Keycloak
Reference Guide

SSO for Web Apps and REST Services

1.0.1.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.
- OAuth Bearer token auth for REST Services
- Integrated Browser App to REST Service token propagation
- OAuth Bearer token auth for REST Services
- OAuth 2.0 Grant requests
- OpenID Connect 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, applications, user sessions, allowed CORS web origins, and OAuth clients.
- 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.
- Multitenancy support. You can host and manage multiple realms for multiple organizations.
- Supports JBoss AS7, EAP 6.x, Wildfly and 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, applications, and registered oauth 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.

An *application* is a service that is secured by a realm. 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 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 application level and assigned to specific users. Depending on the application type, you may also be able to view and manage user sessions from the administration console.

An *oauth client* is similar to an application in that it can request something like a login when a user visits the site of the oauth client. The difference is that oauth clients are 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 oauth client.

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 application and oauth client are configured with a set of permission scopes. These are a set of roles that an application or oauth client is allowed to ask permission for. Access tokens are always granted at the request of a specific application or oauth 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. The role mappings contained within the token are the intersection between the set of user role mappings and the permission scope of the application/oauth client. So, access tokens are tailor made for each application/oauth client and contain only the information required for by them.

Chapter 3. Installation and Configuration of Keycloak Server

The Keycloak Server has two downloadable distributions.

- keycloak-appliance-dist-all-1.0.1.Final.zip
- keycloak-war-dist-all-1.0.1.Final.zip

3.1. Appliance Install

The `keycloak-appliance-dist-all-1.0.1.Final.zip` is quite large, but contains a complete server (backed by Wildfly) that runs out of the box. The only thing you'll have to enable and configure is SSL. Unzipping it, the directory layout looks something like this:

```
keycloak-appliance-dist-all-1.0.1.Final/
    keycloak/
        bin/
            standalone.sh
            standalone.bat
            standalone/deployments/
                auth-server.war/
            standalone/configuration/
                keycloak-server.json
                themes/
    adapters/
        keycloak-as7-adapter-dist-1.0.1.Final.zip
        keycloak-eap6-adapter-dist-1.0.1.Final.zip
        keycloak-wildfly-adapter-dist-1.0.1.Final.zip
    examples/
    docs/
```

The `standalone.sh` or `standalone.bat` script is used to start the server. After executing that, log into the admin console at *http://localhost:8080/auth/admin/index.html* [http://localhost:8080/auth/admin/index.html]. Username: *admin* Password: *admin*. Keycloak will then prompt you to enter in a new password.

3.2. WAR Distribution Installation

The `keycloak-war-dist-all-1.0.1.Final.zip` contains just the bits you need to install keycloak on your favorite web container. We currently only support installing it on top of an existing JBoss AS 7.1.1, JBoss EAP 6.x, or Wildfly 8 distribution. We may in the future provide directions

on how to install it on another web container like Tomcat or Jetty. If anybody in the community is interested in pulling this together, please contact us. Its mostly Maven pom work.

The directory structure of this distro looks like this:

```
keycloak-war-dist-all-1.0.1.Final/
    deployments/
        auth-server.war/
        keycloak-ds.xml
    configuration/
        keycloak-server.json
        themes/
    adapters/
        keycloak-as7-adapter-dist-1.0.1.Final.zip
        keycloak-eap6-adapter-dist-1.0.1.Final.zip
        keycloak-wildfly-adapter-dist-1.0.1.Final.zip
    examples/
    docs/
```

After unzipping this file, copy everything in `deployments` directory into the `standalone/deployments` of your JBoss or Wildfly distro. Also, copy everything in `configuration` directory into the `standalone/configuration` directory.

```
    $ cd keycloak-war-dist-all-1.0.1.Final
    $ cp -r deployments $JBOSS_HOME/standalone
    $ cp -r configuration $JBOSS_HOME/standalone
```

After these steps you should also *install the client adapter* as this may contain modules the server needs (like Bouncycastle). You will also need to install the adapter to run the examples on the same server.

After booting up the JBoss or Wildfly distro, you can then make sure it is installed properly by logging into the admin console at *http://localhost:8080/auth/admin/index.html* [http://localhost:8080/auth/admin/index.html]. Username: *admin* Password: *admin*. Keycloak will then prompt you to enter in a new password.

