How to secure your Spring Apps with Keycloak

Thomas Darimont
@thomasdarimont
Thomas Darimont

- Fellow @codecentric
- Spring Data Team Alumni
- Open Source Enthusiast
- Organizer Java User Group Saarland
- Keycloak Contributor for over 3 years

@thomasdarimont
@jugsaar
The Journey

Keycloak

Single Sign-on

Securing Applications

Keycloak in the field
Keycloak

Overview
Add authentication to applications and secure services with minimum fuss. No need to deal with storing users or authenticating users. It’s all available out of the box.

You'll even get advanced features such as User Federation, Identity Brokering and Social Login.

For more details go to about and documentation, and don't forget to try Keycloak. It's easy by design!
• Started in 2013, broad adoption since 2015
• Apache License, Red Hat Developers
• Latest stable version 5.0.0.Final ~ every 6 Weeks
• Commercial Offering Available → Red Hat SSO
• Vital Community with 300+ Contributors 1.700+ Forks
• Very robust, good documentation, many examples
Features

• **Single Sign-on** and Single Sign-out
• **Standard Protocols** OAuth 2.0, OIDC 1.0, SAML 2.0, Docker Auth
• Flexible **Authentication** and **Authorization**
• **Multi-Factor Authentication** One-time Password
• **Social Login** Google, Facebook, Twitter,...
• Provides centralized **User Management**
• Supports **Directory Services**
• **Customizable** and **Extensible**
• **Easy** Setup and Integration
Main Concepts
Keycloak

Quick Tour
Admin Console Login
Admin Console

**Acme**

- **Name**: acme
- **Display name**: Acme Inc.
- **HTML Display name**: <strong>Acme Inc.</strong>
- **Enabled**: ON
- **Endpoints**: OpenID Endpoint Configuration

**Realm Settings**
- Clients
- Client Templates
- Roles
- Identity Providers
- User Federation
- Authentication

**Manage**
- Groups
- Users
- Sessions
- Events
- Import
- Export
Technology Stack 5.0.0.RELEASE

Admin Console
- Angular JS (1.6.x)
- PatternFly
- Bootstrap

Keycloak Server
- Wildfly 15.0.x
- JAX-RS (Resteasy)
- JPA (Hibernate)
- Infinispan (JGroups)
- Freemarker
- Jackson 2.x
- JBoss Logging
- Apache Directory API
- Commons HTTP Client
Single Sign-on with Keycloak

How it works
Single Sign-on & Single Logout

• **SSO ⇒ Login once** to access all applications
• **Standardized Protocols**
  • OpenID Connect 1.0 (OIDC)
  • Security Assertion Markup Language 2.0 (SAML)
• **Browser based “Web SSO”**
• works for Web, Mobile and Desktop Apps
• **Support for Single Logout**
  • Logouts can be propagated to clients
  • Clients can opt-in
Web SSO with OIDC*: Unauthenticated User

1. Unauthenticated User accesses App
2. App redirects to Keycloak for Login
2a. User submits Credentials to Keycloak
2b. Keycloak validates Credentials
3. Keycloak creates SSO Session Cookies and redirects User to App
4. App exchanges Code to Tokens with Keycloak via separate Channel
5. App verifies received Tokens and associates it with a session
5a. User is now logged-in to App

*) OAuth 2 Authorization Code Flow
Web SSO with OIDC: Authenticated User

1. Authenticated user **accesses** App 2
2. App 2 **redirects** user to Keycloak for login
3. Keycloak **detects** SSO Session, **generates** code, **redirects** to App 2
4. App 2 **exchanges** code for tokens with Keycloak via separate channel
5. App 2 **verifies** received tokens and associates it with a session
6. User is now **logged-in** to App 2
Keycloak Tokens

• OAuth 2 / OpenID Connect
  • Signed self-contained JSON Web Token
  • Claims: Metadata + User information
  • Issued by Keycloak, signed with Realm Private Key
  • Verified with Realm Public Key
  • Limited lifespan; can be revoked

