

Alicloud oauth with keycloak

Local keycloak instance:

<https://auth-dev.hmcn.tech/auth/admin/master/console/>

Configure client & user for oauth flow:

Create a realm:

The realm maps to a function domain in the organization. There can be many protected resources under the realm.

Create client:

Each restful webservice application should have its associated client configuration, which becomes a resource server.

Configure → clients → create:

The screenshot displays the Keycloak Admin Console interface. On the left is a dark sidebar with navigation options: Checkout, Configure (with sub-items: Realm Settings, Clients, Client Scopes, Roles, Identity Providers, User Federation, Authentication), and Manage (with sub-items: Groups, Users, Sessions, Events, Import, Export). The main content area is titled 'Clients > cpe'. At the top, there's a header for the client 'cpe' with a trash icon. Below this are tabs for Settings, Credentials, Keys, Roles, Client Scopes, Mappers, Scope, Revocation, and Sessions. The 'Settings' tab is active, showing various configuration fields: Client ID (cpe), Name, Description, Enabled (ON), Always Display in Console (OFF), Consent Required (OFF), Login Theme, Client Protocol (openid-connect), and Access Type (confidential). The 'Standard Flow Enabled' toggle at the bottom is also ON. Red boxes highlight the client name 'cpe' and the 'Client Protocol' and 'Access Type' dropdowns.

Checkout

Configure

- Realm Settings
- Clients
- Client Scopes
- Roles
- Identity Providers
- User Federation
- Authentication

Manage

- Groups
- Users
- Sessions
- Events
- Import
- Export

Clients > cpe

Cpe

Settings Credentials Keys Roles Client Scopes Mappers Scope Revocation Sessions

Clustering Installation

Client ID cpe

Name

Description

Enabled ON

Always Display in Console OFF

Consent Required OFF

Login Theme

Client Protocol openid-connect

Access Type confidential

Standard Flow Enabled ON

Direct Access Grants ☒ ON Enabled ?

Service Accounts ☒ ON Enabled ?

OAuth 2.0 Device Authorization Grant ☐ OFF Enabled ?

OIDC CIBA Grant ☐ OFF Enabled ?

Authorization Enabled ☒ ON

Root URL ?

* Valid Redirect URIs ? - +

Base URL ?

“Access Type” needs to be “confidential”, otherwise, there will be no client secret generated.

“Authorization Enabled” needs to be set to “on”

“Valid Redirect URIs” needs to match the root url of the restful webservice’s root url prefix

“Roles”: configure client roles, this is used to control access permission to the resource server . Note, each realm has two kind of roles: realm roles is general, whereas client role is specific to that client (ie. resource server). The relevant role for oauth flow is client role. This make sense as each client application may have different permission control requirement. So **don’t configure realm roles** as they will not take effect!

pe_dev

Settings Credentials Keys **Roles** Client Scopes ? Mappers ? Scope ? Authorization Revocation Sessions ?

Offline Access ? Clustering Installation ? Service Account Roles ?

Role Name	Composite	Description	Actions	
cpe-dataimport	False	user only able to import idoc	Edit	Delete
cpe-super	False	super user for cpe, able to import idoc and calculate cart	Edit	Delete
cpe-user	False	user only able to send cart calculation request	Edit	Delete
uma_protection	False		Edit	Delete

After saving, oauth configuration can be viewed and exported via Configurer → clients → installation

Checkout

Configure

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

Manage

- Groups
- Users
- Sessions
- Events

Clients > cpe

Cpe

Settings Credentials Keys Roles Client Scopes Mappers Scope Revocation Sessions

Clustering Installation

Format: Keycloak OIDC JSON

Option

Download

```
{
  "realm": "Checkout",
  "auth-server-url": "https://auth-dev.hm-btcloud.tech:9443/auth",
  "ssl-required": "external",
  "resource": "cpe",
  "credentials": {
    "secret": "e...j2"
  },
  "confidential-port": 0
}
```

Create user:

Each external organization that access the protected webservice resource should have its own keycloak user.

