak Group and a Composite Role? Logically they could be used for the same exact thing. The difference is conceptual. Composite roles should be used to compose the permission model of your set of services and applications. So, roles become a set of permissions. Groups on the other hand, would be a set of users that have a set of permissions. Use Groups to manage users, composite roles to manage applications and services.

# Chapter 17. Direct Access Grants

Keycloak allows you to make direct REST invocations to obtain an access token. (See *Resource Owner Password Credentials Grant* [http://tools.ietf.org/html/rfc6749#section-4.3] from OAuth 2.0 spec). To use it you must also have registered a valid Client to use as the "client_id" for this grant request.

**Warning**

It is highly recommended that you do not use Direct Access Grants to write your own login pages for your application. You will lose a lot of features that Keycloak has if you do this. Specifically all the account management, remember me, lost password, account reset features of Keycloak. Instead, if you want to tailor the look and feel of Keycloak login pages, you should create your own *theme*. There are also security implications to using Direct Access Grants compared to the redirect based flows as you are exposing plain text passwords to applications directly.

It is even highly recommended that you use the browser to log in for native mobile applications! Android and iPhone applications allow you to redirect to and from the browser. You can use this to redirect the user from your native mobile app to the web browser to perform login, then the browser will redirect back to your native application.

The REST URL to invoke on is `/{keycloak-root}/realms/{realm-name}/protocol/openid-connect/token`. Invoking on this URL is a POST request and requires you to post the username and credentials of the user you want an access token for. You must also pass along the "client_id" of the client you are creating an access token for. This "client_id" is the Client Id specified in admin console (not it's id from DB!). Depending on whether your client is *"public" or "confidential"*, you may also have to pass along it's client secret as well. We support pluggable client authentication, so alternatively you can use other form of client credentials like signed JWT assertion. See *Client Authentication* section for more details. Finally you need to pass "grant_type" parameter with value "password" .

For public client's, the POST invocation requires form parameters that contain the username, credentials, and client_id of your application. For example:

```
    POST /auth/realms/demo/protocol/openid-connect/token
    Content-Type: application/x-www-form-urlencoded

                            username=bburke&password=geheim&client_id=customer-
portal&grant_type=password
```

The response would be this *standard JSON document* [http://tools.ietf.org/html/rfc6749#section-4.3.3] 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":3600,
    "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
    "id_token":"tGzv3JOkF0XG5Qx2TlKWIA",
    "session_state":"234234-234234-234234"
}
```

For confidential client's, you must create a Basic Auth `Authorization` header that contains the client_id and client secret. And pass in the form parameters for username and for each user credential. For example:

```
    POST /auth/realms/demo/protocol/openid-connect/token
    Authorization: Basic atasdf023l2312023
    Content-Type: application/x-www-form-urlencoded

    username=bburke&password=geheim&grant_type=password
```

As mentioned above, we support also other means of authenticating clients. In adition to default client_id and client secret, we also have signed JWT assertion by default. There is possibility to use any other form of client authentication implemented by you. See *Client Authentication* section for more details.

Here's a Java example using Apache HTTP Client and some Keycloak utility classes.:

```
HttpClient client = new HttpClientBuilder()
                .disableTrustManager().build();


try {
   HttpPost post = new HttpPost(
           KeycloakUriBuilder.fromUri("http://localhost:8080/auth")
```

```
          .path(ServiceUrlConstants.TOKEN_PATH).build("demo"));
   List <NameValuePair> formparams = new ArrayList <NameValuePair>();
          formparams.add(new    BasicNameValuePair(OAuth2Constants.GRANT_TYPE,
 "password"));
   formparams.add(new BasicNameValuePair("username", "bburke"));
   formparams.add(new BasicNameValuePair("password", "password"));

   if (isPublic()) { // if client is public access type
             formparams.add(new  BasicNameValuePair(OAuth2Constants.CLIENT_ID,
 "customer-portal"));
   } else {
         String authorization = BasicAuthHelper.createHeader("customer-portal",
 "secret-secret-secret");
        post.setHeader("Authorization", authorization);
   }
   UrlEncodedFormEntity form = new UrlEncodedFormEntity(formparams, "UTF-8");
   post.setEntity(form);

   HttpResponse response = client.execute(post);
   int status = response.getStatusLine().getStatusCode();
   HttpEntity entity = response.getEntity();
   if (status != 200) {
      throw new IOException("Bad status: " + status);
   }
   if (entity == null) {
      throw new IOException("No Entity");
   }
   InputStream is = entity.getContent();
   try {
          AccessTokenResponse  tokenResponse  =  JsonSerialization.readValue(is,
 AccessTokenResponse.class);
   } finally {
      try {
          is.close();
      } catch (IOException ignored) { }
      }
} finally {
   client.getConnectionManager().shutdown();
}
```

Once you have the access token string, you can use it in REST HTTP bearer token authorized requests, i.e

```
GET /my/rest/api
```

```
Authorization: Bearer 2YotnFZFEjr1zCsicMWpAA
```

To logout you must use the refresh token contained in the AccessTokenResponse object.

```
List<NameValuePair> formparams = new ArrayList<NameValuePair>();
if (isPublic()) { // if client is public access type
    formparams.add(new BasicNameValuePair(OAuth2Constants.CLIENT_ID, "customer-
portal"));
} else {
       String  authorization  =  BasicAuthHelper.createHeader("customer-portal",
 "secret-secret-secret");
    post.setHeader("Authorization", authorization);
}
formparams.add(new           BasicNameValuePair(OAuth2Constants.REFRESH_TOKEN,
 tokenResponse.getRefreshToken()));
HttpResponse response = null;
URI logoutUri = KeycloakUriBuilder.fromUri(getBaseUrl(request) + "/auth")
                    .path(ServiceUrlConstants.TOKEN_SERVICE_LOGOUT_PATH)
                    .build("demo");
HttpPost post = new HttpPost(logoutUri);
UrlEncodedFormEntity form = new UrlEncodedFormEntity(formparams, "UTF-8");
post.setEntity(form);
response = client.execute(post);
int status = response.getStatusLine().getStatusCode();
HttpEntity entity = response.getEntity();
if (status != 204) {
   error(status, entity);
}
if (entity == null) {
   return;
}
InputStream is = entity.getContent();
if (is != null) is.close();
```

# Chapter 18. Service Accounts

Keycloak allows you to obtain an access token dedicated to some Client Application (not to any user). See *Client Credentials Grant* [http://tools.ietf.org/html/rfc6749#section-4.4] from OAuth 2.0 spec.

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. Don't forget that you need those roles to be available in Scopes of this client as well (unless you have `Full Scope Allowed` on). As in normal login, roles from access token are intersection of scopes and the service account roles.

The REST URL to invoke on is `/{keycloak-root}/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 client with signed JWT assertion or any other custom mechanism for client authentication. See *Client Authentication* section for more details. You also need to use parameter `grant_type=client_credentials` as per OAuth2 specification.

For example the POST invocation to retrieve service account can look like this:

```
POST /auth/realms/demo/protocol/openid-connect/token
Authorization: Basic cHJvZHVjdC1zYS1jbGllbnQ6cGFzc3dvcmQ=
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials
```

The response would be this *standard JSON document* [http://tools.ietf.org/html/rfc6749#section-4.4.3] 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":"tGzv3JOkF0XG5Qx2TlKWIA",
    "refresh_expires_in":600,
    "id_token":"tGzv3JOkF0XG5Qx2TlKWIA",
    "not-before-policy":0,
    "session_state":"234234-234234-234234"
}
```

The retrieved access token can be refreshed or logged out by out-of-bound request.

See the example application `service-account` from the main Keycloak `demo` example.

# Chapter 19. CORS

CORS stands for Cross-Origin Resource Sharing. If executing browser Javascript tries to make an AJAX HTTP request to a server's whose domain is different than the one the Javascript code came from, then the request uses the *CORS protocol* [http://www.w3.org/TR/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 and other Javascript-based attacks. Keycloak has support for validated CORS requests.

Keycloak's CORS support is configured per client. You specify the allowed origins in the client's configuration page in the admin console. You can add as many you want. The value must be what the browser would send as a value in the `Origin` header. For example `http://example.com` is what you must specify to allow CORS requests from `example.com`. When an access token is created for the client, these allowed origins are embedded within the token. On authenticated CORS requests, your application's Keycloak adapter will handle the CORS protocol and validate the `Origin` header against the allowed origins embedded in the token. If there is no match, then the request is denied.

To enable CORS processing in your application's server, you must set the `enable-cors` setting to `true` in your *adapter's configuration file*. When this setting is enabled, the Keycloak adapter will handle all CORS preflight requests. It will validate authenticated requests (protected resource requests), but will let unauthenticated requests (unprotected resource requests) pass through.

## 19.1. Handling CORS Yourself

This section is for Java developers securing servlet-based applications using our servlet adapter.

If you don't like our CORS support you can handle it yourself in a filter or something. One problem you will encounter is that our adapter will may trigger for any CORS preflight OPTIONS requests to blindly secured URLs. This will result in 302 redirection or 401 responses for the preflight OPTIONS request. To workaround this problem, you must modify your web.xml security constraints to let OPTIONS requests through

```
<security-constraint>
     <web-resource-collection>
         <web-resource-name>wholesale</web-resource-name>
         <url-pattern>/*</url-pattern>
         <http-method>GET</http-method>
         <http-method>POST</http-method>
         <http-method>PUT</http-method>
         <http-method>DELETE</http-method>
     </web-resource-collection>
...
</security-constraint>
```

The above security constraint will secure all URLs, but only on GET, POST, PUT, and DELETE calls. OPTIONS requests will be let through.

# Chapter 20. Cookie settings, Session Timeouts, and Token Lifespans

Keycloak has a bunch of fine-grain settings to manage browser cookies, user login sessions, and token lifespans. Sessions can be viewed and managed within the admin console for all users, and individually in the user's account management pages. This chapter goes over configuration options for cookies, sessions, and tokens.

## 20.1. Remember Me

If you go to the admin console page of Settings->General, you should see a `Remember Me` on/off switch. Your realm sets a SSO cookie so that you only have to enter in your login credentials once. This `Remember Me` admin config option, when turned on, will show a "Remember Me" checkbox on the user's login page. If the user clicks this, the realm's SSO cookie will be persistent. This means that if the user closes their browser they will still be logged in the next time they start up their browser.

## 20.2. Session Timeouts

If you go to the Sesions and Tokens->Timeout Settings page of the Keycloak adminstration console there is a bunch of fine tuning you can do as far as login session timeouts go.

The `SSO Session Idle Timeout` is the idle time of a user session. If there is no activity in the user's session for this amount of time, the user session will be destroyed, and the user will become logged out. The idle time is refreshed with every action against the keycloak server for that session, i.e.: a user login, SSO, a refresh token grant, etc.

The `SSO Session Max Lifespan` setting is the maximum time a user session is allowed to be alive. This max lifespan countdown starts from when the user first logs in and is never refreshed. This works great with `Remember Me` in that it allow you to force a relogin after a set timeframe.

## 20.3. Token Timeouts

The `Access Token Lifespan` is how long an access token is valid for. An access token contains everything an application needs to authorize a client. It contains roles allowed as well as other user information. When an access token expires, your application will attempt to refresh it using a refresh token that it obtained in the initial login. The value of this configuration option should be however long you feel comfortable with the application not knowing if the user's permissions have changed. This value is usually in minutes.

The `Access Token Lifespan For Implicit Flow` is how long an access token is valid for when using OpenID Connect implicit flow. With implicit flow, there is no refresh

token available. That's why the lifespan is usually bigger than default Access Token Lifespan mentioned above. See *OpenID Connect specification* [http://openid.net/specs/openid-connect-core-1_0.html#ImplicitFlowAuth] for details about implicit flow and *Javascript Adapter* for some additional details on how to use it in Keycloak.

The `Client login timeout` is how long an access code is valid for. An access code is obtained on the 1st leg of the OAuth 2.0 redirection protocol. This should be a short time limit. Usually seconds.

The `Login user action lifespan` is how long a user is allowed to attempt a login. When a user tries to login, they may have to change their password, set up TOTP, or perform some other action before they are redirected back to your application as an authentnicated user. This value is relatively short and is usually measured in minutes.

## 20.4. Offline Access

The Offline access is the feature described in *OpenID Connect specification* [http://openid.net/specs/openid-connect-core-1_0.html#OfflineAccess] . The idea is that during login, your client application will request Offline token instead of classic Refresh token. Then the application can save this offline token in the database and can use it anytime later even if user is logged out. This is useful for example if your application needs to do some "offline" actions on behalf of user even if user is not online. For example periodic backup of some data every night etc.

Your application is responsible for persist the offline token in some storage (usually database) and then use it to manually retrieve new access token from Keycloak server.

The difference between classic Refresh token and Offline token is, that offline token will never expire and is not subject of `SSO Session Idle timeout` . The offline token is valid even after user logout or server restart. However you need to use offline token for refresh at least once per each 30 days ( The value can be changed in admin console. It is `Offline Session Idle timeout` ). Also if you enable option `Revoke refresh tokens` , then each offline token can be used just once. So after refresh, you always need to store new offline token from refresh response into your DB instead of the previous one.

User can revoke the offline tokens in Account management UI. The admin user can revoke offline tokens for individual users in admin console (The `Consent` tab of particular user) and he can see all the offline tokens of all users for particular client application in the settings of the client. Revoking of all offline tokens for particular client is possible by set `notBefore` policy for the client.

For requesting the offline token, user needs to be in realm role `offline_access` and client needs to have scope for this role. If client has `Full scope allowed`, the scope is granted by default. Also users are automatically members of the role as it's the default role.

The client can request offline token by adding parameter `scope=offline_access` when sending authorization request to Keycloak. The adapter automatically adds this parameter when you use it to access secured URL of your application (ie. http://localhost:8080/customer-portal/secured?scope=offline_access ). The *Direct Access Grant* or *Service account* flows also support offline

tokens if you include `scope=offline_access` in the body of the authentication request. For more details, see the `offline-access-app` example from Keycloak demo.

# Chapter 21. Admin REST API

The Keycloak Admin Console is implemented entirely with a fully functional REST admin API. You can invoke this REST API from your Java applications by obtaining an access token. You must have the appropriate permissions set up as described in *Chapter 6, Master Admin Access Control* and *Chapter 7, Per Realm Admin Access Control*

The documentation for this REST API is auto-generated and is contained in the distribution of keycloak under the docs/rest-api/overview-index.html directory, or directly from the docs page at the keycloak website.

There are a number of examples that come with the keycloak distribution that show you how to invoke on this REST API. `examples/preconfigured-demo/admin-access-app` shows you how to access this api from java. `examples/cors/angular-product-app` shows you how to invoke on it from Javascript. Finally there is example in `example/admin-client`, which contains example for Admin client, that can be used to invoke REST endpoints easily as Java methods.

# Chapter 22. Events

Keycloak provides an Events SPI that makes it possible to register listeners for user related events, for example user logins. There are two interfaces that can be implemented, the first is a pure listener, the second is a events store which listens for events, but is also required to store events. An events store provides a way for the admin and account management consoles to view events.

## 22.1. 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 has changed
- Remove TOTP - TOTP has been removed from an account
- Send Verify Email - A 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.