3.3. 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

3.3.1. Relational Database Configuration

By default, Keycloak uses a relational database to store Keycloak data. This datasource is the `standalone/deployments/keycloak-ds.xml` file of your Keycloak Server installation if you used *Section 3.2, "WAR Distribution Installation"* or in `standalone/configuration/standalone.xml` if you used *Section 3.1, "Appliance Install"*. File `keycloak-ds.xml` is used in WAR distribution, so that you have datasource available out of the box and you don't need to edit `standalone.xml` file. However a good thing is to always delete the file `keycloak-ds.xml` and move its configuration text into the centrally managed `standalone.xml` file. This will allow you to manage the database connection pool from the Wildfly/JBoss administration console. Here's what `standalone/configuration/standalone.xml` should look like after you've done this:

```
<subsystem xmlns="urn:jboss:domain:datasources:2.0">
   <datasources>
      <datasource jndi-name="java:jboss/datasources/ExampleDS"
                  pool-name="ExampleDS" enabled="true" use-java-context="true">
                                                                    <connection-
url>jdbc:h2:mem:test;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE</connection-url>
         <driver>h2</driver>
         <security>
            <user-name>sa</user-name>
            <password>sa</password>
         </security>
      </datasource>
      <datasource jndi-name="java:jboss/datasources/KeycloakDS"
                  pool-name="KeycloakDS" enabled="true" use-java-context="true">
                              <connection-url>jdbc:h2:${jboss.server.data.dir}/
keycloak;AUTO_SERVER=TRUE</connection-url>
         <driver>h2</driver>
         <security>
            <user-name>sa</user-name>
            <password>sa</password>
        </security>
      </datasource>
      <drivers>
         <driver name="h2" module="com.h2database.h2">
         <xa-datasource-class>org.h2.jdbcx.JdbcDataSource</xa-datasource-class>
         </driver>
      </drivers>
   </datasources>
</subsystem>
```

Besides moving the database config into the central `standalone.xml` configuration file 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], *JBoss AS7* [https://docs.jboss.org/author/display/AS71/DataSource+configuration], or *JBoss EAP 6.x* [https://docs.jboss.org/author/display/AS71/DataSource+configuration] documentation on how to do this.

Keycloak also 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
: Value of database schema (Hibernate property "hibernate.hbm2ddl.auto").

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)

unitName
: Allow you to specify name of persistence unit if you want to provide your own persistence.xml file for JPA configuration. If this option is used, then all other configuration options are ignored as you are expected to configure all JPA/DB properties in your own persistence.xml file. Hence you can remove properties "dataSource" and "databaseSchema" in this case.

For more info about Hibernate properties, see *Hibernate and JPA documentation* [http://hibernate.org/orm/documentation/] .

3.3.1.1. 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
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
Sybase ASE 15.7	JDBC(TM)/7.07 ESD #5 (Build 26792)/P/EBF20686	auto

3.3.2. 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"
    }
}
```

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.

3.3.3. AS7/EAP6.x Logging

Accessing the admin console will get these annoying log messages:

```
    WARN  [org.jboss.resteasy.core.ResourceLocator] (http-/127.0.0.1:8080-3)
```

```
                    Field providers of subresource xxx will not be injected
according to spec
```

These can be ignored by editing standalone.xml of your jboss installation:

```
<logger category="org.jboss.resteasy.core.ResourceLocator">
    <level name="ERROR"/>
</logger>
```

3.3.4. SSL/HTTPS Requirement/Modes

Warning

Keycloak is not set up by default to handle SSL/HTTPS in either the war distribution or appliance. 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.

3.3.5. 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.

3.3.5.1. 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 JBoss or Wildfly to use this certificate and turn on SSL/HTTPS.

3.3.5.1.1. 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.

3.3.5.1.1.1. 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 the `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
```

3.3.5.1.2. 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.

3.3.5.1.3. Installing the keystore to JBoss EAP6/AS7

Now that you have a Java keystore with the appropriate certificates, you need to configure your JBoss EAP6/AS7 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.

```
<subsystem xmlns="urn:jboss:domain:web:1.1" default-virtual-server="default-
host" native="false">
  <connector name="http" protocol="HTTP/1.1" scheme="http" socket-binding="http"
  redirect-port="443" />
       <connector name="https" scheme="https" protocol="HTTP/1.1" socket-
binding="https"
```

```
              enable-lookups="false" secure="true">
       <ssl name="localhost-ssl" password="secret" protocol="TLSv1"
        key-alias="localhost" certificate-key-file="${jboss.server.config.dir}/
keycloak.jks" />
   </connector>
        ...
</subsystem>
```

Check the *JBoss* [https://docs.jboss.org/author/display/AS71/SSL+setup+guide] documentation for more information on fine tuning the socket connections.