Manage → Users → create:

Configure

- Realm Settings
- Clients
- Client Scopes
- Roles
- Identity Providers
- User Federation
- Authentication

Manage

- Groups
- Users**
- Sessions
- Events
- Import
- Export

Cpe-data-importer-dev

Details Attributes Credentials Role Mappings Groups Consents Sessions

ID: 0fd56dbf-89c0-4730-8e19-f46b7b313673

Created At: 2/11/22 11:47:44 AM

Username: cpe-data-importer-dev

Email:

First Name:

Last Name:

User Enabled: ON

Email Verified: OFF

Required User Actions: Select an action...

Impersonate user: Impersonate

Only username is required here.

After creating the user, we need to assign a credential to it, we need to provide password to obtain oauth token in the oauth2 password flow.

Manage → Users → Credentials

Checkout

Configure

- Realm Settings
- Clients
- Client Scopes
- Roles
- Identity Providers
- User Federation
- Authentication

Manage

- Groups
- Users**
- Sessions
- Events
- Import
- Export

Cpe-data-importer-dev

Details Attributes **Credentials** Role Mappings Groups Consents Sessions

Manage Credentials

Position	Type	User Label	Data
<div>^</div> <div>v</div>	password		Show data...

Reset Password

Password

Password Confirmation

Temporary ☐ OFF

Reset Password

After creating the user, we also need to attach 1 or more client roles to it, so that the role information will be included in the generated oauth token.

Checkout

Users > cpe-data-importer-dev

Cpe-data-importer-dev

Details Attributes Credentials **Role Mappings** Groups Consents Sessions

Realm Roles

Available Roles

Add selected >

Assigned Roles

default-roles-checkout

<< Remove selected

Effective Roles

default-roles-checkout
offline_access
uma_authorization

Client Roles

cpe_dev

Available Roles

cpe-super
cpe-user
uma_protection

Add selected >

Assigned Roles

cpe-dataimport

<< Remove selected

Effective Roles

cpe-dataimport

Configure token expiration time:

Global level: Realm Settings → Tokens → Access Token Lifespan

Manage

- Groups
- Users
- Sessions
- Events
- Import
- Export

SSO Session Idle Remember Me ?

0Minutes

SSO Session Max Remember Me ?

0Minutes

Offline Session Idle ?

30Days

Offline Session Max Limited ?

OFF

Client Session Idle ?

1Days

Client Session Max ?

1Days

Access Token Lifespan ?

5Minutes

Access Token Lifespan For Implicit Flow ?

15Minutes

Client login timeout ?

1Minutes

Per client override: Clients → [client name] → Settings → Advanced settings → Access Token Lifespan

> OpenID Connect Compatibility Modes ?

Advanced Settings ?

Access Token Lifespan ?

1Days

Client Session Idle ?

Minutes

Client Session Max ?

Minutes

Client Offline Session Idle ?

Minutes

Client Offline Session Max ?

Minutes

OAuth 2.0 Mutual TLS Certificate Bound Access Tokens Enabled ?

OFF

Refresh token expiration time:

Global level: Realm Settings → Tokens → SSO Session Idle

Revoke Refresh Token ☐ OFF

SSO Session Idle Days

SSO Session Max Days

SSO Session Idle Remember Me Minutes

SSO Session Max Remember Me Minutes

Offline Session Idle Days

Offline Session Max Limited ☐ OFF

Client Session Idle Days

Client Session Max Days

Access Token Lifespan Minutes

Per client override: Clients → [client name] → Settings → Advanced settings → Client Session Idle (this will only take effect if the configured value is **less than Tokens → SSO Session Idle**). Otherwise, the configuration will not take effect and token refresh period falls back to Tokens → SSO Session Idle

Backchannel Logout ☐ OFF

Revoke Offline Sessions ☐ OFF

> Fine Grain OpenID Connect Configuration

> OpenID Connect Compatibility Modes

Advanced Settings

Access Token Lifespan Days

Client Session Idle Minutes

Client Session Max Minutes

Client Offline Session Minutes

Testing oauth token acquisition:

Request oauth token via password flow:



POST

https://auth-dev.hm-btcloud.tech:9443/auth/realms/checkout/protocol/openid-connect/token

Send

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

Cookies

☐ none
 ☐ form-data
 ☒ x-www-form-urlencoded
 ☐ raw
 ☐ binary
 ☐ GraphQL

	KEY	VALUE	DESCRIPTION	...	BULK EDIT
<input checked="" type="checkbox"/>	grant_type	password			
<input checked="" type="checkbox"/>	username	testuser1			
<input checked="" type="checkbox"/>	password				
<input checked="" type="checkbox"/>	client_id	cpe			
<input checked="" type="checkbox"/>	client_secret				