• Essential Token Types
  • Access-Token short-lived (Minutes) → used for accessing Resources
  • Refresh-Token long-lived (Days) → used for requesting new Tokens
  • Offline-Token special Refresh-Token that “never” expires
  • IDToken contains information about User (OIDC)
JSON Web Tokens

Note
Base64 means **Encoding**

Encoding != Encryption

[https://jwt.io](https://jwt.io)
Keycloak JSON Web Token Example

Encoded

```
eyhbgci0iJSu1Ni1IsInR5cCIgOi1iSldUIwia2
1ki1A6IICMT0xq1ONFRwMFJcrj1SHmYpxsXJKr
v1lZQr3eF9f40tc0kI93In8.eyJqQgk10i1jIM
GyMGrJy8wNmRkLTRiMzgtYTUnOS8Dh10Dq2Njh
Yji1LC1ejHA0jE0TAT2NMDIsO5s51z15MCwiaw
FWbIjoxNDkWniUzND8yLCjpc3Mi0iJodHRw0i8v3c
LvnrkgGFicy6sb2NhDo4CDkJ53UYV8eCy3YwMFsb
XmvamF2YxhbmQiLCJhdWNQOi0j2pGt0YxpzW5G1iw
1c3Vj1joiMj10Yjg3YWQtY2Rkie18NjY3LWF1ODUt
EZZDhmmZDNhNmJ1iwi6H1lj0jQmVhcmVvYiwiYXpw
IjoiaWRtLWNsawVwVduCImF1IdghfdGtZJ5ISMCwic
Vzc2l1vb19dGFI0ZS1iJ2jmZDQ3Mj0KLTw0YjIhQGM4
Ny1iMz1iLk4YTA3w2Nz2M2FkNCIsImFjcI15iEiLC
JjbsG1bnRfcsVzc2l1vb16iJ4Mznk5ZjgyLTKrMnM
NDAYy1y1mEWLT3DIZ3N8vjWizMcIsImF59G3ZW
Qt3bJp2Zulcy16W18s1nJ1WyX2FX2Yvz16gyEj
b2x1cyci6QjybWFyXV0aGyaphG1vbi5ibisimVZ
IiXXz5s1nCqI291cmNX2JfYvZv16eyjHch1Z3Ji
ZXRpbmc2cydvj15JS16eyjyb2x1cyci6Qjyb2y1
19L3JkZm1vLN1cemP2iyu10iscm9aZX1i0lsidX1i
cziJdfSwIYXswLWphdmF1ZStjXJrbGl1uWM10iscm
9z5Xi0lsidX1iJdfSwiYWnb3vUCiEiyjyb2x1
cy16WyjT1W5hZ2UtyWYnb3vUcisInZpbXctcHJvZm
1sZjDfSfXw1YXswLWLRlct2ob3A10iscm9aZX1i0lsid
DNJ1cJdF80sIm5hWUj10iUaGWVF1Ric3RLcIsInM
ByZW1cncj12F91c23yvbmtZS1i61c3RLcIsInMdpd
```

Decoded

```

HEAER: ALGORITHM & TOKEN TYPE

```

```
{
   "alg": "RS256",
   "typ": "JWT",
   "kid": "LQ0qt74T4p8R0r96Jeii1qV5unYVCK7z__gKnsi5LOm",
}
```

PAYLOAD: DATA

```
{
   "jti": "bb820ccc-06d4-4b38-a529-4db88667ab2",
   "exp": 1498865742,
   "nbf": 0,
   "iss": 1498865342,
   "sub": "http://sso.t3labs.local:8899/api/realms/javalemd",
   "aud": "idm-client",
   "typ": "Bearer",
   "azp": "idm-client",
   "auth_time": 0,
   "session_state": "6fd4723d-4b02-4c87-b3b0-98a877ff3ad4",
   "acr": "1",
   "client_session": "38799f82-0bdc-4d2e-aba0-67d27ecee333",
   "allowed_origins": [],
   "realm_access": {
      "roles": [
         "uma_authorization",
         "user"
      ],
      "resource_access": {
         "app-greeting-service": {"roles": ["user"]}
      }
   }
}
```
Calling Backend Services with Access-Token