## 22.2. Event Listener

Keycloak comes with an Email Event Listener and a JBoss Logging Event Listener. The Email Event Listener sends an email to the users account when an event occurs. The JBoss Logging Event Listener writes to a log file when an events occurs.

The Email Event Listener only supports the following events at the moment:

- Login Error
- Update Password
- Update TOTP
- Remove TOTP

You can exclude one or more events by editing `standalone/configuration/keycloak-server.json` and adding for example:

```
"eventsListener": {
    "email": {
        "exclude-events": [ "UPDATE_TOTP", "REMOVE_TOTP" ]
    }
}
```

## 22.3. Event Store

Event Store listen for events and is expected to persist the events to make it possible to query for them later. This is used by the admin console and account management to view events. Keycloak includes providers to persist events to JPA and Mongo.

You can specify events to include or exclude by editing `standalone/configuration/keycloak-server.json`, and adding for example:

```
"eventsStore": {
    "jpa": {
        "exclude-events": [ "LOGIN", "REFRESH_TOKEN", "CODE_TO_TOKEN" ]
    }
}
```

## 22.4. Configure Events Settings for Realm

To enable persisting of events for a realm you first need to make sure you have a event store provider registered for Keycloak. By default the JPA event store provider is registered. Once you've done that open the admin console, select the realm you're configuring, select `Events`. Then click on `Config`. You can enable storing events for your realm by toggling `Save Events` to ON. You can also set an expiration on events. This will periodically delete events from the database that are older than the specified time.

To configure listeners for a realm on the same page as above add one or more event listeners to the `Listeners` select box. This will allow you to enable any registered event listeners with the realm.

# Chapter 23. User Federation SPI and LDAP/AD Integration

Keycloak can federate external user databases. Out of the box we have support for LDAP and Active Directory. Before you dive into this, you should understand how Keycloak does federation.

Keycloak performs federation a bit differently than other products/projects. The vision of Keycloak is that it is an out of the box solution that should provide a core set of feature irregardless of the backend user storage you want to use. Because of this requirement/vision, Keycloak has a set data model that all of its services use. Most of the time when you want to federate an external user store, much of the metadata that would be needed to provide this complete feature set does not exist in that external store. For example your LDAP server may only provide password validation, but not support TOTP or user role mappings. The Keycloak User Federation SPI was written to support these completely variable configurations.

The way user federation works is that Keycloak will import your federated users on demand to its local storage. How much metadata that is imported depends on the underlying federation plugin and how that plugin is configured. Some federation plugins may only import the username into Keycloak storage, others might import everything from name, address, and phone number, to user role mappings. Some plugins might want to import credentials directly into Keycloak storage and let Keycloak handle credential validation. Others might want to handle credential validation themselves. The goal of the Federation SPI is to support all of these scenarios.

## 23.1. LDAP and Active Directory Plugin

Keycloak comes with a built-in LDAP/AD plugin. By default, it is set up only to import username, email, first and last name, but you are free to configure *mappers* and add more attributes or delete default ones. It supports password validation via LDAP/AD protocols and different user metadata synchronization modes. To configure a federated LDAP store go to the admin console. Click on the `Users` menu option to get you to the user management page. Then click on the `Federation` submenu 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.

### 23.1.1. Edit Mode

Edit mode defines various synchronization options with your LDAP store depending on what privileges you have.

READONLY
: Username, email, first and 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 and last name, 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 and 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.

## 23.1.2. Other config options

Display Name
: Name used when this provider is referenced in the admin console

Priority
: The priority of this provider when looking up users or for adding registrations.

Sync Registrations
: If a new user is added through a registration page or admin console, should the user be eligible to be synchronized to this provider.

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 use tooltips in admin console to see some more details about them.

## 23.1.3. Connect to LDAP over SSL

When you configure secured connection URL to LDAP (for example `ldaps://myhost.com:636` ) the Keycloak will use SSL for the communication with LDAP server. The important thing is to properly configure truststore on the Keycloak server side, because SSL won't work if Keycloak can't trust the SSL connection with LDAP (Keycloak acts as the `client` here, when LDAP acts as server).

The global truststore for the Keycloak can be configured with Truststore SPI in the `keycloak-server.json` file and it's described in the details *here*. If you don't configure 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 finally the cacerts file from JDK if even the system property is not set).

There is configuration property `Use Truststore SPI` in the LDAP federation provider configuration, where you can choose whether Truststore SPI is used. By default, the value is `ldaps only`, which is fine for most of deployments, because attempt to use Truststore SPI is done just if connection to LDAP starts with `ldaps` .

## 23.2. Sync of LDAP users to Keycloak

LDAP Federation Provider will automatically take care of synchronization (import) of needed LDAP users into Keycloak database. For example once you first authenticate LDAP user `john` from Keycloak UI, LDAP Federation provider will first import this LDAP user into Keycloak database and then authenticate against LDAP password.

Federation Provider imports just requested users by default, so if you click to `View all users` in Keycloak admin console, you will see just those LDAP users, which were already authenticated/ requested by Keycloak.

If you want to sync all LDAP users into Keycloak database, you may configure and enable Sync, which is in admin console on same page like the configuration of Federation provider itself. There are 2 types of sync:

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 Doe`).

Changed users sync
: This will check LDAP and it will sync into Keycloak just those users, which were created or updated in LDAP from the time of last sync.

In usual cases you may want to trigger full sync at the beginning, so you will import all LDAP users to Keycloak just once. Then you may setup periodic sync of changed users, so Keycloak will periodically ask LDAP server for newly created or updated users and backport them to Keycloak DB. Also you may want to trigger full sync again after some longer time or setup periodic full sync as well.

In admin console, you can trigger sync directly or you can enable periodic changed or full sync.

## 23.3. LDAP/Federation mappers

LDAP mappers are `listeners`, which are triggered by LDAP Federation provider at various points and provide another extension point to LDAP integration. They are triggered during import LDAP user into Keycloak, registration Keycloak user back to LDAP or when querying LDAP user from Keycloak. When you create LDAP Federation provider, Keycloak will automatically provide set of builtin `mappers` for this provider. You are free to change this set and create new mapper or update/delete existing ones.

By default, we have those implementation of LDAP federation mapper:

User Attribute Mapper
: This allows to specify which LDAP attribute is mapped to which attribute of Keycloak User. So for example you can configure that LDAP attribute `mail` is supposed to be mapped to

the UserModel attribute `email` in Keycloak database. For this mapper implementation, there is always one-to-one mapping (one LDAP attribute mapped to one Keycloak UserModel attribute)

FullName Mapper
: This allows to specify that fullname of user, which is saved in some LDAP attribute (usualy `cn` ) will be mapped to `firstName` and `lastname` attributes of UserModel. Having `cn` to contain full name of user is common case for some LDAP deployments.

Role Mapper
: This allows to configure role mappings from LDAP into Keycloak role mappings. One Role mapper can be used to map LDAP roles (usually groups from particular branch of LDAP tree) into roles corresponding to either realm roles or client roles of specified client. It's not a problem to configure more Role mappers for 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 specified Keycloak role to each Keycloak user linked with LDAP.

Group Mapper
: This allows to configure group mappings from LDAP into Keycloak group mappings. Group mapper can be used to map LDAP groups from particular branch of LDAP tree into groups in Keycloak. And it will also propagate user-group mappings from LDAP into user-group mappings in Keycloak.

  You can choose to preserve group inheritance from LDAP as well, but this may fail as Keycloak inheritance is more restrictive than LDAP (For example in Keycloak each group can have just one parent and there is no recursion allowed. In LDAP the recursion is possible and every group can be member of more other groups too).

  As of now, the mapper doesn't provide mapping of LDAP roles-groups to Keycloak roles-groups (For example when LDAP group `cn=role1,ou=roles,dc=example,dc=com` is member of LDAP group `cn=group1,ou=groups,dc=example,dc=com` , we don't support the mapping of Keycloak role `role1` imported from LDAP to corresponding Keycloak group `group1` imported from LDAP).

MSAD User Account Mapper
: Mapper specific to Microsoft Active Directory (MSAD). It's able to tightly integrate the MSAD user account state into Keycloak account state (account enabled, password is expired etc). It's using `userAccountControl` and `pwdLastSet` LDAP attributes for that (both are specific to MSAD and are not LDAP standard). For example if pwdLastSet is 0, the Keycloak user is required to update password (there will be UPDATE_PASSWORD required action added to him in Keycloak). Or if userAccountControl is 514 (disabled account) the Keycloak user is disabled as well etc.

For writable LDAP, the mapping is bi-directional and the state from Keycloak is propagated to LDAP (For example enable user in Keycloak admin console will update the value of userAccountControl in MSAD and effectively enable him in MSAD as well).

For writable LDAPs, mapper also provides mapping of error codes during MSAD user authentication to the appropriate action in Keycloak. For example if MSAD user authentication fails due to the fact, that MSAD password is expired, the mapper will allow user to authenticate into Keycloak, but it will add UPDATE_PASSWORD required action to the user, so user must update his password.

By default, there is set of User Attribute mappers to map basic UserModel attributes username, first name, lastname and email to corresponding LDAP attributes. You are free to extend this and provide more attribute mappings (For example to street, postalCode etc), delete firstName/ lastname mapper and put fullName mapper instead, add role mappers etc. Admin console provides tooltips, which should help on how to configure corresponding mappers.

We have an example, which is showing LDAP integration and set of base mappers and sample mappers (mappers for street and postalCode) . It's in `examples/ldap` in the Keycloak example distribution or demo distribution download. You can also check the example sources directly *here* [https://github.com/keycloak/keycloak/blob/master/examples/ldap] .

## 23.4. Writing your own User Federation Provider

The keycloak examples directory contains an example of a simple User Federation Provider backed by a simple properties file. See `examples/providers/federation-provider`. Most of how to create a federation provider is explained directly within the example code, but some information is here too.

Writing a User Federation Provider starts by implementing the `UserFederationProvider` and `UserFederationProviderFactory` interfaces. Please see the Javadoc and example for complete details on how to do this. Some important methods of note: getUserByUsername() and getUserByEmail() require that you query your federated storage and if the user exists create and import the user into Keycloak storage. How much metadata you import is fully up to you. This import is done by invoking methods on the object returned `KeycloakSession.userStorage()` to add and import user information. The proxy() method will be called whenever Keycloak has found an imported UserModel. This allows the federation provider to proxy the UserModel which is useful if you want to support external storage updates on demand.

After your code is written you must package up all your classes within a JAR file. This jar file must contain a file called `org.keycloak.models.UserFederationProviderFactory` within the `META-INF/services` directory of the JAR. This file is a list of fully qualified classnames of all implementations of `UserFederationProviderFactory`. For more details on writing provider implementations and how to deploy to Keycloak refer to the *providers* section.

# Chapter 24. Kerberos brokering

Keycloak supports login with Kerberos ticket through SPNEGO. 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 sends HTML login screen together with status 401 and HTTP header `WWW-Authenticate: Negotiate`
5. In case that 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 login screen.
6. Keycloak validates token from browser and authenticate 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

## 24.1. 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 "kadmin" session. If you are on same machine where is MIT Kerberos, you can simply use command:

```
sudo kadmin.local
```

  Then add HTTP principal and export his key to 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
```

  Keytab file `/tmp/http.keytab` will need to be accessible on the host where keycloak server will be running.

## 24.2. Setup and configuration of Keycloak server

- Install kerberos client. This is again platform dependent. If you are on Fedora, Ubuntu or RHEL, you can install package `freeipa-client`, which contains Kerberos client and bunch of other stuff.
- Configure 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 `domain_realm` section like this:

```
[domain_realm]
  .mydomain.org = MYDOMAIN.ORG
  mydomain.org = MYDOMAIN.ORG
```

- Export keytab file with 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 MIT Kerberos example above, we already exported keytab to `/tmp/http.keytab` . If your KDC and Keycloak are running on same host, you have file already available.

- Finally run Keycloak server and configure SPNEGO/Kerberos authentication in Keycloak admin console. Keycloak supports Kerberos authentication through *Federation provider SPI* . We have 2 federation providers with Kerberos authentication support:

  Kerberos
  : This provider is useful if you want to authenticate with Kerberos `NOT` backed by LDAP server. In this case, users are usually created to Keycloak database after first successful SPNEGO/Kerberos login and they may need to update profile after first login, as Kerberos protocol itself doesn't provision any data like first name, last name or email.

    You can also choose if users can authenticate with classic username/password. In this case, if user doesn't have SPNEGO ticket available, Keycloak will display login screen and user can fill his Kerberos username and password on login screen. Username/password works also for non-web flows like *Direct Access grants*.

  LDAP
  : This provider is useful if you want to authenticate with Kerberos backed by LDAP server. In this case, data about users are provisioned from LDAP server after successful Kerberos authentication.

## 24.3. 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 for example *this* [http://www.microhowto.info/howto/configure_firefox_to_authenticate_using_spnego_and_kerberos.html] for more info about Firefox. URI `.mydomain.org` must be allowed in `network.negotiate-auth.trusted-uris` config option.

In 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.

## 24.4. Example setups

For easier testing with Kerberos, we provided some example setups to test.

### 24.4.1. Keycloak and FreeIPA docker image

Once you install *docker* [https://www.docker.com/], you can run docker image with *FreeIPA* [http://www.freeipa.org/] server installed. FreeIPA provides integrated security solution with MIT Kerberos and 389 LDAP server among other things . The image provides also Keycloak server configured with LDAP Federation provider and enabled SPNEGO/Kerberos authentication against the FreeIPA server. See details *here* [https://github.com/mposolda/keycloak-freeipa-docker/blob/master/README.md] .

### 24.4.2. ApacheDS testing Kerberos server

For quick testing and unit tests, we use very simple *ApacheDS* [http://directory.apache.org/apacheds/] Kerberos server. You need to build Keycloak from sources and then run Kerberos server with maven-exec-plugin from our testsuite. See details *here* [https://github.com/keycloak/keycloak/blob/master/misc/Testsuite.md#kerberos-server] .

## 24.5. Credential delegation

One scenario supported by Kerberos 5 is credential delegation. In this case when user receives forwardable TGT and authenticates to the web server, then web server might be able to reuse the ticket and forward it to another service secured by Kerberos (for example LDAP server or IMAP server).

The scenario is supported by Keycloak, but there is tricky thing that SPNEGO authentication is done by Keycloak server but GSS credential will need to be used by your application. So you need to enable built-in `gss delegation credential` protocol mapper in admin console for your application. This will cause that Keycloak will deserialize GSS credential and transmit it to the application in access token. Application will need to deserialize it and use it for further GSS calls against other services. We have an example, which is showing it in details. It's in `examples/kerberos` in the Keycloak example distribution or demo distribution download. You can also check the example sources directly *here* [https://github.com/keycloak/keycloak/blob/master/examples/kerberos] .

Once you deserialize the credential from the access token to the GSSCredential object, then GSSContext will need to be created with this credential passed to the method `GSSManager.createContext` for example like this:

```
GSSContext context = gssManager.createContext(serviceName, krb5Oid,
    deserializedGssCredFromKeycloakAccessToken, GSSContext.DEFAULT_LIFETIME);
```

Note that you also need to configure `forwardable` kerberos tickets in `krb5.conf` file and add support for delegated credentials to your browser. For details, see the kerberos example from Keycloak examples set as mentioned above.

**Warning**

Credential delegation has some security implications. So enable the protocol claim and support in browser just if you really need it. It's highly recommended to use it together with HTTPS. See for example *this article* [http://www.microhowto.info/howto/configure_firefox_to_authenticate_using_spnego_and_kerberos.html#idp27072] for details.

## 24.6. Troubleshooting

If you have issues, we recommend to enable more logging by:

- Enable `Debug` flag in admin console for Kerberos or LDAP federation providers
- Enable TRACE logging for category `org.keycloak` in logging section of `$WILDFLY_HOME/standalone/configuration/standalone.xml` to receive more info `$WILDFLY_HOME/standalone/log/server.log`
- Add system properties `-Dsun.security.krb5.debug=true` and `-Dsun.security.spnego.debug=true`

# Chapter 25. Export and Import

## 25.1. Startup export/import

Export/import is useful especially if you want to migrate your whole Keycloak database from one environment to another or migrate to different database (For example from MySQL to Oracle). You can trigger export/import at startup of Keycloak server and it's configurable with System properties right now. The fact it's done at server startup means that no-one can access Keycloak UI or REST endpoints and edit Keycloak database on the fly when export or import is in progress. Otherwise it could lead to inconsistent results.

You can export/import your database either to:

- Directory on local filesystem
- Single JSON file on your filesystem

When importing using the "dir" 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 import to Directory, you can specify also the number of users to be stored in each JSON file. So if you have very large amount of users in your database, you likely don't want to import them into single file as the file might be very big. Processing of each file is done in separate transaction as exporting/importing all users at once could also lead to memory issues.

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
: can be 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
: can be used to specify for Directory providers to specify where to import users. Possible values are:

  - DIFFERENT_FILES - Users will be exported into more different files according to 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 realm (So file like "foo-realm.json" with both realm data and users)
  - SAME_FILE - All users will be exported to same file but different than realm (So file like "foo-realm.json" with realm data and "foo-users.json" with users)

-Dkeycloak.migration.usersPerFile
: can be used to specify number of users per file (and also per DB transaction). It's 5000 by default. It's used only if usersExportStrategy is DIFFERENT_FILES

-Dkeycloak.migration.strategy
: is used during import. It can be used to specify how to proceed if realm with same name already exists in the database where you are going to import data. Possible values are:

  - IGNORE_EXISTING - Ignore importing if realm of this name already exists
  - OVERWRITE_EXISTING - Remove existing realm and import it again with new data from JSON file. If you want to fully migrate one environment to another and ensure that the new environment will contain same data like 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

## 25.2. Admin console export/import

Import of most resources can be performed from the admin console. Exporting resources will be supported in future versions.

The files created during a "startup" export can 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.

**Warning**

The admin console import allows you to "overwrite" resources if you choose. Use this feature with caution, especially on a production system.

# Chapter 26. Server Cache

By default, Keycloak caches realm metadata and users. There are two separate caches, one for realm metadata (realm, application, client, roles, etc...) and one for users. These caches greatly improves the performance of the server.

## 26.1. Disabling Caches

The realm and user caches can be cleared through the management console. To disable the realm or user cache, you must edit the `keycloak-server.json` file in your distribution. Here's what the config looks like initially.

```
"userCache": {
    "infinispan" : {
        "enabled": true
    }
},

"realmCache": {
    "infinispan" : {
        "enabled": true
    }
},
```

You must then change it to:

```
"userCache": {
    "infinispan" : {
        "enabled": false
    }
},

"realmCache": {
    "infinispan" : {
        "enabled": false
    }
},
```

## 26.2. Clear Caches

To clear the realm or user cache, go to the Keycloak admin console Realm Settings->Cache Config page. Disable the cache you want. This will cause the cache to be cleared.

# Chapter 27. SAML SSO

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. We do not yet support logout via redirects. All logouts happen via a background POST binding request to the application that will be logged out. We do not support SAML 1.1 either. If you want support for either of those, please log a JIRA request and we'll schedule it.

When you create an application in the admin console, you can choose which protocol the application will log in with. In the application create screen, choose `saml` from the protocol list. After that there are a bunch of configuration options. Here is a description of each item:

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.

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.

Multi-valued Roles
: If this switch is off, any user role mappings will have a corresponding attribute created for it. If this switch is turn on, only one role attribute will be created, but it will have multiple values within in.

Sign Documents
: When turned on, Keycloak will sign the document using the realm's private key.

Sign Assertions
: With the `Sign Documents` switch signs the whole document. With this setting you just assign the assertions of the document.

Signature Algorithm
: Choose between a variety of algorithms for signing SAML documents.

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 keys set up in the `Application Keys` submenu item.

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 by a 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.

  username
  : Format: `urn:oasis:names:tc:SAML:1.1:nameid-format:unspecified`

    Source: `UserModel.userName` property

  email
  : Format: `urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress`

    Source: `UserModel.email` property

  transient
  : Format: `urn:oasis:names:tc:SAML:2.0:nameid-format:transient`

    Source:`G-$randomUuid`, I.E. `G-5ef5b38f-864f-41ad-82a0-04ade9139500`

  persistent
  : Format: `urn:oasis:names:tc:SAML:2.0:nameid-format:persistent`

    The persistent identifier will be evaluated in the following order. If one step does not yield a value, processing will continue on to the next until a value is found.

    1) saml.persistent.name.id.for.$clientId `UserModel` attribute
       identifier unique to this client/user pair.

    2) saml.persistent.name.id.for.* `UserModel` attribute
       user identifier for all clients, unless otherwise overridden by a $clientId attribute.

    3) G-$randomUuid
       I.E.`G-5ef5b38f-864f-41ad-82a0-04ade9139500`. If neither a $clientId or wildcard user attribute is found, a persistent identifier will be generated for the given client.

Note that once this identifier has been generated, a user attribute with the key *saml.persistent.name.id.for.$clientId* will be persisted and used on all subsequent requests against the given client.

Master SAML Processing URL
: This URL will be used for all SAML requests and responsed 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 valided 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.

For login to work, Keycloak needs to be able to resolve the URL for the Assertion Consumer Service of the SP. If you are relying on the SP to provide this URL in the login request, then you must register valid redirect uri patterns so that this URL can be validated. You can set the Master SAML Processing URL as well, or alternatively, you can specify the Assertion Consumer Service URL per binding.

For logout to work, you must specify a Master SAML Processing URL, or the Loging Service URL for the binding you want Keycloak to use.

One thing to note is that roles are not treated as a hierarchy. So, any role mappings will just be added to the role attributes in the SAML document using their basic name. So, if you have multiple application roles you might have name collisions. You can use the Scope Mapping menu item to control which role mappings are set in the response.

## 27.1. SAML Entity Descriptor

If you go into the admin console in the application list menu page you will see an `Import` button. If you click on that you can import SAML Service Provider definitions using the *Entity Descriptor* [http://docs.oasis-open.org/security/saml/v2.0/saml-metadata-2.0-os.pdf] format described in SAML 2.0. You should review all the information there to make sure everything is set up correctly.

Each realm has a URL where you can view the XML entity descriptor for the IDP. `root/auth/realms/{realm}/protocol/saml/descriptor`

## 27.2. IDP Initiated Login

IDP Initiated Login is a feature that where you can set up a URL on the Keycloak server that will log you into a specific application/client. To set this up go to the client page in the admin console of the client you want to set this up for. 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}`

If your client requires a special relay state, you can also configure this in the admin console. Alternatively, you can specify the relay state in a `RelayState` query parameter, i.e. : `root/auth/realms/{realm}/protocol/saml/clients/{url-name}?RelayState=thestate`

# Chapter 28. Security Vulnerabilities

This chapter discusses possible security vulnerabilities Keycloak could have, how Keycloak mitigates those vulnerabilities, and what steps you need to do to configure Keycloak to mitigate some 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* [http://tools.ietf.org/html/rfc6819] document put out by the IETF. Many of those vulnerabilities are discussed here.

## 28.1. SSL/HTTPS Requirement

If you do not use SSL/HTTPS for all communication between the Keycloak 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.

## 28.2. 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 (e.g., via HTTP redirects or HTML forms). Any site that uses cookie based authentication is vulnerable for these types of attacks. These attacks are mitigated by matching a state cookie against a posted form or query parameter.

OAuth 2.0 login specification requires that a state cookie be used and matched against a transmitted state parameter. Keycloak fully implements this part of the specification so all logins are protected.

The Keycloak adminstration console is a pure Javascript/HTML5 application that makes REST calls to the backend Keycloak admin 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 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.

## 28.3. 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 an "Authorize" 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 *X-FRAME_OPTIONS* [http://tools.ietf.org/html/rfc7034] and *Content-Security-Policy* [http://www.w3.org/TR/CSP/]. You should take a look at both of these headers. In the admin console you can specify the values these headers will have. By default, Keycloak only sets up a same-origin policy for iframes.

## 28.4. Compromised Access Codes

It would be very hard for an attacker to compromise Keycloak access codes. Keycloak generates a cryptographically strong random value for its access codes so it would be very hard to guess an access token. An access code can only be turned into an access token once so it can't be replayed. In the admin console you can specify how long an access token is valid for. This value should be really short. Like a seconds. Just long enough for the client to make the request to turn the code into an token.

## 28.5. Compromised access and refresh tokens

There's a few things you can do to mitigate access tokens and refresh tokens from being stolen. Most importantly is to enforce SSL/HTTPS communication between Keycloak and its clients and applications. Short lifespans (minutes) for access tokens allows Keycloak to check the validity of a refresh token. Making sure refresh tokens always stay private to the client and are never transmitted ever is very important as well.

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. You can also disable specific applications, clients, and users if you feel that any one of those entities is completely compromised.

## 28.6. 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.

## 28.7. 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. The downside of this is that this makes Keycloak vulnerable to denial of service attacks. Eventually we will expand this functionality to take client IP address into account when deciding whether to block a user.

Another thing you can do to prevent password guessing is to point a tool like *Fail2Ban* [http://fail2ban.org] to the Keycloak server's log file. Keycloak logs every login failure and client IP address that had the failure.

In the admin console, per realm, you can set up a password policy to enforce that users pick hard to guess passwords. A password has to match all policies. The password policies that can be configured are hash iterations, length, digits, lowercase, uppercase, special characters, not username, regex patterns, password history and force expired password update. Force expired password update policy forces or requires password updates after specified span of time. Password history policy restricts a user from resetting his password to N old expired passwords. Multiple regex patterns can be specified. If there's more than one regex added, password has to match all fully. Increasing number of Hash Iterations (n) does not worsen anything (and certainly not the cipher) and it greatly increases the resistance to dictionary attacks. However the drawback to increasing n is that it has some cost (CPU usage, energy, delay) for the legitimate parties. Increasing n also slightly increases the odds that a random password gives the same result as the right password due to hash collisions, and is thus a false but accepted password; however that remains very unlikely, in the order of n*[1/(2^256)] for practical values of n, and can be entirely ignored in practice. Keycloak also uses PBKDF2 internally to cryptographically derive passwords to refine and improve the ratio of cost between attacker and legitimate parties. Good practice is to pay attention to the time complexity of hash_password and hash; then increase n as much as tolerable in the situation(s) at hand and and revise parameters such as n every few years to account for time complexity trade off.

Finally, the best way to mitigate against brute force attacks is to require user to set up a one-time-password (OTP).

## 28.8. Password database compromised

Keycloak does not store passwords in raw text. It stores a hash of them. Because of performance reasons, Keycloak only hashes passwords once. While a human could probably never crack a hashed password, it is very possible that a computer could. The security community suggests around 20,000 (yes thousand!) hashing iterations to be done to each password. This number grows every year due to increasing computing power (It was 1000 12 years ago). The problem with this is that password hashing is a huge performance hit as each login would require the entered

password to be hashed that many times and compared to the stored hash. So, its up to the admin to configure the password hash iterations. This can be done in the admin console password policy configuration. Again, the default value is 1 as we thought it might be more important for Keycloak to scale out of the box. There's a lot of other measures admins can do to protect their password databases.

## 28.9. SQL Injection attacks

At this point in time, there is no knowledge of any SQL injection vulnerabilities in Keycloak

## 28.10. Limiting Scope

Using the `Scope` menu in the admin console for clients, you can control exactly which role mappings will be included within the token sent back to the client application. This allows you to limit the scope of permissions given to the client which is great if the client isn't very trusted and is known to not being very careful with its tokens.

# Chapter 29. Clustering

To improve availability and scalability Keycloak can be deployed in a cluster.

It's fairly straightforward to configure a Keycloak cluster, the steps required are:

- Configure a shared database
- Configure Infinispan
- Enable realm and user cache invalidation
- Enable distributed user sessions
- Start in HA mode

## 29.1. Configure a shared database

Keycloak doesn't replicate realms and users, but instead relies on all nodes using the same database. This can be a relational database or Mongo. To make sure your database doesn't become a single point of failure you may also want to deploy your database to a cluster.

### 29.1.1. DB lock

Note that Keycloak supports concurrent startup by more cluster nodes at the same. This is ensured by DB lock, which prevents that some startup actions (migrating database from previous version, importing realms at startup, initial bootstrap of admin user) are always executed just by one cluster node at a time and other cluster nodes need to wait until the current node finishes startup actions and release the DB lock.

By default, the maximum timeout for lock is 900 seconds, so in case that second node is not able to acquire the lock within 900 seconds, it fails to start. The lock checking is done every 2 seconds by default. Typically you won't need to increase/decrease the default value, but just in case it's possible to configure it in `standalone/configuration/keycloak-server.json`:

```
"dblock": {
    "jpa": {
        "lockWaitTimeout": 900,
        "lockRecheckTime": 2
    }
}
```

or similarly if you're using Mongo (just by replace `jpa` with `mongo`)

## 29.2. Configure Infinispan

Keycloak uses *Infinispan* [http://www.infinispan.org/] caches to share information between nodes.

For realm and users Keycloak uses a invalidation cache. An invalidation cache doesn't share any data, but simply removes stale data from remote caches and makes sure all nodes re-load data from the database when it is changed. This reduces network traffic, as well as preventing sensitive data (such as realm keys and password hashes) from being sent between the nodes.

User sessions and login failures supports either distributed caches or fully replicated caches. We recommend using a distributed cache. A distributed cache splits user sessions into segments where each node holds one or more segment. It is possible to replicate each segment to multiple nodes, but this is not strictly necessary since the failure of a node will only result in users having to log in again. If you need to prevent node failures from requiring users to log in again, set the `owners` attribute to 2 or more for the `sessions` cache of `infinispan/Keycloak` container as described below.

The infinispan container is set by default in `standalone/configuration/keycloak-server.json`:

```
"connectionsInfinispan": {
    "default" : {
        "cacheContainer" : "java:jboss/infinispan/Keycloak"
    }
}
```

As you can see in this file, the realmCache, userCache and userSession providers are configured to use infinispan by default, which applies for both cluster and non-cluster environment.

For non-cluster configuration (server executed with `standalone.xml` ) is the infinispan container `infinispan/Keycloak` just uses local infinispan caches for realms, users and userSessions.

For cluster configuration, you can edit the configuration of `infinispan/Keycloak` container in `standalone/configuration/standalone-ha.xml` (or `standalone-keycloak-ha.xml` if you are using overlay or demo distribution) .

## 29.3. Start in HA mode

To start the server in HA mode, start it with:

```
# bin/standalone --server-config=standalone-ha.xml
```

or if you are using overlay or demo distribution with:

```
# bin/standalone --server-config=standalone-keycloak-ha.xml
```

Alternatively you can copy `standalone/config/standalone-ha.xml` to `standalone/config/standalone.xml` to make it the default server config.

## 29.4. Enabling cluster security

By default there's nothing to prevent unauthorized nodes from joining the cluster and sending potentially malicious messages to the cluster. However, as there's no sensitive data sent there's not much that can be achieved. For realms and users all that can be done is to send invalidation messages to make nodes load data from the database more frequently. For user sessions it would be possible to modify existing user sessions, but creating new sessions would have no affect as they would not be linked to any access tokens. There's not too much that can be achieved by modifying user sessions. For example it would be possible to prevent sessions from expiring, by changing the creation time. However, it would for example have no effect adding additional permissions to the sessions as these are rechecked against the user and application when the token is created or refreshed.

In either case your cluster nodes should be in a private network, with a firewall protecting them from outside attacks. Ideally isolated from workstations and laptops. You can also enable encryption of cluster messages, this could for example be useful if you can't isolate cluster nodes from workstations and laptops on your private network. However, encryption will obviously come at a cost of reduced performance.

To enable encryption of cluster messages you first have to create a shared keystore (change the key and store passwords!):

```
# keytool -genseckey -alias keycloak -keypass <PASSWORD> -storepass <PASSWORD> \
 -keyalg Blowfish -keysize 56 -keystore defaultStore.keystore -storetype JCEKS
```

Copy this keystore to all nodes (for example to standalone/configuration). Then configure JGroups to encrypt all messages by adding the `ENCRYPT` protocol to the JGroups sub-system (this should be added after the `pbcast.GMS` protocol):

```
<subsystem xmlns="urn:jboss:domain:jgroups:2.0" default-stack="udp">
    <stack name="udp">
        ...
        <protocol type="pbcast.GMS"/>
```

```
        <protocol type="ENCRYPT">
            <property name="key_store_name">
                ${jboss.server.config.dir}/defaultStore.keystore
            </property>
            <property name="key_password">PASSWORD</property>
            <property name="store_password">PASSWORD</property>
            <property name="alias">keycloak</property>
        </protocol>
        ...
    </stack>
    <stack name="tcp">
        ...
        <protocol type="pbcast.GMS"/>
        <protocol type="ENCRYPT">
            <property name="key_store_name">
                ${jboss.server.config.dir}/defaultStore.keystore
            </property>
            <property name="key_password">PASSWORD</property>
            <property name="store_password">PASSWORD</property>
            <property name="alias">keycloak</property>
        </protocol>
        ...
    </stack>
    ...
</subsystem>
```

See the *JGroups manual* [http://www.jgroups.org/manual/index.html#ENCRYPT] for more details.

## 29.5. Troubleshooting

Note that when you run 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/keycloak|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, so consult EAP or JGroups documentation for more info.

# Chapter 30. Application Clustering

This chapter is focused on clustering support for your own AS7, EAP6 or Wildfly applications, which are secured by Keycloak. We support various deployment scenarios according if your application is:

- stateless or stateful
- distributable (replicated http session) or non-distributable and just relying on sticky sessions provided by loadbalancer
- deployed on same or different cluster hosts where keycloak servers are deployed

The situation is a bit tricky as application communicates with Keycloak directly within user's browser (for example redirecting to login screen), but there is also backend (out-of-bound) communication between keycloak and application, which is hidden from end-user and his browser and hence can't rely on sticky sessions.

**Note**

To enable distributable (replicated) HTTP Sessions in your application, you may need to do some additional steps. Usually you need to put <distributable /> tag into `WEB-INF/web.xml` file of your application and possibly do some additional steps to configure underlying cluster cache (In case of Wildfly, the implementation of cluster cache is based on Infinispan). These steps are server specific, so consult documentation of your application server for more details.

## 30.1. Stateless token store

By default, the servlet web application secured by Keycloak uses HTTP session to store information about authenticated user account. This means that this info could be replicated across cluster and your application will safely survive failover of some cluster node.

However if you don't need or don't want to use HTTP Session, you may alternatively save all info about authenticated account into cookie. This is useful especially if your application is:

- stateless application without need of HTTP Session, but with requirement to be safe to failover of some cluster node
- stateful application, but you don't want sensitive token data to be saved in HTTP session
- stateless application relying on loadbalancer, which is not aware of sticky sessions (in this case cookie is your only way)

To configure this, you can add this line to configuration of your adapter in `WEB-INF/keycloak.json` of your application:

```
"token-store": "cookie"
```

Default value of `token-store` is `session`, hence saving data in HTTP session.

One limitation of cookie store is, that whole info about account is passed in cookie KEYCLOAK_ADAPTER_STATE in each HTTP request. Hence it's not the best for network performance. Another small limitation is limited support for Single-Sign out. It works without issues if you init servlet logout (HttpServletRequest.logout) from this application itself as the adapter will delete the KEYCLOAK_ADAPTER_STATE cookie. But back-channel logout initialized from different application can't be propagated by Keycloak to this application with cookie store. Hence it's recommended to use very short value of access token timeout (1 minute for example).

## 30.2. Relative URI optimization

In many deployment scenarios will be Keycloak and secured applications deployed on same cluster hosts. For this case Keycloak already provides option to use relative URI as value of option *auth-server-url* in `WEB-INF/keycloak.json` . In this case, the URI of Keycloak server is resolved from the URI of current request.

For example if your loadbalancer is on *https://loadbalancer.com/myapp* and auth-server-url is */auth*, then relative URI of Keycloak is resolved to be *https://loadbalancer.com/auth* .

For cluster setup, it may be even better to use option *auth-server-url-for-backend-request* . This allows to configure that backend requests between Keycloak and your application will be sent directly to same cluster host without additional round-trip through loadbalancer. So for this, it's good to configure values in `WEB-INF/keycloak.json` like this:

```
"auth-server-url": "/auth",
"auth-server-url-for-backend-requests": "http://${jboss.host.name}:8080/auth"
```

This would mean that browser requests (like redirecting to Keycloak login screen) will be still resolved relatively to current request URI like *https://loadbalancer.com/myapp*, but backend (out-of-bound) requests between keycloak and your app are sent always to same cluster host with application .

Note that additionally to network optimization, you may not need "https" in this case as application and keycloak are communicating directly within same cluster host.

## 30.3. Admin URL configuration

Admin URL for particular application can be configured in Keycloak admin console. It's used by Keycloak server to send backend requests to application for various tasks, like logout users or push revocation policies.

For example logout of user from Keycloak works like this:

1. User sends logout request from one of applications where he is logged.
2. Then application will send logout request to Keycloak
3. Keycloak server logout user in itself, and then it re-sends logout request by backend channel to all applications where user is logged. Keycloak is using admin URL for this. So logout is propagated to all apps.

You may again use relative values for admin URL, but in cluster it may not be the best similarly like in *previous section* .

Some examples of possible values of admin URL are:

http://${jboss.host.name}:8080/myapp
: This is best choice if "myapp" is deployed on same cluster hosts like Keycloak and is distributable. In this case Keycloak server sends logout request to itself, hence no communication with loadbalancer or other cluster nodes and no additional network traffic.

  Note that since the application is distributable, the backend request sent by Keycloak could be served on any application cluster node as invalidation of HTTP Session on *node1* will propagate the invalidation to other cluster nodes due to replicated HTTP sessions.

http://${application.session.host}:8080/myapp
: Keycloak will track hosts where is particular HTTP Session served and it will send session invalidation message to proper cluster node.

  For example application is deployed on *http://node1:8080/myapp* and *http://node2:8080/myapp* . Now HTTP Session *session1* is sticky-session served on cluster node *node2* . When keycloak invalidates this session, it will send request directly to *http://node2:8080/myapp* .

  This is ideal configuration for distributable applications deployed on different host than keycloak or for non-distributable applications deployed either on same or different nodes than keycloak. Good thing is that it doesn't send requests through load-balancer and hence helps to reduce network traffic.

## 30.4. Registration of application nodes to Keycloak

Previous section describes how can Keycloak send logout request to proper application node. However in some cases admin may want to propagate admin tasks to all registered cluster nodes,

not just one of them. For example push new notBefore for realm or application, or logout all users from all applications on all cluster nodes.

In this case Keycloak should be aware of all application cluster nodes, so it could send event to all of them. To achieve this, we support auto-discovery mechanism:

1. Once new application node joins cluster, it sends registration request to Keycloak server
2. The request may be re-sent to Keycloak in configured periodic intervals
3. If Keycloak won't receive re-registration request within specified timeout (should be greater than period from point 2) then it automatically unregister particular node
4. Node is also unregistered in Keycloak when it sends 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, so here you need to rely on automatic unregistration from point 3 .

Sending startup registrations and periodic re-registration is disabled by default, as it's main usecase is just cluster deployment. In `WEB-INF/keycloak.json` of your application, you can specify:

```
"register-node-at-startup": true,
"register-node-period": 600,
```

which means that registration is sent at startup (accurately when 1st request is served by the application node) and then it's resent each 10 minutes.

In Keycloak admin console you can specify the maximum node re-registration timeout (makes sense to have it bigger than *register-node-period* from adapter configuration for particular application). Also you can manually add and remove cluster nodes in admin console, which is useful if you don't want to rely on adapter's automatic registration or if you want to remove stale application nodes, which weren't unregistered (for example due to forced shutdown).

## 30.5. Refresh token in each request

By default, application adapter tries to refresh access token when it's expired (period can be specified as *Access Token Lifespan*) . However if you don't want to rely on the fact, that Keycloak is able to successfully propagate admin events like logout to your application nodes, then you have possibility to configure adapter to refresh access token in each HTTP request.

In `WEB-INF/keycloak.json` you can configure:

```
"always-refresh-token": true
```

Note that this has big performance impact. It's useful just if performance is not priority, but security is critical and you can't rely on logout and push notBefore propagation from Keycloak to applications.

# Chapter 31. Keycloak Security Proxy

Keycloak has an HTTP(S) proxy that you can put in front of web applications and services where it is not possible to install the keycloak adapter. You can set up URL filters so that certain URLs are secured either by browser login and/or bearer token authentication. You can also define role constraints for URL patterns within your applications.

## 31.1. Proxy Install and Run

Download the keycloak proxy distribution from the Keycloak download pages and unzip it.

```
$ unzip keycloak-proxy-dist.zip
```

To run it you must have a proxy config file (which we'll discuss in a moment).

```
$ java -jar bin/launcher.jar [your-config.json]
```

If you do not specify a path to the proxy config file, the launcher will look in the current working directory for the file named `proxy.json`

## 31.2. Proxy Configuration

Here's an example configuration file.

```
{
    "target-url": "http://localhost:8082",
    "send-access-token": true,
    "bind-address": "localhost",
    "http-port": "8080",
    "https-port": "8443",
    "keystore": "classpath:ssl.jks",
    "keystore-password": "password",
    "key-password": "password",
    "applications": [
        {
            "base-path": "/customer-portal",
            "error-page": "/error.html",
```

```
            "adapter-config": {
                "realm": "demo",
                "resource": "customer-portal",
                "realm-public-key": "MIGfMA0GCSqGSIb",
                "auth-server-url": "http://localhost:8081/auth",
                "ssl-required" : "external",
                "principal-attribute": "name",
                "credentials": {
                    "secret": "password"
                }
            }
            ,
            "constraints": [
                {
                    "pattern": "/users/*",
                    "roles-allowed": [
                        "user"
                    ]
                },
                {
                    "pattern": "/admins/*",
                    "roles-allowed": [
                        "admin"
                    ]
                },
                {
                    "pattern": "/users/permit",
                    "permit": true
                },
                {
                    "pattern": "/users/deny",
                    "deny": true
                }
            ]
        }
    ]
}
```

### 31.2.1. Basic Config

The basic configuration options for the server are as follows:

target-url
: The URL this server is proxying *REQUIRED.*.

send-access-token
: Boolean flag. If true, this will send the access token via the KEYCLOAK_ACCESS_TOKEN header to the proxied server. *OPTIONAL.*. Default is false.

bind-address
: DNS name or IP address to bind the proxy server's sockets to. *OPTIONAL.*. The default value is *localhost*

http-port
: Port to listen for HTTP requests. If you do not specify this value, then the proxy will not listen for regular HTTP requests. *OPTIONAL.*.

https-port
: Port to listen for HTTPS requests. If you do not specify this value, then the proxy will not listen for HTTPS requests. *OPTIONAL.*.

keystore
: Path to a Java keystore file that contains private key and certificate for the server to be able to handle HTTPS requests. Can be a file path, or, if you prefix it with `classpath:` it will look for this file in the classpath. *OPTIONAL.*. If you have enabled HTTPS, but have not defined a keystore, the proxy will auto-generate a self-signed certificate and use that.

buffer-size
: HTTP server socket buffer size. Usually the default is good enough. *OPTIONAL.*.

buffers-per-region
: HTTP server socket buffers per region. Usually the default is good enough. *OPTIONAL.*.

io-threads
: Number of threads to handle IO. Usually default is good enough. *OPTIONAL.*. The default is the number of available processors * 2.

worker-threads
: Number of threads to handle requests. Usually the default is good enough. *OPTIONAL.*. The default is the number of available processors * 16.

## 31.2.2. Application Config

Next under the `applications` array attribute, you can define one or more applications per host you are proxying.

base-path
: The base context root for the application. Must start with '/' *REQUIRED.*.

error-page
: If the proxy has an error, it will display the target application's error page relative URL *OPTIONAL.*. This is a relative path to the base-path. In the example above it would be `/customer-portal/error.html`.

adapter-config
: *REQUIRED.*. Same configuration as any other keycloak adapter. See *Adapter Config*

#### 31.2.2.1. Constraint Config

Next under each application you can define one or more constraints in the `constraints` array attribute. A constraint defines a URL pattern relative to the base-path. You can deny, permit, or require authentication for a specific URL pattern. You can specify roles allowed for that path as well. More specific constraints will take precedence over more general ones.

pattern
: URL pattern to match relative to the base-path of the application. Must start with '/' *REQUIRED.*. You may only have one wildcard and it must come at the end of the pattern. Valid `/foo/bar/*` and `/foo/*.txt` Not valid: `/*/foo/*`.

roles-allowed
: Array of strings of roles allowed to access this url pattern. *OPTIONAL.*.

methods
: Array of strings of HTTP methods that will exclusively match this pattern and HTTP request. *OPTIONAL.*.

excluded-methods
: Array of strings of HTTP methods that will be ignored when match this pattern. *OPTIONAL.*.

deny
: Deny all access to this URL pattern. *OPTIONAL.*.

permit
: Permit all access without requiring authentication or a role mapping. *OPTIONAL.*.

permit-and-inject
: Permit all access, but inject the headers, if user is already authenticated.*OPTIONAL.*.

authenticate
: Require authentication for this pattern, but no role mapping. *OPTIONAL.*.

### 31.2.3. Header Names Config

Next under the list of applications you can override the defaults for the names of the header fields injected by the proxy (see Keycloak Identity Headers). This mapping is optional.

keycloak-subject
: e.g. MYAPP_USER_ID

keycloak-username
: e.g. MYAPP_USER_NAME

keycloak-email
: e.g. MYAPP_USER_EMAIL

keycloak-name
: e.g. MYAPP_USER_ID

keycloak-access-token
: e.g. MYAPP_ACCESS_TOKEN

## 31.3. Keycloak Identity Headers

When forwarding requests to the proxied server, Keycloak Proxy will set some additional headers with values from the OIDC identity token it received for authentication.

KEYCLOAK-SUBJECT
: User id. Corresponds to JWT `sub` and will be the user id Keycloak uses to store this user.

KEYCLOAK-USERNAME
: Username. Corresponds to JWT `preferred_username`

KEYCLOAK-EMAIL
: Email address of user if set.

KEYCLOAK-NAME
: Full name of user if set.

KEYCLOAK-ACCESS-TOKEN
: Send the access token in this header if the proxy was configured to send it. This token can be used to make bearer token requests.

Header field names can be configured using a map of `header-names` in configuration file:

```
{
    "header-names" {
        "keycloak-subject": "MY-SUBJECT"
    }
}
```

# Chapter 32. Custom User Attributes

If you have custom user data you want to store and manage in the admin console, registration page, and user account service, you can easily add support for it by extending and modifying various Keycloak *themes*.

## 32.1. In admin console

To be able to enter custom attributes in the admin console, take the following steps

1. Create a new theme within the `themes/mytheme/admin` directory in your distribution. Where `mytheme` is whatever you want to name your theme.

2. Create a `theme.properties` file in this directory that extends the main admin console theme.