Body Cookies Headers (14) Test Results



200 OK

1926 ms

2.84 KB

Save Response

Pretty

Raw

Preview

Visualize

JSON



```

1 {
2   "access_token":
   "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXLTJ1Iiwia2lkIiA6ICJRSjU1cS0xcXhuOE0zcy1pZnRLX0M5aHk5YWt2b3U5S2ZDV"

```

```

1 curl --location --request POST 'https://auth-dev.hmcn.tech/auth/realms/checkout/protocol/openid-connect/token' \
2 --header 'Content-Type: application/x-www-form-urlencoded' \
3 --data-urlencode 'grant_type=password' \
4 --data-urlencode 'username=testuser1' \
5 --data-urlencode 'password=xxx' \
6 --data-urlencode 'client_id=cpe_dev' \
7 --data-urlencode 'client_secret=xxx'

```

Refresh token:



POST

https://auth-dev.hm-btcloud.tech:9443/auth/realms/checkout/protocol/openid-connect/token

Send

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

Cookies

☐ none
 ☐ form-data
 ☒ x-www-form-urlencoded
 ☐ raw
 ☐ binary
 ☐ GraphQL

<input checked="" type="checkbox"/>	grant_type	refresh_token	
<input checked="" type="checkbox"/>	refresh_token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXLTJ1Iiwia2lkIiA6ICJRSjU1cS0xcXhuOE0zcy1pZnRLX0M5aHk5YWt2b3U5S2ZDV	
<input checked="" type="checkbox"/>	client_id	cpe_dev	
<input checked="" type="checkbox"/>	client_secret		
	Key	Value	Description

Body Cookies Headers (12) Test Results



200 OK

12.76 s

2.65 KB

Save Response

Pretty

Raw

Preview

Visualize

JSON



```

1 {
2   "access_token":
   "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXLTJ1Iiwia2lkIiA6ICJRSjU1cS0xcXhuOE0zcy1pZnRLX0M5aHk5YWt2b3U5S2ZDV"
   "E0wZF1DbjZIn0."
   "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXLTJ1Iiwia2lkIiA6ICJRSjU1cS0xcXhuOE0zcy1pZnRLX0M5aHk5YWt2b3U5S2ZDV"

```

```

1 curl --location --request POST 'https://auth-dev.hmcn.tech/auth/realms/checkout/protocol/openid-connect/token' \
2 --header 'Content-Type: application/x-www-form-urlencoded' \
3 --data-urlencode 'grant_type=refresh_token' \
4 --data-urlencode 'refresh_token=[previous refresh token]' \
5 --data-urlencode 'client_id=cpe_dev' \
6 --data-urlencode 'client_secret=xxx'

```

Summary of artifact created:

1 realm per function domain

1 client per environment (dev/prod) per microservice within the function domain or 1 client per environment (dev/prod) per microservice within the function domain per integration counterparty, if you want a different clientId/secret assigned to different integration counterparty. But this is not absolute necessary, as different integration counterparty will be assigned a different username/password, thus obtain different oauth token

1 role per permission group per client, for example we created two roles for common price engine, one for price/promotion uploading (write) and one for requesting price calculation (execute) and one super user role that can perform both operations.

1 user per microservice per environment per integration counterparty per application per role. Eg. cpe-data-importer-dev-vendor1 is the user generated for vendor1 in dev environment for common price engine with data importer role.

Enable oAuth2 flow for Spring boot applications:

1. Import springboot adaptors as pom dependencies

```

1 <dependencyManagement>
2   <dependencies>
3     <dependency>
4       <groupId>org.keycloak.bom</groupId>
5       <artifactId>keycloak-adapter-bom</artifactId>
6       <version>16.1.1</version>
7       <type>pom</type>
8       <scope>import</scope>
9     </dependency>
10  </dependencies>
11 </dependencyManagement>
12
13 <dependency>
14   <groupId>org.keycloak</groupId>
15   <artifactId>keycloak-spring-boot-starter</artifactId>
16 </dependency>