1. Authenticated User accesses App
2. App uses Access-Token in HTTP Header to access backend
3. Backend looks-up Realm Public Key in cache with Kid from JWT
   - If not found, fetch Public Key with Kid from Keycloak
3a. Keycloak returns Realm Public Key
3b. Keycloak returns Realm Public Key
4. Backend verifies signature of Access-Token with Realm Public Key
5. Backend Service grants access and returns user data
Keycloak Client Integrations
Keycloak Integration Options

- OpenID Connect Adapters
  - Spring Security, Spring Boot, ServletFilter, Tomcat, Jetty, Undertow, Wildfly, JBoss EAP,…
  - NodeJS, JavaScript, Angular, AngularJS, Aurelia, CLI & Desktop Apps…

- SAML Adapters
  - ServletFilter, Tomcat, Jetty, Wildfly ...

- Reverse Proxies
  - Keycloak Gatekeeper, dedicated Proxy, written in Go, injects auth info into HTTP headers
  - Apache mod_auth_oidc for OpenID Connect
  - Apache mod_auth_mellon for SAML

- Others see [OIDC](http://example.com) and [SAML](http://example.com)
Demo Environment

Authorization: Bearer $ACCESS_TOKEN
Demo Securing Apps
thomasdarimont/keycloak-docker-demo

Let’s see Keycloak in action :)
Keycloak in the Field

How can a Keycloak environment look like?
Demo Environment

- Desktop App
  - JavaFX

- PlainJS App
  - JavaScript

- Frontend
  - Spring Boot

- Backend
  - Spring Boot

- SAML App
  - Spring Boot

- Reverse Proxy
  - Load Balancer / WAF
  - SSL Termination

- Graylog
  - GELF/JSON

- Keycloak
  - JDBC

- Active MQ
  - JMS
  - Message Broker
    - Provisioning
    - Messages

- Postgres
  - HTTPS
  - HTTP(S)

- Distributed Cache
  - JGroups / Infinispan

- Log Monitoring
  - Alerts
  - Dashboards
Keycloak with Graylog+ActiveMQ in Action
Summary

• Easy to get started
  • unzip & run, Keycloak Docker Images

• Provides many features out of the box
  • SSO, Social Login, Federation, User Management,…

• Builds on proven and robust standards
  • OAuth 2.0, OpenID Connect 1.0, SAML 2.0

• Very extensible and easy to integrate
  • Many extension points & customization options

• A pivotal part of modern Identity Management
THANKS!

Q & A

Thomas Darimont

@thomasdarimont
Links

- Keycloak Website
- Keycloak Docs
- Keycloak Blog
- Keycloak User Mailing List
- Keycloak Developer Mailing List
- OpenID Connect
- SAML
- JSON Web Tokens
- Awesome Keycloak
- Keycloak Dockerized Examples
- Keycloak Quickstarts Example Projects
Tips for working with Keycloak

- Learn to configure Wildfly → Booktip: Wildfly Cookbook
- Keep your Tokens small → HTTP Header limits!
  - Only put in the tokens what you really need → Full Scope Allowed = off
- Keycloak provides a Realm-scoped Admin Console
  - http://kc-host:8080/auth/admin/my-realm/console
  - Admin users need permissions for realm-management in my-realm
- Secure your Keycloak Installation!
  - Keycloak exposes some undocumented Endpoints by default on server AND client!
  - Inspect other Keycloak instances to learn what to hide
    - Google Search for Keycloak Endpoints
    - Shodan search for Keycloak
Keycloak Extensions

How to tech Keycloak new tricks?
Keycloak Extension Points

- Extensions via Service Provider Interfaces
- Custom Authentication Mechanisms
- Custom “Required Actions”
- Custom User Storage (JDBC, REST, etc.)
- Event Listener (Provisioning, JMS)
- Credential Hashing Mechanisms
- Custom REST Endpoints
- Custom Themes
- ... many more
BeerCloak: a comprehensive Keycloak extension example
Custom Dashboard Extension

Please vote :) https://issues.jboss.org/browse/KEYCLOAK-1840
Authentication & Authorization