```
parent=keycloak
import=common/keycloak
```

3. Copy the file `themes/base/admin/resources/partials/user-attributes.html` into the a mirror directory in your theme: `themes/mytheme/admin/resources/partials/user-attributes.html`. What you are doing here is overriding the user attribute entry page in the admin console and putting in what attributes you want. This file already contains an example of entering address data. You can remove this if you want and replace it with something else. Also, if you want to edit this file directly instead of creating a new theme, you can.

4. In the `user-attributes.html` file add your custom user attribute entry form item. For example

```
    <div class="form-group clearfix block">
        <label class="col-sm-2 control-label" for="mobile">Mobile</label>
        <div class="col-sm-6">
              <input ng-model="user.attributes.mobile" class="form-control"
type="text" name="mobile" id="mobile" />
       </div>
         <span tooltip-placement="right" tooltip="Mobile number." class="fa
fa-info-circle"></span>
    </div>
```

   The `ng-model` names the user attribute you will store in the database and must have the form of `user.attributes.ATTR_NAME`.

5. Change the theme for the admin console. Save it, then refresh your browser, and you should now see these fields in the User detail page for any user.

## 32.2. In registration page

To be able to enter custom attributes in the registration page, take the following steps

1. Create a new theme within the `themes/mytheme/login` directory in your distribution. Where `mytheme` is whatever you want to name your theme.
2. Create a `theme.properties` file in this directory that extends the main admin console theme.

```
parent=keycloak
import=common/keycloak
styles=  ../patternfly/lib/patternfly/css/patternfly.css  ../patternfly/css/
login.css ../patternfly/lib/zocial/zocial.css css/login.css
```

3. Copy the file `themes/base/login/register.ftl` into the a mirror directory in your theme: `themes/mytheme/login/register.ftl`. What you are doing here is overriding the registration page and adding what attributes you want. This file already contains an example of entering address data. You can remove this if you want and replace it with something else. Also, if you want to edit this file directly instead of creating a new theme, you can.
4. In the `register.ftl` file add your custom user attribute entry form item. For example

```
<div class="form-group">
   <div class="${properties.kcLabelWrapperClass!}">
                                      <label     for="user.attributes.mobile"
 class="${properties.kcLabelClass!}">Mobile number</label>
   </div>

   <div class="col-sm-10 col-md-10">
                 <input   type="text"   class="${properties.kcInputClass!}"
 id="user.attributes.mobile" name="user.attributes.mobile"/>
   </div>
</div>
```

   Make sure the input field id ane name match the user attribute you want to store in the database. This must have the form of `user.attributes.ATTR_NAME`. You might also want to replace the label text with a message property. This will help later if you want to internationalize your pages.

5. Change the theme for the login to your new theme. Save it, then refresh your browser, and you should now see these fields in the registration.

## 32.3. In user account profile page

To be able to manage custom attributes in the user account profile page, take the following steps

1. Create a new theme within the `themes/mytheme/account` directory in your distribution. Where `mytheme` is whatever you want to name your theme.

2. Create a `theme.properties` file in this directory that extends the main admin console theme.

```
parent=patternfly
import=common/keycloak

styles=  ../patternfly/lib/patternfly/css/patternfly.css  ../patternfly/css/
account.css css/account.css
```

3. Copy the file `themes/base/account/account.ftl` into the a mirror directory in your theme: `themes/mytheme/account/account.ftl`. What you are doing here is overriding the profile page and adding what attributes you want to manage. This file already contains an example of entering address data. You can remove this if you want and replace it with something else. Also, if you want to edit this file directly instead of creating a new theme, you can.

4. In the `account.ftl` file add your custom user attribute entry form item. For example

```
<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!'')?html}"/>
   </div>
</div>
```

   Make sure the input field id ane name match the user attribute you want to store in the database. This must have the form of `user.attributes.ATTR_NAME`. You might also want to replace the label text with a message property. This will help later if you want to internationalize your pages.

5. Change the theme for the account to your new theme. Save it, then refresh your browser, and you should now see these fields in the account profile page.

# Chapter 33. 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. Basicall 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" sub-menu item. This is the place where you can control how a OIDC ID Token, Access Token, and SAML login response assertions look like. When you click on this you'll see some default mappers that have been set up for you. Clicking the "Add Builtin" button gives you the option to add other preconfigured mappers. Clicking on "Create" allows you to define your own protocol mappers. The tooltips are very helpful to learn exactly what you can do to tailor your tokens and assertions. They should be enough to guide you through the process.

# Chapter 34. Custom Authentication, Registration, and Required Actions

Keycloak comes out of the box with a bunch of different authentication mechanisms: kerberos, password, and otp. These mechanisms may not meet all of your requirements and you may want to plug in your own custom ones. Keycloak provides an authentication SPI that you can use to write new plugins. The admin console supports applying, ordering, and configuring these new mechanisms.

Keycloak also supports a simple registration form. Different aspects of this form can be enabled and disabled i.e. Recaptcha support can be turned off and on. The same authentication SPI can be used to add another page to the registration flow or reimplement it entirely. There's also an additional fine-grain SPI you can use to add specific validations and user extensions to the built in registration form.

A required action in Keycloak is an action that a user has to perform after he authenticates. After the action is performed successfully, the user doesn't have to perform the action again. Keycloak comes with some built in required actions like "reset password". This action forces the user to change their password after they have logged in. You can write and plug in your own required actions.

## 34.1. Terms

To first learn about the Authentication SPI, let's go over some of the terms used to describe it.

Authentication Flow
: A flow is a container for all authentications that must happen during login or registration. If you go to the admin console authentication page, you can view all the defined flows in the system and what authenticators they are made up of. Flows can contain other flows. You can also bind a new different flow for browser login, direct grant access, and registration.

Authenticator
: An authenticator is a pluggable component that hold the logic for performing the authentication or action within a flow. It is usually a singleton.

Execution
: An execution is an object that binds the authenticator to the flow and the authenticator to the configuration of the authenticator. Flows contain execution entries.

Execution Requirement
: Each execution defines how an authenticator behaves in a flow. The requirement defines whether the authenticator is enabled, disabled, optional, required, or an alternative. An

alternative requirement means that the authentiactor is optional unless no other alternative authenticator is successful in the flow. For example, cookie authentication, kerberos, and the set of all login forms are all alternative. If one of those is successful, none of the others are executed.

Authenticator Config
: This object defines the configuration for the Authenticator for a specific execution within an authentication flow. Each execution can have a different config.

Required Action
: After authentication completes, the user might have one or more one-time actions he must complete before he is allowed to login. The user might be required to set up an OTP token generator or reset an expired password or even accept a Terms and Conditions document.

## 34.2. Algorithm Overview

Let's talk about how this all works for browser login. Let's assume the following flows, executions and sub flows.

```
Cookie - ALTERNATIVE
Kerberos - ALTERNATIVE
Forms Subflow - ALTERNATIVE
           Username/Password Form - REQUIRED
           OTP Password Form - OPTIONAL
```

In the top level of the form we have 3 executions of which all are alternatively required. This means that if any of these are successful, then the others do not have to execute. The Username/Password form is not executed if there is an SSO Cookie set or a successful Kerberos login. Let's walk through the steps from when a client first redirects to keycloak to authenticate the user.

1. The OpenID Connect or SAML protocol provider unpacks relevent data, verifies the client and any signatures. It creates a ClientSessionModel. It looks up what the browser flow should be, then starts executing the flow.

2. The flow looks at the cookie execution and sees that it is an alternative. It loads the cookie provider. It checks to see if the cookie provider requires that a user already be associated with the client session. Cookie provider does not require a user. If it did, the flow would abort and the user would see an error screen. Cookie provider then executes. Its purpose is to see if there is an SSO cookie set. If there is one set, it is validated and the UserSessionModel is verified and associated with the ClientSessionModel. The Cookie provider returns a success() status if the SSO cookie exists and is validated. 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. If there is no SSO cookie, the cookie provider returns with a status of attempted(). This means there was no error condition, but no success either. The provider tried, but the request just wasn't set up to handle this authenticator.

3. Next the flow looks at the Kerberos execution. This is also an alternative. The kerberos provider also does not require a user to be already set up and associated with the ClientSessionModel so this provider is executed. Kerberos uses the SPNEGO browser protocol. This requires a series of challenge/responses between the server and client exchanging negotiation headers. The kerberos provider does not see any negotiate header, so it assumes that this is the first interaction between the server and client. It therefore creates an HTTP challenge response to the client and sets a forceChallenge() status. A forceChallenge() means that this HTTP response cannot be ignored by the flow and must be returned to the client. If instead the provider returned a challenge() status, the flow would hold the challenge response until all other alternatives are attempted. So, in this initial phase, the flow would stop and the challenge response would be sent back to the browser. If the browser then responds with a successful negotiate header, the provider associates the user with the ClientSession and the flow ends because the rest of the executions on this level of the flow are all alternatives. Otherwise, again, the kerberos provider sets an attempted() status and the flow continues.

4. The next execution is a subflow called Forms. The executions for this subflow are loaded and the same processing logic occurs

5. The first execution in the Forms subflow is the UsernamePassword provider. This provider also does not require for a user to already be associated with the flow. This provider creates challenge HTTP response and sets its status to challenge(). This execution is required, so the flow honors this challenge and sends the HTTP response back to the browser. This response is a rendering of the Username/Password HTML page. The user enters in their username and password and clicks submit. This HTTP request is directed to the UsernamePassword provider. If the user entered an invalid username or password, a new challenge response is created and a status of failureChallenge() is set for this execution. A failureChallenge() means that there is a challenge, but that the flow should log this as an error in the error log. This error log can be used to lock accounts or IP Addresses that have had too many login failures. If the username and password is valid, the provider associated the UserModel with the ClientSessionModel and returns a status of success()

6. The next execution is the OTP Form. This provider requires that a user has been associated with the flow. This requirement is satisfied because the UsernamePassword provider already associated the user with the flow. Since a user is required for this provider, the provider is also asked if the user is configured to use this provider. If user is not configured, and this execution is required, then the flow will then set up a required action that the user must perform after authentication is complete. For OTP, this means the OTP setup page. If the execution was optional, then this execution is skipped.

7. After the flow is complete, the authentication processor creates a UserSessionModel and associates it with the ClientSessionModel. It then checks to see if the user is required to complete any required actions before logging in.

8. First, each required action's evaluateTriggers() method is called. This allows the required action provider to figure out if there is some state that might trigger the action to be fired. For example, if your realm has a password expiration policy, it might be triggered by this method.

9. Each required action associated with the user that has its requiredActionChallenge() method called. Here the provider sets up an HTTP response which renders the page for the required action. This is done by setting a challenge status.

10. If the required action is ultimately successful, then the required action is removed from the user's require actions list.

11. After all required actions have been resolved, the user is finally logged in.

## 34.3. Authenticator SPI Walk Through

In this section, we'll take a look at the Authenticator interface. For this, we are going to implement an authenticator that requires that a user enter in the answer to a secret question like "What is your mother's maiden name?". This example is fully implemented and contained in the examples/providers/authenticator directory of the demo distribution of Keycloak.

The classes you must implement are the org.keycloak.authentication.AuthenticatorFactory and Authenticator interfaces. The Authenticator interface defines the logic. The AuthenticatorFactory is responsible for creating instances of an Authenticator. They both extend a more generic Provider and ProviderFactory set of interfaces that other Keycloak components like User Federation do.

### 34.3.1. Packaging Classes and Deployment

You will package your classes within a single jar. This jar must contain a file named `org.keycloak.authentication.AuthenticatorFactory` and must be contained in the `META-INF/services/` directory of your jar. This file must list the fully qualified classname of each AuthenticatorFactory implementation you have in the jar. For example:

```
org.keycloak.examples.authenticator.SecretQuestionAuthenticatorFactory
   org.keycloak.examples.authenticator.AnotherProviderFactory
```

This services/ file is used by Keycloak to scan the providers it has to load into the system.

To deploy this jar, just copy it to the standalone/configuration/providers directory.

### 34.3.2. Implementing an Authenticator

When implementing the Authenticator interface, the first method that needs to be implemented is the requiresUser() method. For our example, this method must return true as we need to validate

the secret question associated with the user. A provider like kerberos would return false from this method as it can resolve a user from the negotiate header. This example, however, is validating a specific credential of a specific user.

The next method to implement is the configuredFor() method. This method is responsible for determining if the user is configured for this particular authenticator. For this example, we need to check if the answer to the secret question has been set up by the user or not. In our case we are storing this information, hashed, within a UserCredentialValueModel within the UserModel (just like passwords are stored). Here's how we do this very simple check:

```
   @Override
     public boolean configuredFor(KeycloakSession session, RealmModel realm,
UserModel user) {
         return session.users().configuredForCredentialType("secret_question",
realm, user);
   }
```

The configuredForCredentialType() call queries the user to see if it supports that credential type.

The next method to implement on the Authenticator is setRequiredActions(). If configuredFor() returns false and our example authenticator is required within the flow, this method will be called. It is responsible for registering any required actions that must be performed by the user. In our example, we need to register a required action that will force the user to set up the answer to the secret question. We will implement this required action provider later in this chapter. Here is the implementation of the setRequiredActions() method.

```
   @Override
    public void setRequiredActions(KeycloakSession session, RealmModel realm,
UserModel user) {
       user.addRequiredAction("SECRET_QUESTION_CONFIG");
   }
```

Now we are getting into the meat of the Authenticator implementation. The next method to implement is authenticate(). This is the initial method the flow invokes when the execution is first visited. What we want is that if a user has answered the secret question already on their browser's machine, then the user doesn't have to answer the question again, making that machine "trusted". The authenticate() method isn't responsible for processing the secret question form. Its sole purpose is to render the page or to continue the flow.