```

2. Inject keycloak related configurations in application.yml , the information should match configuration we exported in step 1.

```

1 keycloak:
2   realm: Checkout
3   auth-server-url: "http://keycloak-http.tech-foundation-dev/auth"
4   resource: "cpe_dev"
5   credentials:
6     secret: [match secret in keycloak client's installation]
7   ssl-required: external
8   use-resource-role-mappings: true
9   public-client: true

```



```

10 security-constraints:
11     - auth-roles:
12         - "cpe-super"
13         - "cpe-dataimport"
14     security-collections:
15         - patterns:
16             - "/checkout/cpeinbound/v1/*"
17         methods:
18             - "POST"
19             - "GET"

```

For a full list of configuration properties: see: https://www.keycloak.org/docs/latest/securing_apps/#_java_adapter_config

Note that the security-constraints are required, otherwise, the webservice endpoint will be un-protected (accessible even without a token). Here the roles will be the client role configured in keycloak admin console, and you can configure a list of allowed url patterns for each role.

In addition, for embedded tomcat for spring boot, there is a parameter **maxSavePostSize** need to be configured. Otherwise, keycloak adaptor will report an error:

```

1 java.io.IOException: Buffer overflow and no sink is set, limit [4,096] and buffer length [4,096]
2 at org.keycloak.adapters.tomcat.CatalinaAdapterSessionStore.saveRequest(CatalinaAdapterSessionStore.java:40)
3 at org.keycloak.adapters.tomcat.AbstractKeycloakAuthenticatorValve.keycloakSaveRequest(AbstractKeycloakAuthenticatorValve.java:100)
4 at org.apache.catalina.authenticator.FormAuthenticator.saveRequest(FormAuthenticator.java:686)
5 at org.apache.tomcat.util.buf.ByteChunk.append(ByteChunk.java:315)
6 at org.apache.tomcat.util.buf.ByteChunk.flushBuffer(ByteChunk.java:515)
7 "Caused by: java.io.IOException: Buffer overflow and no sink is set, limit [4,096] and buffer length [4,096]"

```

The parameter can be injected via TomcatConnectorCustomizer

```

1 @Bean
2 open fun tomcatConnectorCustomizer(): TomcatConnectorCustomizer {
3     return TomcatConnectorCustomizer { connector ->
4         connector.maxSavePostSize = 2097152
5     }
6 }

```

Enable oauth2 flow for kubernetes container

A keycloak proxy can be injected into the kubernetes container as a sidecar. Incoming restful requests that are sent to the pod will be intercepted by the sidecar first (controlled via port exposed by sidecar instead of the main container). The sidecar checks whether oauth token exists in the request header and perform verification according. Only when the verification is successful, the request will be forwarded to the main container.

Keycloak gatekeeper:

The official docker image for <https://hub.docker.com/r/keycloak/keycloak-gatekeeper/> is end of life and is replaced by <https://hub.docker.com/r/bitnami/keycloak-gatekeeper>

Installation:

Step 1: Create a configMap object that hosts the proxy configuration file: The configMap will be injected into the proxy container via volume mount.

```

1 {
2     "listen": "0.0.0.0:30000",
3     "upstream-url": "http://127.0.0.1:8080",

```

```

4  "discovery-url": "https://auth-dev.hmcn.tech/auth/realms/checkout",
5  "openid-provider-proxy": "http://keycloak-http.tech-foundation-dev",
6  "client-id": "cpe_dev",
7  "client-secret": "xxxx",
8  "resources":[
9    {"uri": "/restapi",
10     "methods": ["POST"],
11     "roles": ["cpe-super"]}
12  ],
13  {"uri": "/test/*",
14   "white-listed": true
15  }
16  ]
17  }

```

Where the required attributes are

“listen”: local listening interface ipaddress and port, note ipaddress must be configured as **0.0.0.0** instead of 127.0.0.1, otherwise, external request will be rejected.

“upstream-url”: the main container’s host port, in this case, host will be 127.0.0.1

“discovery-url”: the realm url of the keycloak server’s

“client-id”: the configured client id

“client-secret”: the configured client secret

Step 2: Make change to kubernetes deployment

```

1  spec:
2    containers:
3      - name: main container
4        image: [main container image name]
5      - name: keycloakgateway
6        image: >-
7          registry-vpc.cn-beijing.aliyuncs.com/bt-apac-service/keycloak-gatekeeper:latest
8        command:
9          - /keycloak-gatekeeper
10       args:
11         - '--config=/opt/bitnami/keycloak-gatekeeper/data'
12       imagePullPolicy: IfNotPresent
13       ports:
14         - containerPort: 30000
15           protocol: TCP
16       volumeMounts:
17         - mountPath: /opt/bitnami/keycloak-gatekeeper
18           name: volume-1647882528665
19     volumes:
20       - configMap:
21         name: keycloak-gatekeeper-config
22         name: volume-1647882528665

```

Note:

A new sidecar container keycloakgateway created.