3.3.5.2. 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. This is done by editing `standalone/configuration/standalone.xml`.
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.

3.3.5.3. Enforce HTTPS For Server Connections

Servlet containers can force browsers and other HTTP clients to use HTTPS. You have to configure this in `.../standalone/deployments/auth-server.war/WEB-INF/web.xml`. All you have to do is uncomment out the security constraint.

```
<web-app>
 ...
       <security-constraint>
           <web-resource-collection>
               <url-pattern>/*</url-pattern>
           </web-resource-collection>
           <user-data-constraint>
               <transport-guarantee>CONFIDENTIAL</transport-guarantee>
           </user-data-constraint>
       </security-constraint>
</web-app>
```

Chapter 4. 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.

4.1. 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.

4.2. 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.

4.3. 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 5. 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.

5.1. 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.

5.2. 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
- `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 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 6. Per Realm Admin Access Control

Administering your realm through the `master` realm as discussed in *Chapter 5, 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 privleges 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`

6.1. 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
- `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 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 7. 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.

7.1. 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,
  "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 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*.

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 JSON list of strings. 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*.

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 if 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. This is the file path to a Java keystore file. 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`. The default value is*false*.

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.

7.2. 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.

7.2.1. Adapter Installation

This is a adapter zip file for AS7, EAP, and Wildfly in the `adapters/` directory in the Keycloak distribution.

Install on Wildfly:

```
$ cd $WILDFLY_HOME
$ unzip keycloak-wildfly-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`.

For Wildfly:

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

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

    <profile>
        <subsystem xmlns="urn:jboss:domain:keycloak:1.0"/>
         ...
    </profile>
```

For JBoss AS 7.1.1 and EAP 6.x:

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

    <extensions>
        <extension module="org.keycloak.keycloak-as7-subsystem"/>
         ...
    </extensions>

    <profile>
        <subsystem xmlns="urn:jboss:domain:keycloak:1.0"/>
          ...
    </profile>
```

Finally, for both AS7, EAP 6.x, and Wildfly installations you must specify a shared keycloak security domain. This 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.

7.2.2. 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>
    </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>
    </security-constraint>

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

7.2.3. 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 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.

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

  <profile>
    <subsystem xmlns="urn:jboss:domain:keycloak:1.0">
       <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 `security-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:1.0">
    <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>
```

7.3. Pure Client Javascript Adapter

The Keycloak Server comes with a Javascript library you can use to secure pure HTML/Javascript applications. 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 end up having a non-confidential, public client. This can be mitigated by registering valid redirect URLs. You are still vulnerable if somebody hijacks the IP/DNS name of your pure HTML/Javascript application though.

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 as

well. Once you've configured the application click on the `Installation` tab and download the `keycloak.json` file. This file should be hosted in your web-server at the same root as your HTML pages. Alternatively you can either specify the URL for this file, or manually configure the adapter.

Next you have to initialize the adapter in your application. An example on how to do this 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 redirect to the login form on the server, while check-sso will 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="loadData()">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.onValidAccessToken()` passing in the `loadData()` and `loadFailure()` callbacks. The `keycloak.onValidAcessToken()` 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 if it's 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');
);
```

7.3.1. Session status iframe

By default the JavaScript adapter creates a non-visible iframe that is used to detect if a single-sign out has occured. 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.

7.3.2. JavaScript Adapter reference

7.3.2.1. Constructor

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

7.3.2.2. 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
- 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

7.3.2.3. 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
- checkLoginIframe - set to enable/disable monitoring login state (default is true)
- checkLoginIframeInterval - set the interval to check login state (default is 5 seconds)

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

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

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');
    });
```

7.3.2.4. 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)

7.4. Installed Applications

Keycloak provides two special redirect uris for installed applications.

7.4.1. 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`.

7.4.2. 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.

7.5. Logout

There are multiple ways you can logout from a web application. For Java EE servlet containers, you can call HttpServletRequest.logout(). For any other browser application, you can point the browser at 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.