```
   @Override
   public void authenticate(AuthenticationFlowContext context) {
```

```
        if (hasCookie(context)) {
           context.success();
           return;
        }
      Response challenge = loginForm(context).createForm("secret_question.ftl");
        context.challenge(challenge);
    }
```

The hasCookie() method checks to see if there is already a cookie set on the browser which indicates that the secret question has already been answered. If that returns true, we just mark this execution's status as SUCCESS using the AuthenticationFlowContext.success() method and returning from the authentication() method.

If the hasCookie() method returns false, we must return a response that renders the secret question HTML form. AuthenticationFlowContext has a form() method that initializes a Freemarker page builder with appropriate base information needed to build the form. This page builder is called `org.keycloak.login.LoginFormsProvider`. the LoginFormsProvider.createForm() method loads a Freemarker template file from your login theme. Additionally you can call the LoginFormsProvider.setAttribute() method if you want to pass additional information to the Freemarker template. We'll go over this later.

Calling LoginFormsProvider.createForm() returns a JAX-RS Response object. We then call AuthenticationFlowContext.challenge() passing in this response. This sets the status of the execution as CHALLENGE and if the execution is Required, this JAX-RS Response object will be sent to the browser.

So, the HTML page asking for the answer to a secret question is displayed to the user and the user enteres in the answer and clicks submit. The action URL of the HTML form will send an HTTP request to the flow. The flow will end up invoking the action() method of our Authenticator implementation.

```
    @Override
    public void action(AuthenticationFlowContext context) {
        boolean validated = validateAnswer(context);
        if (!validated) {
           Response challenge = context.form()
                                .setError("badSecret")
                                .createForm("secret-question.ftl");
          context.failureChallenge(AuthenticationFlowError.INVALID_CREDENTIALS,
 challenge);
           return;
        }
        setCookie(context);
        context.success();
    }
```

If the answer is not valid, we rebuild the HTML Form with an additional error message. We then call AuthenticationFlowContext.failureChallenge() passing in the reason for the value and the JAX-RS response. failureChallenge() works the same as challenge(), but it also records the failure so it can be analyzed by any attack detection service.

If validation is successful, then we set a cookie to remember that the secret question has been answered and we call AuthenticationFlowContext.success().

The last thing I want to go over is the setCookie() method. This is an example of providing configuration for the Authenticator. In this case we want the max age of the cookie to be configurable.

```
    protected void setCookie(AuthenticationFlowContext context) {
        AuthenticatorConfigModel config = context.getAuthenticatorConfig();
        int maxCookieAge = 60 * 60 * 24 * 30; // 30 days
        if (config != null) {
                                                                maxCookieAge     =
 Integer.valueOf(config.getConfig().get("cookie.max.age"));

        }
        ... set the cookie ...
    }
```

We obtain an AuthenticatorConfigModel from the AuthenticationFlowContext.getAuthenticatorConfig() method. If configuration exists we pull the max age config out of it. We will see how we can define what should be configured when we talk about the AuthenticatorFactory implementation. The config values can be defined within the admin console if you set up config definitions in your AuthenticatorFactory implementation.

### 34.3.3. Implementing an AuthenticatorFactory

The next step in this process is to implement an AuthenticatorFactory. This factory is responsible for instantiating an Authenticator. It also provides deployment and configuration metadata about the Authenticator.

The getId() method is just the unique name of the component. The create() method is called by the runtime to allocate and process the Authenticator.

```
public class SecretQuestionAuthenticatorFactory implements AuthenticatorFactory,
 ConfigurableAuthenticatorFactory {
```

```
    public static final String PROVIDER_ID = "secret-question-authenticator";
        private static final SecretQuestionAuthenticator SINGLETON = new
 SecretQuestionAuthenticator();

    @Override
    public String getId() {
        return PROVIDER_ID;
    }

    @Override
    public Authenticator create(KeycloakSession session) {
        return SINGLETON;
    }
```

The next thing the factory is responsible for is specify the allowed requirement switches. While there are four different requirement types: ALTERNATIVE, REQUIRED, OPTIONAL, DISABLED, AuthenticatorFactory implementations can limit which requirement options are shown in the admin console when defining a flow. In our example, we're going to limit our requirement options to REQUIRED and DISABLED.

```
  private static AuthenticationExecutionModel.Requirement[] REQUIREMENT_CHOICES
 = {
            AuthenticationExecutionModel.Requirement.REQUIRED,
            AuthenticationExecutionModel.Requirement.DISABLED
    };
    @Override
    public AuthenticationExecutionModel.Requirement[] getRequirementChoices() {
        return REQUIREMENT_CHOICES;
    }
```

The AuthenticatorFactory.isUserSetupAllowed() is a flag that tells the flow manager whether or not Authenticator.setRequiredActions() method will be called. If an Authenticator is not configured for a user, the flow manager checks isUserSetupAllowed(). If it is false, then the flow aborts with an error. If it returns true, then the flow manager will invoke Authenticator.setRequiredActions().

```
    @Override
    public boolean isUserSetupAllowed() {
        return true;
    }
```

The next few methods define how the Authenticator can be configured. The isConfigurable() method is a flag which specifies to the admin console on whether the Authenticator can be configured within a flow. The getConfigProperties() method returns a list of ProviderConfigProperty objects. These objects define a specific configuration attribute.

```
    @Override
    public List<ProviderConfigProperty> getConfigProperties() {
        return configProperties;
    }

     private static final List<ProviderConfigProperty> configProperties = new
 ArrayList<ProviderConfigProperty>();

    static {
        ProviderConfigProperty property;
        property = new ProviderConfigProperty();
        property.setName("cookie.max.age");
        property.setLabel("Cookie Max Age");
        property.setType(ProviderConfigProperty.STRING_TYPE);
      property.setHelpText("Max age in seconds of the SECRET_QUESTION_COOKIE.");
        configProperties.add(property);
    }
```

Each ProviderConfigProperty defines the name of the config property. This is the key used in the config map stored in AuthenticatorConfigModel. The label defines how the config option will be displayed in the admin console. The type defines if it is a String, Boolean, or other type. The admin console will display different UI inputs depending on the type. The help text is what will be shown in the tooltip for the config attribute in the admin console. Read the javadoc of ProviderConfigProperty for more detail.

The rest of the methods are for the admin console. getHelpText() is the tooltip text that will be shown when you are picking the Authenticator you want to bind to an execution. getDisplayType() is what text that will be shown in the admin console when listing the Authenticator. getReferenceCategory() is just a category the Authenticator belongs to.

### 34.3.4. Adding Authenticator Form

Keycloak comes with a Freemarker *theme and template engine*. The createForm() method you called within authenticate() of your Authenticator class, builds an HTML page from a file within your login theme: secret-question.ftl. This file should be placed in the login theme with all the other .ftl files you see for login.

Let's take a bigger look at secret-question.ftl Here's a small code snippet:

```
             <form  id="kc-totp-login-form"  class="${properties.kcFormClass!}"
 action="${url.loginAction}" method="post">
            <div class="${properties.kcFormGroupClass!}">
                <div class="${properties.kcLabelWrapperClass!}">
                        <label for="totp" class="${properties.kcLabelClass!}">
${msg("loginSecretQuestion")}</label>
                </div>

                <div class="${properties.kcInputWrapperClass!}">
                            <input  id="totp"  name="secret_answer"  type="text"
 class="${properties.kcInputClass!}" />
                </div>
            </div>
```

Any piece of text enclosed in `${}` corresponds to an attribute or template funtion. If you see the form's action, you see it points to `${url.loginAction}`. This value is automatically generated when you invoke the AuthenticationFlowContext.form() method. You can also obtain this value by calling the AuthenticationFlowContext.getActionURL() method in Java code.

You'll also see `${properties.someValue}`. These correspond to properties defined in your theme.properties file of our theme. `${msg("someValue")}` corresponds to the internationalized message bundles (.properties files) included with the login theme messages/ directory. If you're just using english, you can just add the value of the `loginSecretQuestion` value. This should be the question you want to ask the user.

When you call AuthenticationFlowContext.form() this gives you a LoginFormsProvider instance. If you called, `LoginFormsProvider.setAttribute("foo", "bar")`, the value of "foo" would be available for reference in your form as `${foo}`. The value of an attribute can be any Java bean as well.

### 34.3.5. Adding Authenticator to a Flow

Adding an Authenticator to a flow must be done in the admin console. If you go to the Authentication menu item and go to the Flow tab, you will be able to view the currently defined flows. You cannot modify an built in flows, so, to add the Authenticator we've created you have to copy an existing flow or create your own. I'm hoping the UI is intuitive enough so that you can figure out for yourself how to create a flow and add the Authenticator.

After you've created your flow, you have to bind it to the login action you want to bind it to. If you go to the Authentication menu and go to the Bindings tab you will see options to bind a flow to the browser, registration, or direct grant flow.

## 34.4. Required Action Walkthrough

In this section we will discuss how to define a required action. In the Authenticator section you may have wondered, "How will we get the user's answer to the secret question entered into the

system?". As we showed in the example, if the answer is not set up, a required action will be triggered. This section discusses how to implement the required action for the Secret Question Authenticator.

### 34.4.1. Packaging Classes and Deployment

You will package your classes within a single jar. This jar does not have to be separate from other provider classes but it must contain a file named `org.keycloak.authentication.RequiredActionFactory` and must be contained in the `META-INF/services/` directory of your jar. This file must list the fully qualified classname of each RequiredActionFactory implementation you have in the jar. For example:

```
org.keycloak.examples.authenticator.SecretQuestionRequiredActionFactory
```

This services/ file is used by Keycloak to scan the providers it has to load into the system.

To deploy this jar, just copy it to the standalone/configuration/providers directory.

### 34.4.2. Implement the RequiredActionProvider

Required actions must first implement the RequiredActionProvider interface. The RequiredActionProvider.requiredActionChallenge() is the initial call by the flow manager into the required action. This method is responsible for rendering the HTML form that will drive the required action.

```
   @Override
   public void requiredActionChallenge(RequiredActionContext context) {
                                                Response      challenge      =
context.form().createForm("secret_question_config.ftl");
       context.challenge(challenge);

   }
```

You see that RequiredActionContext has similar methods to AuthenticationFlowContext. The form() method allows you to render the page from a Freemarker template. The action URL is preset by the call to this form() method. You just need to reference it within your HTML form. I'll show you this later.

The challenge() method notifies the flow manager that a required action must be executed.

The next method is responsible for processing input from the HTML form of the required action. The action URL of the form will be routed to the RequiredActionProvider.processAction() method

```
    @Override
    public void processAction(RequiredActionContext context) {
                                                    String      answer      =
 (context.getHttpRequest().getDecodedFormParameters().getFirst("answer"));
        UserCredentialValueModel model = new UserCredentialValueModel();
        model.setValue(answer);
        model.setType(SecretQuestionAuthenticator.CREDENTIAL_TYPE);
        context.getUser().updateCredentialDirectly(model);
        context.success();
    }
```

The answer is pulled out of the form post. A UserCredentialValueModel is created and the type and value of the credential are set. Then UserModel.updateCredentialDirectly() is invoked. Finally, RequiredActionContext.success() notifies the container that the required action was successful.

### 34.4.3. Implement the RequiredActionFactory

This class is really simple. It is just responsible for creating the required actin provider instance.

```
public       class       SecretQuestionRequiredActionFactory       implements
 RequiredActionFactory {

        private  static  final  SecretQuestionRequiredAction  SINGLETON  =  new
 SecretQuestionRequiredAction();

    @Override
    public RequiredActionProvider create(KeycloakSession session) {
        return SINGLETON;
    }


    @Override
    public String getId() {
        return SecretQuestionRequiredAction.PROVIDER_ID;
    }

    @Override
    public String getDisplayText() {
        return "Secret Question";
    }
```

The getDisplayText() method is just for the admin console when it wants to display a friendly name for the required action.

### 34.4.4. Enable Required Action

The final thing you have to do is go into the admin console. Click on the Authentication left menu. Click on the Required Actions tab. Click on the Register button and choose your new Required Action. Your new required action should now be displayed and enabled in the required actions list.

## 34.5. Modifying/Extending the Registration Form

It is entirely possible for you to implement your own flow with a set of Authenticators to totally change how regisration is done in Keycloak. But what you'll usually want to do is just add a little bit of validation to the out of the box registration page. An additional SPI was created to be able to do this. It basically allows you to add validation of form elements on the page as well as to initialize UserModel attributes and data after the user has been registered. We'll look at both the implementation of the user profile registration processing as well as the registration Google Recaptcha plugin.

### 34.5.1. Implementation FormAction Interface

The core interface you have to implement is the FormAction interface. A FormAction is responsible for rendering and processing a portion of the page. Rendering is done in the buildPage() method, validation is done in the validate() method, post validation operations are done in success(). Let's first take a look at buildPage() method of the Recaptcha plugin.

```
    @Override
    public void buildPage(FormContext context, LoginFormsProvider form) {
      AuthenticatorConfigModel captchaConfig = context.getAuthenticatorConfig();
         if (captchaConfig == null || captchaConfig.getConfig() == null
                 || captchaConfig.getConfig().get(SITE_KEY) == null
                 || captchaConfig.getConfig().get(SITE_SECRET) == null
                 ) {
                                          form.addError(new    FormMessage(null,
 Messages.RECAPTCHA_NOT_CONFIGURED));
             return;
         }
         String siteKey = captchaConfig.getConfig().get(SITE_KEY);
         form.setAttribute("recaptchaRequired", true);
         form.setAttribute("recaptchaSiteKey", siteKey);
         form.addScript("https://www.google.com/recaptcha/api.js");
    }
```

The Recaptcha buildPage() method is a callback by the form flow to help render the page. It receives a form parameter which is a LoginFormsProvider. You can add additional attributes to the form provider so that they can be displayed in the HTML page generated by the registration Freemarker template.

The code above is from the registration recaptcha plugin. Recaptcha requires some specific settings that must be obtained from configuration. FormActions are configured in the exact same was as Authenticators are. In this example, we pull the Google Recaptcha site key from configuration and add it as an attribute to the form provider. Our regstration template file can read this attribute now.

Recaptcha also has the requirement of loading a javascript script. You can do this by calling LoginFormsProvider.addScript() passing in the URL.

For user profile processing, there is no additional information that it needs to add to the form, so its buildPage() method is empty.

The next meaty part of this interface is the validate() method. This is called immediately upon receiving a form post. Let's look at the Recaptcha's plugin first.

```
   @Override
   public void validate(ValidationContext context) {
                            MultivaluedMap<String,    String>    formData    =
context.getHttpRequest().getDecodedFormParameters();
       List<FormMessage> errors = new ArrayList<>();
       boolean success = false;

       String captcha = formData.getFirst(G_RECAPTCHA_RESPONSE);
       if (!Validation.isBlank(captcha)) {
                                  AuthenticatorConfigModel   captchaConfig   =
context.getAuthenticatorConfig();
           String secret = captchaConfig.getConfig().get(SITE_SECRET);

           success = validateRecaptcha(context, success, captcha, secret);
       }
       if (success) {
           context.success();
       } else {
           errors.add(new FormMessage(null, Messages.RECAPTCHA_FAILED));
           formData.remove(G_RECAPTCHA_RESPONSE);
           context.validationError(formData, errors);
           return;


       }
```

```
    }
```

Here we obtain the form data that the Recaptcha widget adds to the form. We obtain the Recaptcha secret key from configuration. We then validate the recaptcha. If successful, ValidationContext.success() is called. If not, we invoke ValidationContext.validationError() passing in the formData (so the user doesn't have to re-enter data), we also specify an error message we want displayed. The error message must point to a message bundle property in the internationalized message bundles. For other registration extensions validate() might be validating the format of a form element, i.e. an alternative email attribute.

Let's also look at the user profile plugin that is used to validate email address and other user information when registering.