The configMap will be mounted as a volume with path “/opt/bitnami/keycloak-gatekeeper”, and “data” is the key of the configMap. This becomes the full path of the gateway config file which matches the argument “config”.

main container’s port does not have to be exposed, we will expose keycloak gateway’s local listener port instead.

Step 3: Make changes to kubernetes service

Service port mapping will be changed from 8080 → 80 to 30000 → 80. So that incoming request will be forwarded to the sidecar's listening port.

Pros:

Support fine grain role control that is similar to spring boot adapter

Cons:

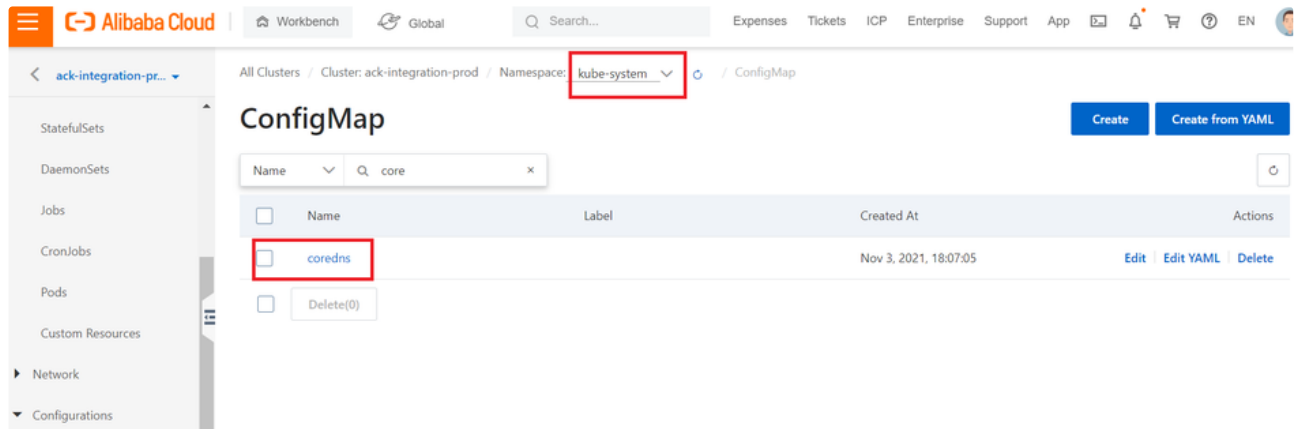
Does not support "discovery-url" to be external domain name. Otherwise, when the proxy starts and tries to retrieve . It will report the following error:

```
1 warn    failed to get provider configuration from discovery {"error": "\"issuer\" in config (http://keycloak-http-tech-foundation-dev/auth/realms/Checkout/.well-known/openid-configuration) does not match provided issuer URL (https://auth-dev.hm-btcloud.tech:9443/auth/realms/checkout)"}
2
```

Where the former is derived from discover-url property in gateway config, and the latter is derived from "issuer" property value in `http://[keycloak server address]/auth/realms/[domain name]/.well-known/openid-configuration`

```
root@cpe:/usr/local/tomcat# curl http://keycloak-http-tech-foundation-dev/auth/realms/Checkout/.well-known/openid-configuration
{"issuer": "http://keycloak-http-tech-foundation-dev/auth/realms/Checkout", "authorization_endpoint": "http://keycloak-http-tech-foundation-dev/auth/realms/Checkout/protocol/openid-connect/auth", "token_endpoint": "http://keycloak-http-tech-foundation-dev/auth/realms/Checkout/protocol/openid-connect/token", "introspection_endpoint": "http://keycloak-http-tech-foundation-dev/auth/realms/Checkout/protocol/openid-connect/token/introspect", "userinfo_endpoint": "http://keycloak-http-tech-foundation-dev/auth/realms/Checkout/protocol/openid-connect/userinfo", "end_session_endpoint": "http://keycloak-http-tech-foundation-dev/auth/realms/Checkout/protocol/openid-connect/logout", "jwks_uri": "http://keycloak-http-tech-foundation-dev/auth/realms/Checkout/protocol/openid-connect/certs", "check_session_iframe": "http://keycloak-http-tech-foundation-dev/auth/realms/Checkout/protocol/openid-connect/login-status-iframe.html", "grant_types_supported": ["authorization_code", "implicit", "refresh_token", "password", "client_credentials"]}
```