- **Authentication (AuthN)**
  - Determines *who the user is*
  - Internal & Federated User Storage Kerberos, LDAP, Custom
  - Customizable

- **Authorization (AuthZ)**
  - Determines *what the user is allowed to do*
  - Hierarchical Role-based Access Control (HRBAC)
  - Authorization Services
    - Flexible Access Control Management
    - More Variants like ABAC, UBAC, CBAC supported
Supported Single Sign-on Protocols

- **OpenID Connect 1.0**
  - Protocol based on OAuth 2.0
  - Uses OAuth 2.0 tokens + `IDToken` to encode `Identity`
  - Tokens are encoded as JSON Web Tokens (JWT)
  - Requires secure channel HTTPS/TLS

- **SAML 2.0 Security Assertion Markup Language**
  - Very mature standard & common in enterprise environments
  - XML based protocol
  - Uses XML signature and encryption

- **Docker Registry v2 Authentication**
Accessing the API Backend with CURL

1. Request new Tokens via Password Credentials Grant (Direct Access Grants in Keycloak)

```bash
KC_RESPONSE=$(curl -X POST \
  http://sso.tdlabs.local:8899/u/auth/realms/acme/protocol/openid-connect/token \
  -d 'grant_type=password' \
  -d 'username=tester&password=test' \
  -d 'client_id=app-frontend-springboot&client_secret=4822a740-20b9-4ff7-bbed-e664f4a70eb6' \
)
```

2. Extract AccessToken

```bash
KC_ACCESS_TOKEN=$(echo $KC_RESPONSE | jq -r .access_token)
# eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJGY3RMVHJqeWRxYkpISGZ0d29U ...
```

3. Use AccessToken in Authorization Header

```bash
curl \
  -H "Authorization: Bearer $KC_ACCESS_TOKEN" \
  http://apps.tdlabs.local:20000/todos/search/my-todos
```
Desktop Applications

- Two ways to integrate Desktop Applications
  - Direct Access Grants - *no* SSO
  - KeycloakInstalled Adapter - SSO

- Direct Access Grants
  - Client sends HTTP POST request to Keycloaks /token Endpoint
  - `client_id`, `username`, `password`, `grant_type=password`
  - Keycloak returns Tokens (Access-, ID-, Refresh-Token)
  - Client needs to parse & validate tokens
  - Client sees password → *Password Anti-Pattern*

- KeycloakInstalled Adapter
  - Enables OAuth2 authorization code flow for Desktop / CLI apps
  - Code to Token exchange via short lived ServerSocket@localhost
  - Uses Keycloak Login via Browser
  - Can reuse existing SSO session
Using the KeycloakInstalled Adapter

1. **Add Maven Dependency**

   ```xml
   <dependency>
     <groupId>org.keycloak</groupId>
     <artifactId>keycloak-installed-adapter</artifactId>
     <version>${keycloak.version}</version>
   </dependency>
   ```

2. **Export keycloak.json for Client**

   ```json
   {
     "realm": "acme",
     "auth-server-url": "http://sso.tdlabs.local:8899/u/auth",
     "ssl-required": "external",
     "resource": "app-frontend-javafx",
     "public-client": true,
     "use-resource-role-mappings": true
   }
   ```

3. **Create KeycloakInstalled**

   KeycloakInstalled `keycloak = new KeycloakInstalled();`

4. **Trigger Browser login**

   `keycloak.loginDesktop();`

5. **Read current username**

   `keycloak.getIdToken().getPreferredUsername()`

6. **Read & use AccessToken string**

   String `token = keycloak.getTokenString(10, TimeUnit.SECONDS);`

   ```java
   httpClient.header("Authorization", "Bearer " + token);
   ```

7. **Trigger Browser Logout**

   `keycloak.logout();`
Keycloak OAuth / OpenID Connect Tokens

Token Types
- **Access**-Token
- **Refresh**-Token
- **Offline**-Token
- **ID**-Token (OIDC)

Claims
- **UserInfo + Metadata**
- Signed self-contained **JWT**
- **Limited Lifespan**
- May be **revoked**

Verification
- **Verify Signature** with **Realm Public Key**
- **Token Introspection Endpoint**