Chapter 8. Social

Keycloak makes it easy to let users log in to your application using an existing account with a social network. Currently Facebook, Google and Twitter is supported with more planned for the future. There's also a Social Provider SPI that makes it relatively simple to add additional social networks.

8.1. Social Login Config

To enable log in with a social network you need to enable social login for your realm and configure one or more social providers.

8.1.1. Enable social login

To configure social login, open the `Keycloak Admin Console`, select your realm from the drop-down box in the top left corner. In the `Login Options` section click on `Social login` to set it to `ON`. Click save settings, then click on `Social` in the menu at the top.

To enable a social provider select the provider you want from the drop-down and click on `Add Provider`. Then continue to the section below that provides specific instructions for the provider you are adding.

8.1.2. Social-only login

It's possible to configure a realm to only allow social login. To do this open the `Keycloak Admin Console`, select your realm from the drop-down box in the top left corner. Click the `Credentials` tab, and click on the `x` next to `password` in the `Required User Credentials`. This will disable login with username and password.

8.1.3. Social Callback URL

There is a single callback url used by all realms and social providers. This makes it possible to share the configuration for a social network between multiple realms. An example callback url is `http://localhost:8080/auth/rest/social/callback`. To get the callback url for your server replace `http://localhost:8080` with the base address of your server. You can also find the callback url in the Keycloak Admin Console under social settings.

8.2. Facebook

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

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. In `Valid OAuth redirect URIs` insert the *social callback url*. 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/] into the settings page in the Keycloak Admin Console as the `Key` and `Secret`. Then click `Save` in the Keycloak Admin Console to enable login with Facebook.

8.3. 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.

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. In `Authorization callback URL` enter the *social callback url* for your realm. Click the `Register application` button.

2. Copy `Client ID` and `Client secret` from the *GitHub Settings* [https://github.com/settings/applications] into the settings page in the Keycloak Admin Console as the `Key` and `Secret`. Then click `Save` in the Keycloak Admin Console to enable login with Google.

8.4. 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.

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. In `Authorized redirect URI` enter the *social callback url* for your realm. Click the `Create Client ID` button.

5. Copy `Client ID` and `Client secret` from the *Google Developer Console* [https://cloud.google.com/console/project] into the settings page in the Keycloak Admin Console as the `Key` and `Secret`. Then click `Save` in the Keycloak Admin Console to enable login with Google.

8.5. 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.

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] into the settings page in the Keycloak Admin Console as the `Key` and `Secret`. Then click `Save` in the Keycloak Admin Console to enable login with Twitter.

Tip

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

8.6. Social Provider SPI

Keycloak provides an SPI to make it easy to add additional social providers. This is done by implementing `org.keycloak.social.SocialProvider` in `social/core` and adding a provider configuration file (`META-INF/services/org.keycloak.social.SocialProvider`).

A good reference for implementing a Social Provider is the Google provider which you can find in `social/google` on GitHub or in the source download.

Chapter 9. Themes

Keycloak provides theme support for login forms and account management. This allows customizing the look and feel of end-user facing pages so they can be integrated with your brand and applications.

9.1. Configure theme

To configure the theme used by 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`.

9.2. Default themes

Keycloak comes bundled with default themes in `standalone/configuration/themes`. It is not recommended to edit these themes directly. Instead you should create a new theme to extend a default theme. A good reference is to copy the keycloak themes as these extend the base theme to add styling.

9.3. Creating a theme

There are several types of themes in Keycloak:

- Account - Account management
- Admin - Admin console
- Common - Shared resources for themes
- Email - Emails
- Login - Login forms

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. It should be possible to achieve the customization required by styling these templates.

To create a new theme, create a folder in `.../standalone/configuration/themes/<theme type>`. The name of the folder is the name of the theme. Then create a file `theme.properties` inside the theme folder. The contents of the file should be:

```
parent=base
```

You have now created your theme. Check that it works by configuring it for a realm. It should look the same as the base theme as you've not added anything to it yet. The next sections will describe how to modify the theme.

9.3.1. 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
```

9.3.2. 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
```

9.3.3. 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">
```

9.3.4. Messages

Text in the templates are loaded from message bundles. Currently internationalization isn't supported, but that will be added in a later release. 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
```

9.3.5. 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.