To resolve this issue, The keycloak's installation property "KEYCLOAK_FRONTEND_URL" has to be set to internal domain name (Eg. `http://keycloak-http-tech-foundation-dev`). But this prevents admin console to be accessed by external domain name anymore. If we want the external domain name to be resolved correctly to ip address within kubernetes cluster, we have to change the **core-dns** config-map under kube-system namespace.



Key	Value
Corefile	<pre> .:53 { errors health { lameduck 15s } ready rewrite stop { name regex auth-dev.hm-btcloud.tech keycloak-http.tech-foundation-dev.svc.hm.local answer name keycloak-http.tech-foundation-dev.svc.hm.local auth-dev.hm-btcloud.tech } kubernetes hm.local in-addr.arpa ip6.arpa { pods verified fallthrough in-addr.arpa ip6.arpa } prometheus :9153 forward . /etc/resolv.conf cache 30 loop reload loadbalance } </pre>

Oauth proxy:

Official website: <https://oauth2-proxy.github.io/oauth2-proxy/docs/>

Official docker image: quay.io/oauth2-proxy/oauth2-proxy

Installation:

Step1: Create a configMap object that hosts the proxy configuration file: The configMap will be injected into the proxy container via volume mount.

```

1 http_address = "0.0.0.0:30000"
2 upstreams = [
3     "http://127.0.0.1:8080"
4 ]
5 provider = "keycloak-oidc"
6 client_id = "cpe_dev"
7 client_secret = "e4e06214-b258-4b9d-a37d-16bd261914d2"
8 redirect_url = "https://myapp.com/oauth2/callback"
9 oidc_issuer_url = "https://auth-dev.hmcn.tech/auth/realms/Checkout"
10 email_domains = "*"
11 cookie_secret = "xxx"

```

Note that the cookie_secret is a mandatory attribute. It can be generated via the following command in linux:

```
python -c 'import os,base64; print base64.b64encode(os.urandom(16))'
```

Step 2: Make change to kubernetes deployment

```

1 spec:
2   containers:
3     - name: main container
4       image: [main container image name]
5     - name: oauth-proxy
6       image: 'quay.io/oauth2-proxy/oauth2-proxy:latest'
7     args:
8       - '--config=/opt/oauth-proxy/data'
9     imagePullPolicy: IfNotPresent
10    ports:

```

```
11       - containerPort: 30000
12         protocol: TCP
13       volumeMounts:
14         - mountPath: /opt/oauth-proxy
15           name: volume-1647869399928
16     volumes:
17     - configMap:
18       name: oauth-proxy-config
19     name: volume-1647869399928
```

The deployment configuration will be almost the same with keycloak-gateway, so refer to the previous section for detailed explanation.

Step 3: Make changes to kubernetes service

Service port mapping will be changed from 8080 → 80 to 30000 → 80. So that incoming request will be forwarded to the sidecar's listening port.