```
    @Override
    public void validate(ValidationContext context) {
                              MultivaluedMap<String,    String>    formData    =
context.getHttpRequest().getDecodedFormParameters();
        List<FormMessage> errors = new ArrayList<>();

        String eventError = Errors.INVALID_REGISTRATION;

                                                                                if
(Validation.isBlank(formData.getFirst((RegistrationPage.FIELD_FIRST_NAME)))) {
                  errors.add(new  FormMessage(RegistrationPage.FIELD_FIRST_NAME,
Messages.MISSING_FIRST_NAME));
        }

                                                                                if
(Validation.isBlank(formData.getFirst((RegistrationPage.FIELD_LAST_NAME)))) {
                   errors.add(new  FormMessage(RegistrationPage.FIELD_LAST_NAME,
Messages.MISSING_LAST_NAME));
        }

        String email = formData.getFirst(Validation.FIELD_EMAIL);
        if (Validation.isBlank(email)) {
                       errors.add(new  FormMessage(RegistrationPage.FIELD_EMAIL,
Messages.MISSING_EMAIL));
        } else if (!Validation.isEmailValid(email)) {
            formData.remove(Validation.FIELD_EMAIL);
                       errors.add(new  FormMessage(RegistrationPage.FIELD_EMAIL,
Messages.INVALID_EMAIL));
        }

                        if   (context.getSession().users().getUserByEmail(email,
context.getRealm()) != null) {
```

```
            formData.remove(Validation.FIELD_EMAIL);
                      errors.add(new  FormMessage(RegistrationPage.FIELD_EMAIL,
 Messages.EMAIL_EXISTS));
        }

        if (errors.size() > 0) {
            context.validationError(formData, errors);
            return;

        } else {
            context.success();
        }
    }
```

As you can see, this validate() method of user profile processing makes sure that the email, first, and last name are filled in in the form. It also makes sure that email is in the right format. If any of these validations fail, an error message is queued up for rendering. Any fields in error are removed from the form data. Error messages are represented by the FormMessage class. The first parameter of the constructor of this class takes the HTML element id. The input in error will be highlighted when the form is re-rendered. The second parameter is a message reference id. This id must correspond to a property in one of the localized message bundle files. in the theme.

After all validations have been processed then, the form flow then invokes the FormAction.success() method. For recaptcha this is a no-op, so we won't go over it. For user profile processing, this method fills in values in the registered user.

```
    @Override
    public void success(FormContext context) {
        UserModel user = context.getUser();
                              MultivaluedMap<String,    String>    formData    =
 context.getHttpRequest().getDecodedFormParameters();
       user.setFirstName(formData.getFirst(RegistrationPage.FIELD_FIRST_NAME));
        user.setLastName(formData.getFirst(RegistrationPage.FIELD_LAST_NAME));
        user.setEmail(formData.getFirst(RegistrationPage.FIELD_EMAIL));
    }
```

Pretty simple implementation. The UserModel of the newly registered user is obtained from the FormContext. The appropriate methods are called to initialize UserModel data.

Finally, you are also required to define a FormActionFactory class. This class is implemented similarly to AuthenticatorFactory, so we won't go over it.

### 34.5.2. Packaging the Action

You will package your classes within a single jar. This jar must contain a file named `org.keycloak.authentication.FormActionFactory` and must be contained in the `META-INF/services/` directory of your jar. This file must list the fully qualified classname of each FormActionFactory implementation you have in the jar. For example:

```
org.keycloak.authentication.forms.RegistrationProfile
org.keycloak.authentication.forms.RegistrationRecaptcha
```

This services/ file is used by Keycloak to scan the providers it has to load into the system.

To deploy this jar, just copy it to the standalone/configuration/providers directory.

### 34.5.3. Adding FormAction to the Registration Flow

Adding an FormAction to a registration page flow must be done in the admin console. If you go to the Authentication menu item and go to the Flow tab, you will be able to view the currently defined flows. You cannot modify an built in flows, so, to add the Authenticator we've created you have to copy an existing flow or create your own. I'm hoping the UI is intuitive enough so that you can figure out for yourself how to create a flow and add the FormAction.

Basically you'll have to copy the registration flow. Then click Actions menu to the right of the Registration Form, and pick "Add Execution" to add a new execution. You'll pick the FormAction from the selection list. Make sure your FormAction comes after "Registration User Creation" by using the down errors to move it if your FormAction isn't already listed after "Registration User Creation". You want your FormAction to come after user creation because the success() method of Regsitration User Creation is responsible for creating the new UserModel.

After you've created your flow, you have to bind it to registration. If you go to the Authentication menu and go to the Bindings tab you will see options to bind a flow to the browser, registration, or direct grant flow.

## 34.6. Modifying Forgot Password/Credential Flow

Keycloak also has a specific authentication flow for forgot password, or rather credential reset initiated by a user. If you go to the admin console flows page, there is a "reset credentials" flow. By default, Keycloak asks for the email or username of the user and sends an email to them. If the user clicks on the link, then they are able to reset both their password and OTP (if an OTP has been set up). You can disable automatic OTP reset by disabling the "Reset OTP" authenticator in the flow.

You can add additional functionality to this flow as well. For example, many deployments would like for the user to answer one or more secret questions in additional to sending an email with a

link. You could expand on the secret question example that comes with the distro and incorporate it into the reset credential flow.

One thing to note if you are extending the reset credentials flow. The first "authenticator" is just a page to obtain the username or email. If the username or email exists, then the AuthenticationFlowContext.getUser() will return the located user. Otherwise this will be null. This form *WILL NOT* re-ask the user to enter in an email or username if the previous email or username did not exist. You need to prevent attackers from being able to guess valid users. So, if AuthenticationFlowContext.getUser() returns null, you should proceed with the flow to make it look like a valid user was selected. I suggest that if you want to add secret questions to this flow, you should ask these questions after the email is sent. In other words, add your custom authenticator after the "Send Reset Email" authenticator.

## 34.7. Modifying First Broker Login Flow

First Broker Login flow is used during first login with some identity provider. Term `First Login` means that there is not yet existing Keycloak account linked with the particular authenticated identity provider account. More details about this flow are in the *Identity provider chapter*.

## 34.8. Authentication of clients

Keycloak actually supports pluggable authentication for *OpenID Connect* [http://openid.net/specs/openid-connect-core-1_0.html] client applications. Authentication of client (application) is used under the hood by the *Keycloak adapter* during sending any backchannel requests to the Keycloak server (like the request for exchange code to access token after successful authentication or request to refresh token). But the client authentication can be also used directly by you during *Direct Access grants* or during *Service account* authentication.

### 34.8.1. Default implementations

Actually Keycloak has 2 builtin implementations of client authentication:

Traditional authentication with client_id and client_secret
: This is default mechanism mentioned in the *OpenID Connect* [http://openid.net/specs/openid-connect-core-1_0.html] or *OAuth2* [http://tools.ietf.org/html/rfc6749] specification and Keycloak supports it since it's early days. The public client needs to include `client_id` parameter with it's ID in the POST request (so it's defacto not authenticated) and the confidential client needs to include `Authorization: Basic` header with the clientId and clientSecret used as username and password.

  For the public/javascript clients, you don't need to add anything into your keycloak.json configuration file. For the confidential (server) clients, you need to add something like this:

```
"credentials": {
```

```
    "secret": "mysecret"
}
```

where the `mysecret` needs to be replaced with the real value of client secret. You can obtain it from admin console from client configuration.

Authentication with signed JWT

This is based on the *JWT Bearer Token Profiles for OAuth 2.0* [https://tools.ietf.org/html/rfc7523] specification. The client/adapter generates the *JWT* [https://tools.ietf.org/html/rfc7519] and signs it with his private key. The Keycloak then verifies the signed JWT with the client's public key and authenticates client based on it.

To achieve this, you need those steps:

- Your client needs to have private key and Keycloak needs to have client public key. This can be either:
  - Generated in Keycloak admin console - In this case, Keycloak will generate pair of keys and it will save public key and certificate in it's DB. The keystore file with the private key will be downloaded and you need to save it in the location accessible to your client application
  - Uploaded in Keycloak admin console - This option is useful if you already have existing private key of your client. In this case, you just need to upload the public key and certificate to the Keycloak server.

  In both cases, the private key is not saved in Keycloak DB, but it's owned exclusively by your client. The Keycloak DB has just public key.
- As second step, you need to use the configuration like this in your `keycloak.json` adapter configuration:

```
"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
    }
}
```

The `client-keystore-file` is the location of the keystore file, which is either on classpath (for example if bundled in the WAR itself) or somewhere on the filesystem. Other options specify type of keystore and password of keystore itself and of the private key. Last option

`token-expiration` is the expiration of JWT in seconds. The token needs to be valid just for single request, so 10 seconds is usually sufficient.

See the demo example and especially the `examples/preconfigured-demo/service-account` for the example application showing service accounts authentication with both clientId +clientSecret and with signed JWT.

## 34.8.2. Implement your own client authenticator

For plug your own client authenticator, you need to implement few interfaces on both client (adapter) and server side.

Client side

Here you need to implement `org.keycloak.adapters.authentication.ClientCredentialsProvider` and put the implementation either to:

- your WAR file into WEB-INF/classes . But in this case, the implementation can be used just for this single WAR application
- Some JAR file, which will be added into WEB-INF/lib of your WAR
- Some JAR file, which will be used as jboss module and configured in jboss-deployment-structure.xml of your WAR.

In all cases, you also need to create the file `META-INF/services/org.keycloak.adapters.authentication.ClientCredentialsProvider` either in the WAR or in your JAR.

You also need to configure your clientCredentialsProvider in `keycloak.json` . See the javadoc for more details.

Server side

Here you need to implement `org.keycloak.authentication.ClientAuthenticatorFactory` and `org.keycloak.authentication.ClientAuthenticator` . You also need to add the file `META-INF/services/org.keycloak.authentication.ClientAuthenticatorFactory` with the name of the implementation classes. See *authenticators* for more details.

Finally you need to configure admin console . You need to create new client authentication flow and define execution with your authenticator (you can also add the builtin authenticators and configure requirements etc) and finally configure Clients binding . See *Adding Authenticator* for more details. Then you need to go to Client credentials tab and choose the method for authentication your client and configure client credentials (if possible).

# Chapter 35. Migration from older versions

To upgrade to a new version of Keycloak first download and install the new version of Keycloak. You then have to migrate the database, keycloak-server.json, providers, themes and applications from the old version.

## 35.1. Migrate database

Keycloak provides automatic migration of the database. It's highly recommended that you backup your database prior to upgrading Keycloak.

To enable automatic upgrading of the database if you're using a relational database make sure `databaseSchema` is set to `update` for `connectionsJpa`:

```
"connectionsJpa": {
    "default": {
        ...
        "databaseSchema": "update"
    }
}
```

For MongoDB do the same, but for `connectionsMongo`:

```
"connectionsMongo": {
    "default": {
        ...
        "databaseSchema": "update"
    }
}
```

When you start the server with this setting your database will automatically be migrated if the database schema has changed in the new version.

## 35.2. Migrate keycloak-server.json

You should copy `standalone/configuration/keycloak-server.json` from the old version to make sure any configuration changes you've done are added to the new installation. The version specific section below will list any changes done to this file that you have to do when upgrading from one version to another.

## 35.3. Migrate providers

If you have implemented any SPI providers you need to copy them to the new server. The version specific section below will mention if any of the SPI's have changed. If they have you may have to update your code accordingly.

## 35.4. Migrate themes

If you have created a custom theme you need to copy them to the new server. The version specific section below will mention if changes have been made to themes. If there is you may have to update your themes accordingly.

## 35.5. Migrate application

If you deploy applications directly to the Keycloak server you should copy them to the new server. For any applications including those not deployed directly to the Keycloak server you should upgrade the adapter. The version specific section below will mention if any changes are required to applications.

## 35.6. Version specific migration

### 35.6.1. Migrating to 1.9.0

#### Themes and providers directory moved

We've moved the themes and providers directories from `standalone/configuration/themes` and `standalone/configuration/providers` to `themes` and `providers` respectively. If you have added custom themes and providers you need to move them to the new location. You also need to update `keycloak-server.json` as it's changed due to this.

#### Adapter Subsystems only bring in dependencies if keycloak is on

Previously, if you had installed our saml or oidc keycloak subsystem adapters into Wildfly or JBoss EAP, we would automatically include Keycloak client jars into EVERY application irregardless if you were using Keycloak or not. These libraries are now only added to your deployment if you have keycloak authentication turned on for that adapter (via the subsystem, or auth-method in web.xml

#### Client Registration service endpoints moved

The Client Registration service endpoints have been moved from `{realm}/clients` to `{realm}/clients-registrations`.

#### Session state parameter in authentication response renamed

In the OpenID Connect authentication response we used to return the session state as `session-state` this is not correct according to the specification and has been renamed to `session_state`.

### Deprecated OpenID Connect endpoints

In 1.2 we deprecated a number of endpoints that where not consistent with the OpenID Connect specifications, these have now been removed. This also applies to the validate token endpoints that was replaced with the new introspect endpoint in 1.8.

### Updates to theme templates

Feedback in template.ftl has been moved and format has changed slightly.

### Module and Source Code Re-org

Most of our modules and source code have been consolidated into two maven modules: keycloak-server-spi and keycloak-services. SPI interfaces are in server-spi, implementations are in keycloak-services. All JPA dependent modules have been consolidated under keycloak-model-jpa. Same goes with mongo and infinispan under modules keycloak-model-mongo and keycloak-model-infinispan.

### For adapters, session id changed after login

To plug a security attack vector, for platforms that support it (Tomcat 8, Undertow/Wildfly, Jetty 9), the keycloak oidc and saml adapters will change the session id after login. You can turn off this behavior check adapter config switches.

### SAML SP Client Adapter Changes

Keycloak SAML SP Client Adapter now requires a specific endpoint, `/saml` to be registered with your IDP.

## 35.6.2. Migrating to 1.8.0

### Admin account

In previous releases we shipped with a default admin user with a default password, this has now been removed. If you are doing a new installation of 1.8 you will have to create an admin user as a first step. This can be done easily by following the steps in *Admin User*.

### OAuth2 Token Introspection

In order to add more compliance with OAuth2 specification, we added a new endpoint for token introspection. The new endpoint can reached at `/realms/{realm}/protocols/openid-connect/token/introspect` and it is solely based on `RFC-7662`.

The `/realms/{realm}/protocols/openid-connect/validate` endpoint is now deprecated and we strongly recommend you to move to the new introspection endpoint as soon as possible. The reason for this change is that RFC-7662 provides a more standard and secure introspection endpoint.

The new token introspection URL can now be obtained from OpenID Connect Provider's configuration at `/realms/{realm}/.well-known/openid-configuration`. There you will find a claim with name `token_introspection_endpoint` within the response. Only `confidential clients` are allowed to invoke the new endpoint, where these clients will be usually acting as a resource server and looking for token metadata in order to perform local authorization checks.

## 35.6.3. Migrating to 1.7.0.CR1

### Direct access grants disabled by default for clients

In order to add more compliance with OpenID Connect specification, we added flags into admin console to Client Settings page, where you can enable/disable various kinds of OpenID Connect/OAuth2 flows (Standard flow, Implicit flow, Direct Access Grants, Service Accounts). As part of this, we have `Direct Access Grants` (corresponds to OAuth2 `Resource Owner Password Credentials Grant`) disabled by default for new clients.

Clients migrated from previous version have `Direct Access Grants` enabled just if they had flag `Direct Grants Only` on. The `Direct Grants Only` flag was removed as if you enable Direct Access Grants and disable both Standard+Implicit flow, you will achieve same effect.

We also added builtin client `admin-cli` to each realm. This client has `Direct Access Grants` enabled. So if you're using Admin REST API or Keycloak admin-client, you should update your configuration to use `admin-cli` instead of `security-admin-console` as the latter one doesn't have direct access grants enabled anymore by default.

### Option 'Update Profile On First Login' moved from Identity provider to Review Profile authenticator

In this version, we added `First Broker Login`, which allows you to specify what exactly should be done when new user is logged through Identity provider (or Social provider), but there is no existing Keycloak user yet linked to the social account. As part of this work, we added option `First Login Flow` to identity providers where you can specify the flow and then you can configure this flow under `Authentication` tab in admin console.

We also removed the option `Update Profile On First Login` from the Identity provider settings and moved it to the configuration of `Review Profile` authenticator. So once you specify which flow should be used for your Identity provider (by default it's `First Broker Login` flow), you go to `Authentication` tab, select the flow and then you configure the option under `Review Profile` authenticator.

### Element 'form-error-page' in web.xml not supported anymore

form-error-page in web.xml will no longer work for client adapter authentication errors. You must define an error-page for the various HTTP error codes. See documentation for more details if you want to catch and handle adapter error conditions.

### IdentityProviderMapper changes

There is no change in the interface itself or method signatures. However there is some change in behaviour. We added `First Broker Login` flow in this release and the method `IdentityProviderMapper.importNewUser` is now called after `First Broker Login` flow is finished. So if you want to have any attribute available in `Review Profile` page, you would need to use the method `preprocessFederatedIdentity` instead of `importNewUser` . You can set any attribute by invoke `BrokeredIdentityContext.setUserAttribute` and that will be available on `Review profile` page.

## 35.6.4. Migrating to 1.6.0.Final

### Option that refresh tokens are not reusable anymore

Old versions of Keycloak allowed reusing refresh tokens multiple times. Keycloak still permits this, but also have an option `Revoke refresh token` to disallow it. Option is in in admin console under token settings. When a refresh token is used to obtain a new access token a new refresh token is also included. When option is enabled, then this new refresh token should be used next time the access token is refreshed. It won't be possible to reuse old refresh token multiple times.

### Some packages renamed

We did a bit of restructure and renamed some packages. It is mainly about renaming internal packages of util classes. The most important classes used in your application ( for example AccessToken or KeycloakSecurityContext ) as well as the SPI are still unchanged. However there is slight chance that you will be affected and will need to update imports of your classes. For example if you are using multitenancy and KeycloakConfigResolver, you will be affected as for example class HttpFacade was moved to different package - it is `org.keycloak.adapters.spi.HttpFacade` now.

### Persisting user sessions

We added support for offline tokens in this release, which means that we are persisting "offline" user sessions into database now. If you are not using offline tokens, nothing will be persisted for you, so you don't need to care about worse performance for more DB writes. However in all cases, you will need to update `standalone/configuration/keycloak-server.json` and add `userSessionPersister` like this:

```
"userSessionPersister": {
    "provider": "jpa"
},
```

If you want sessions to be persisted in Mongo instead of classic RDBMS, use provider `mongo` instead.

## 35.6.5. Migrating to 1.5.0.Final

### Realm and User cache providers

Infinispan is now the default and only realm and user cache providers. In non-clustered mode a local Infinispan cache is used. We've also removed our custom in-memory cache and the no cache providers. If you have realmCache or userCache set in keycloak-server.json to mem or none please remove these. As Infinispan is the only provider there's no longer any need for the realmCache and userCache objects so these can be removed.

### Uses Session providers

Infinispan is now the default and only user session provider. In non-clustered mode a local Infinispan cache is used. We've also removed the JPA and Mongo user session providers. If you have userSession set in keycloak-server.json to mem, jpa or mongo please remove it. As Infinispan is the only provider there's no longer any need for the userSession object so it can be removed.

For anyone that wants to achieve increased durability of user sessions this can be achieved by configuring the user session cache with more than one owner or use a replicated cache. It's also possible to configure Infinispan to persist caches, although that would have impacts on performance.

### Contact details removed from registration and account management

In the default theme we have now removed the contact details from the registration page and account management. The admin console now lists all the users attributes, not just contact specific attributes. The admin console also has the ability to add/remove attributes to a user. If you want to add contact details, please refer to the address theme included in the examples.

## 35.6.6. Migrating to 1.3.0.Final

### Direct Grant API always enabled

In the past Direct Grant API (or Resource Owner Password Credentials) was disabled by default and there was an option on a realm to enable it. The Direct Grant API is now always enabled and the option to enable/disable for a realm is removed.

### Database changed

There are again few database changes. Remember to backup your database prior to upgrading.

### UserFederationProvider changed

There are few minor changes in UserFederationProvider interface. You may need to sync your implementation when upgrade to newer version and upgrade few methods, which has changed signature. Changes are really minor, but were needed to improve performance of federation.

### WildFly 9.0.0.Final

Following on from the distribution changes that was done in the last release the standalone download of Keycloak is now based on WildFly 9.0.0.Final. This als affects the overlay which can only be deployed to WildFly 9.0.0.Final or JBoss EAP 6.4.0.GA. WildFly 8.2.0.Final is no longer supported for the server.

### WildFly, JBoss EAP and JBoss AS7 adapters

There are now 3 separate adapter downloads for WildFly, JBoss EAP and JBoss AS7:

- `eap6` - for JBoss EAP 6.x
- `wf9` - for WildFly 9.x
- `wf8` - for WildFly 8.x
- `as7` - for JBoss AS 7.x

Make sure you grab the correct one.

You also need to update standalone.xml as the extension module and subsystem definition has changed. See *Adapter Installation* for details.

## 35.6.7. Migrating from 1.2.0.Beta1 to 1.2.0.RC1

### Distribution changes

Keycloak is now available in 3 downloads: standalone, overlay and demo bundle. The standalone is intended for production and non-JEE developers. Overlay is aimed at adding Keycloak to an existing WildFly 8.2 or EAP 6.4 installation and is mainly for development. Finally we have a demo (or dev) bundle that is aimed at developers getting started with Keycloak. This bundle contains a WildFly server, with Keycloak server and adapter included. It also contains all documentation and examples.

### Database changed

This release contains again a number of changes to the database. The biggest one is Application and OAuth client merge. Remember to backup your database prior to upgrading.

### Application and OAuth client merge

Application and OAuth clients are now merged into `Clients`. The UI of admin console is updated and database as well. Your data from database should be automatically updated. The previously set Applications will be converted into Clients with `Consent required` switch off and OAuth Clients will be converted into Clients with this switch on.

## 35.6.8. Migrating from 1.1.0.Final to 1.2.0.Beta1

### Database changed

This release contains a number of changes to the database. Remember to backup your database prior to upgrading.

### `iss` in access and id tokens

The value of `iss` claim in access and id tokens have changed from `realm name` to `realm url`. This is required by OpenID Connect specification. If you're using our adapters there's no change required, other than if you've been using bearer-only without specifying `auth-server-url` you have to add it now. If you're using another library (or RSATokenVerifier) you need to make the corresponding changes when verifying `iss`.

### OpenID Connect endpoints

To comply with OpenID Connect specification the authentication and token endpoints have been changed to having a single authentication endpoint and a single token endpoint. As per-spec `response_type` and `grant_type` parameters are used to select the required flow. The old endpoints (`/realms/{realm}/protocols/openid-connect/login`, `/realms/{realm}/protocols/openid-connect/grants/access`, `/realms/{realm}/protocols/openid-connect/refresh`, `/realms/{realm}/protocols/openid-connect/access/codes`) are now deprecated and will be removed in a future version.

### Theme changes

The layout of themes have changed. The directory hierarchy used to be `type/name` this is now changed to `name/type`. For example a login theme named `sunrise` used to be deployed to `standalone/configuration/themes/login/sunrise`, which is now moved to `standalone/configuration/themes/sunrise/login`. This change was done to make it easier to have group the different types for the same theme into one folder.

If you deployed themes as a JAR in the past you had to create a custom theme loader which required Java code. This has been simplified to only requiring a plain text file (`META-INF/keycloak-themes.json`) to describe the themes included in a JAR. See the *themes* section in the docs for more information.

### Claims changes

Previously there was `Claims` tab in admin console for application and OAuth clients. This was used to configure which attributes should go into access token for particular application/client. This was removed and replaced with *Protocol mappers*, which are more flexible.

You don't need to care about migration of database from previous version. We did migration scripts for both RDBMS and Mongo, which should ensure that claims configured for particular application/client will be converted into corresponding protocol mappers (Still it's safer to backup

DB before migrating to newer version though). Same applies for exported JSON representation from previous version.

### Social migration to identity brokering

We refactored social providers SPI and replaced it with *identity brokering SPI*, which is more flexible. The `Social` tab in admin console is renamed to `Identity Provider` tab.

Again you don't need to care about migration of database from previous version similarly like for Claims/protocol mappers. Both configuration of social providers and "social links" to your users will be converted to corresponding Identity providers.

Only required action from you would be to change allowed `Redirect URI` in the admin console of particular 3rd party social providers. You can first go to the Keycloak admin console and copy Redirect URI from the page where you configure the identity provider. Then you can simply paste this as allowed Redirect URI to the admin console of 3rd party provider (IE. Facebook admin console).

## 35.6.9. Migrating from 1.1.0.Beta2 to 1.1.0.Final

- Providers are no longer loaded from `WEB-INF/lib`, they are now loaded from `standalone/configuration/providers`. See the *providers* section for more details.

## 35.6.10. Migrating from 1.1.0.Beta1 to 1.1.0.Beta2

- Adapters are now a separate download. They are not included in appliance and war distribution. We have too many now and the distro is getting bloated.
- The tomcat adapter valve has moved to a different package. From `org.keycloak.adapters.tomcat7.KeycloakAuthenticatorValve` to `org.keycloak.adapters.tomcat.KeycloakAuthenticatorValve` From the 'tomcat7' package to just 'tomcat'.
- JavaScript adapter now has idToken and idTokenParsed properties. If you use idToken to retrieve first name, email, etc. you need to change this to idTokenParsed.
- The as7-eap-subsystem and keycloak-wildfly-subsystem have been merged into one keycloak-subsystem. If you have an existing standalone.xml or domain.xml, you will need edit near the top of the file and change the extension module name to org.keycloak.keycloak-subsystem. For AS7 only, the extension module name is org.keycloak.keycloak-as7-subsystem.
- Server installation is no longer supported on AS7. You can still use AS7 as an application client.

## 35.6.11. Migrating from 1.0.x.Final to 1.1.0.Beta1

- RealmModel JPA and Mongo storage schema has changed

- UserSessionModel JPA and Mongo storage schema has changed as these interfaces have been refactored
- Upgrade your adapters, old adapters are not compatible with Keycloak 1.1. We interpreted JSON Web Token and OIDC ID Token specification incorrectly. 'aud' claim must be the client id, we were storing the realm name in there and validating it.

### 35.6.12. Migrating from 1.0 RC-1 to RC-2

- A lot of info level logging has been changed to debug. Also, a realm no longer has the jboss-logging audit listener by default. If you want log output when users login, logout, change passwords, etc. enable the jboss-logging audit listener through the admin console.

### 35.6.13. Migrating from 1.0 Beta 4 to RC-1

- logout REST API has been refactored. The GET request on the logout URI does not take a session_state parameter anymore. You must be logged in in order to log out the session. You can also POST to the logout REST URI. This action requires a valid refresh token to perform the logout. The signature is the same as refresh token minus the grant type form parameter. See documentation for details.

### 35.6.14. Migrating from 1.0 Beta 1 to Beta 4

- LDAP/AD configuration is changed. It is no longer under the "Settings" page. It is now under Users->Federation. Add Provider will show you an "ldap" option.
- Authentication SPI has been removed and rewritten. The new SPI is UserFederationProvider and is more flexible.
- `ssl-not-required` property in adapter config has been removed. Replaced with `ssl-required`, valid values are `all` (require SSL for all requests), `external` (require SSL only for external request) and `none` (SSL not required).
- DB Schema has changed again.
- Created applications now have a full scope by default. This means that you don't have to configure the scope of an application if you don't want to.
- Format of JSON file for importing realm data was changed. Now role mappings is available under the JSON record of particular user.

### 35.6.15. Migrating from 1.0 Alpha 4 to Beta 1

- DB Schema has changed. We have added export of the database to Beta 1, but not the ability to import the database from older versions. This will be supported in future releases.

- For all clients except bearer-only applications, you must specify at least one redirect uri. Keycloak will not allow you to log in unless you have specified a valid redirect uri for that application.
- Resource Owner Password Credentials flow is now disabled by default. It can be enabled by setting the toggle for `Direct Grant API ON` under realm config in the admin console.
- Configuration is now done through `standalone/configuration/keycloak-server.json`. This should mainly affect those that use MongoDB.
- JavaScript adapter has been refactored. See the *JavaScript adapter* section for more details.
- The "Central Login Lifespan" setting no longer exists. Please see the *Session Timeout* section for me details.

### 35.6.16. Migrating from 1.0 Alpha 2 to Alpha 3

- SkeletonKeyToken, SkeletonKeyScope, SkeletonKeyPrincipal, and SkeletonKeySession have been renamed to: AccessToken, AccessScope, KeycloakPrincipal, and KeycloakAuthenticatedSession respectively.
- ServleOAuthClient.getBearerToken() method signature has changed. It now returns an AccessTokenResponse so that you can obtain a refresh token too.
- Adapters now check the access token expiration with every request. If the token is expired, they will attempt to invoke a refresh on the auth server using a saved refresh token.
- Subject in AccessToken has been changed to the User ID.

### 35.6.17. Migrating from 1.0 Alpha 1 to Alpha 2

- DB Schema has changed. We don't have any data migration utilities yet as of Alpha 2.
- JBoss and Wildfly adapters are now installed via a JBoss/Wildfly subsystem. Please review the adapter installation documentation. Edits to standalone.xml are now required.
- There is a new credential type "secret". Unlike other credential types, it is stored in plain text in the database and can be viewed in the admin console.
- There is no longer required Application or OAuth Client credentials. These client types are now hard coded to use the "secret" credential type.
- Because of the "secret" credential change to Application and OAuth Client, you'll have to update your keycloak.json configuration files and regenarate a secret within the Application or OAuth Client credentials tab in the administration console.