9.4. SPIs

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

9.4.1. Theme SPI

The Theme SPI allows creating different mechanisms to providing themes for the default FreeMarker based implementations of login forms and account management. To create a theme provider you will need to implement `org.keycloak.freemarker.ThemeProvider` and `org.keycloak.freemarker.Theme` in `forms/common-freemarker`.

Keycloak comes with two theme providers, one that loads themes from the classpath (used by default themes) and another that loads themes from a folder (used by custom themes). Looking at these would be a good place to start to create your own theme provider. You can find them inside `forms/common-themes` on GitHub or the source download.

9.4.2. 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` in `forms/account-api`.

Keycloaks default account management provider is built on the FreeMarker template engine (`forms/account-freemarker`). To make sure your provider is loaded you will either need to delete `standalone/deployments/auth-server.war/WEB-INF/lib/keycloak-account-freemarker-1.0.1.Final.jar` or disable it with the system property `org.keycloak.account.freemarker.FreeMarkerAccountProviderFactory`.

9.4.3. 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`.

Keycloaks default login forms provider is built on the FreeMarker template engine (`forms/login-freemarker`). To make sure your provider is loaded you will either need to delete `standalone/deployments/auth-server.war/WEB-INF/lib/keycloak-login-freemarker-1.0.1.Final.jar` or disable it with the system property `org.keycloak.login.freemarker.FreeMarkerLoginFormsProviderFactory`.

Chapter 10. 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.

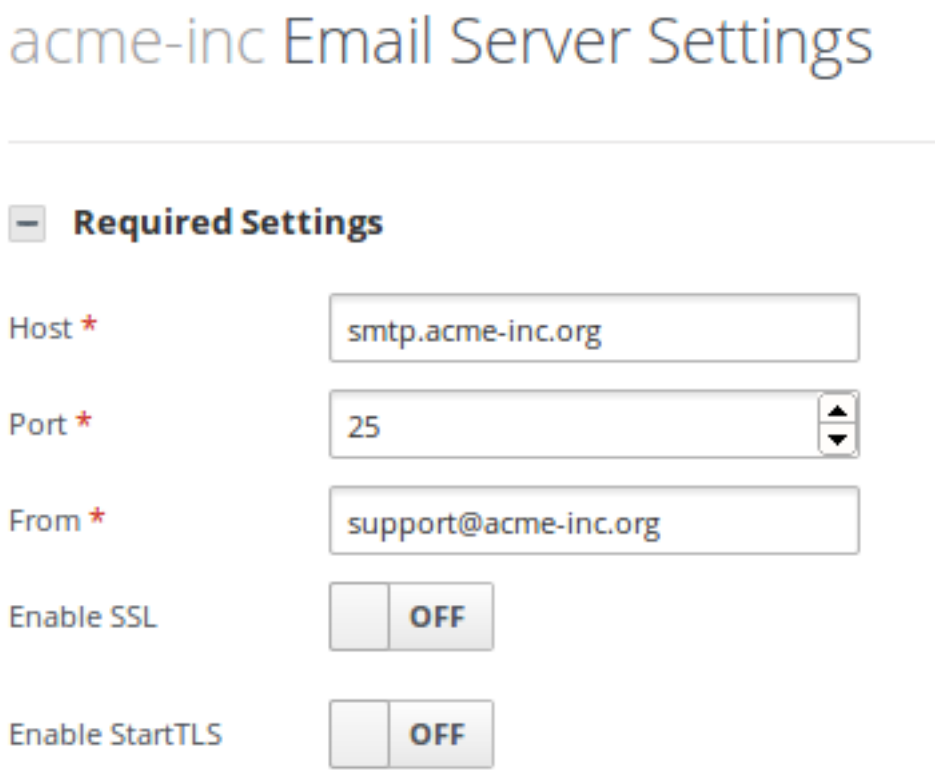
10.1. Email Server Config

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.



10.1.1. 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).

10.1.2. Authentication

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

Chapter 11. Application and Client Access Types

When you create an Application or OAuth Client 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
: For OAuth clients, you would also see a "Direct Access Only" switch when creating the OAuth Client. This switch is for oauth clients that only use the *Direct Access Grant* protocol to obtain access tokens.

Chapter 12. Roles

In Keycloak, roles (or permissions) 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.