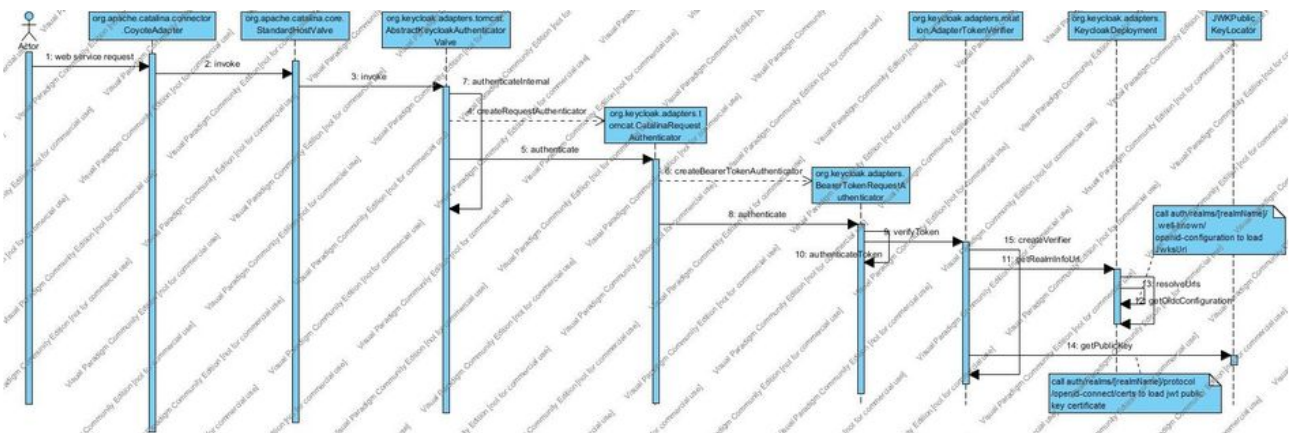
Pros:

Has more backend oauth providers supported out of box, and not limited to key-cloak

Cons:

No fine grained access right control by url to different roles

SpringBoot keycloak authorization under the hood:



Bootstrap:

`org.keycloak.adapters.springboot.KeycloakSpringBootProperties` captures keycloak related configurations in `application.yml`.

`org.keycloak.adapters.springboot.KeycloakAutoConfiguration` registers `KeycloakBaseTomcatContextCustomizer` which injects `securityConstraints` defined in `KeycloakSpringBootProperties` into tomcat server configuration.

`org.keycloak.adapters.springboot.KeycloakAutoConfiguration` also registers `org.keycloak.adapters.tomcat.KeycloakAuthenticatorValve` which performs the actual oauth2 token verification.

Authorization:

Once the client sends request to the restful webservice, it will pass `KeycloakAuthenticatorValve` to perform the bearer token verification. `KeycloakAuthenticatorValve` will constructs a `KeycloakDeployment` first, then constructs a `RequestAuthenticator` with the `KeycloakDeployment` as a parameter. `RequestAuthenticator`'s `authenticate` method will in turn create a `BearerTokenRequestAuthenticator` which does the actual work. Internally, `KeycloakDeployment` will send a request to `auth/realms/[realmName]/.well-known/openid-configuration` on lazy loading to get the `jwksUrl`. It then sends a request to `auth/realms/[realmName]/protocol/openid-connect/certs` to load public key, which is used to verify the signature of the bearer token.

After token verification is performed, `RequestAuthenticator.completeAuthentication` will create the principal with roles parsed from bearer token and save it in the `HttpRequest`.

Finally, the roles in the principal will be matched against the security constraints configured in `application.yml`. See `org.apache.catalina.realm.RealmBase.hasResourcePermission`

Compare keycloak and springboot oauth2:

- Key cloak supports both authentication and authorization. This becomes very handy as realm admin can manage user and oauth roles in a centralized spot. Springboot oauth server, however only handles authorization, additional development work is required to manage user and associate user role with oauth scope.
- Key cloak adapter for springboot application performs interception at webcontainer layer (Tomcat) while springboot oauth performs interception via spring security filters
- Key cloak supports a wide range of integrations with other types of applications such as nodejs, nginx, envoy etc. while springboot oauth2 is specific for spring boot application.
- Key cloak supports adaptor in application as well as sidecar injection for container deployment, while spring boot oauth2 only supports adaptor. Supporting both mode makes key cloak very flexible and less intrusive into the resource server application itself. Especially when the resource server application is thirdparty, where you don't have right to modify it, key cloak is the only sensible solution.

Reference:

<https://www.jianshu.com/p/a845cc38abe2>

<https://www.jianshu.com/p/d65aceb0246e>

<https://www.jianshu.com/p/b3c5e536de9b>

https://www.keycloak.org/docs/latest/securing_apps/#_spring_boot_adapter

<https://www.baeldung.com/postman-keycloak-endpoints>

https://blog.csdn.net/little_kelvin/article/details/111239241

<https://www.springcloud.io/post/2022-02/keycloak-springboot/>

<https://cloud.tencent.com/developer/article/1468310>

<https://blog.fleeto.us/post/sidecar-oauth-for-kubernetes-apps/>

https://help.aliyun.com/document_detail/380963.html