12.1. 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 Application or OAuth 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 13. 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, Direct Access Grants must be allowed by your realm. This is a configuration switch in the admin console under Settings->General, specifically the "Direct Grant API" switch. You must also have registered a valid OAuth Client or Application 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*.

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}/tokens/grants/access`. 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 application or oauth client you are creating an access token for. This "client_id" is the application or oauth client name (not it's id!). Depending on whether your application/oauth client is *"public" or "confidential"*, you may also have to pass along it's client secret as well.

For public applications or oauth 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/tokens/grants/access
Content-Type: application/x-www-form-urlencoded

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

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 applications or oauth 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/tokens/grants/access
    Authorization: Basic atasdf023l2312023
    Content-Type: application/x-www-form-urlencoded

    username=bburke&password=geheim
```

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_SERVICE_DIRECT_GRANT_PATH).build("demo"));
   List <NameValuePair> formparams = new ArrayList <NameValuePair>();
   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 14. 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 application and oauth client. You specify the allowed origins in the application's or oauth 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 application or OAuth 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.

Chapter 15. 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.

15.1. 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.

15.2. 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.

15.3. 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 `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.

Chapter 16. 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 describe in *Chapter 5, Master Admin Access Control* and *Chapter 6, 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.

Chapter 17. 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.

17.1. 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.

17.2. Event Listener

Keycloak comes with an Email Event Listener and a JBogg 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:

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

17.3. 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" ]
    }
}
```

17.4. 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 18. 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. Thegoal of the Federation SPI is to support all of these scenarios.

18.1. LDAP and Active Directory Plugin

Keycloak comes with a built-in LDAP/AD plugin. Currently it is set up only to import username, email, first and last name. 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.

18.1.1. 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 will be unchangable. 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, 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.

18.1.2. Other config options

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

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.

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.

18.2. 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.

Thing is that Federation Provider import 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.

18.3. 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 explain 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 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`. This is how Keycloak discovers which providers have been deployment. Place the JAR in the keycloak WAR deployment in the `WEB-INF/lib` directory.

Chapter 19. Export and 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:

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

Encrypted ZIP is recommended as export contains many sensitive informations like passwords of your users (even if they are hashed), but also their email addresses, and especially private keys of the realms. Directory and Single JSON file are useful especially for testing as data in the files are not protected. On the other hand, it's useful if you want to look at all your data in JSON files directly.

If you import to ZIP or 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.

So to export the content of your Keycloak database into encrypted ZIP, you can execute Keycloak server with the System properties like:

```
bin/standalone.sh -Dkeycloak.migration.action=export
-Dkeycloak.migration.provider=zip -Dkeycloak.migration.zipFile=<FILE TO EXPORT
 TO>
-Dkeycloak.migration.zipPassword=<PASSWORD TO DECRYPT EXPORT>
```

Then you can move or copy the encrypted ZIP file into second environment and you can trigger import from it into Keycloak server with the same command but use `-Dkeycloak.migration.action=import` instead of `export` .

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 ZIP or 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.

Chapter 20. 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.

20.1. Disabling Caches

The realm and user caches can be disabled through configuration or through the management console. To manally 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.

```
"realmCache": {
    "provider": "${keycloak.realm.cache.provider:mem}"
},

"userCache": {
    "provider": "${keycloak.user.cache.provider:mem}",
    "mem": {
        "maxSize": 20000
    }
},
```

You must then change it to:

```
"realmCache": {
    "provider": "${keycloak.realm.cache.provider:none}"
},

"userCache": {
    "provider": "${keycloak.user.cache.provider:none}"
},
```

You can also disable either of the caches at runtime through the Keycloak admin console Realm Settings page. This will not permanently disable the cache. If you reboot the server, the cache will be re-enabled unless you manualy disable the cache in the `keycloak-server.json` file.

20.2. 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. Save the settings. Then re-enable the cache. This will cause the cache to be cleared.

20.3. Cache Config

Cache configuration is done within `keycloak-server.json`. Changes to this file will not be seen by the server until you reboot. Currently you can only configure the max size of the user cache.

```
"userCache": {
    "provider": "${keycloak.user.cache.provider:mem}",
    "mem": {
        "maxSize": 20000
    }
},
```

Chapter 21. Migration from older versions

21.1. 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.

21.2. 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.

21.3. 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.

21.4. 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.

21.5. 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.

21.6. 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.