Alicloud oauth with keycloak

Local keycloak instance:

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

Configure client & user for oauth flow:

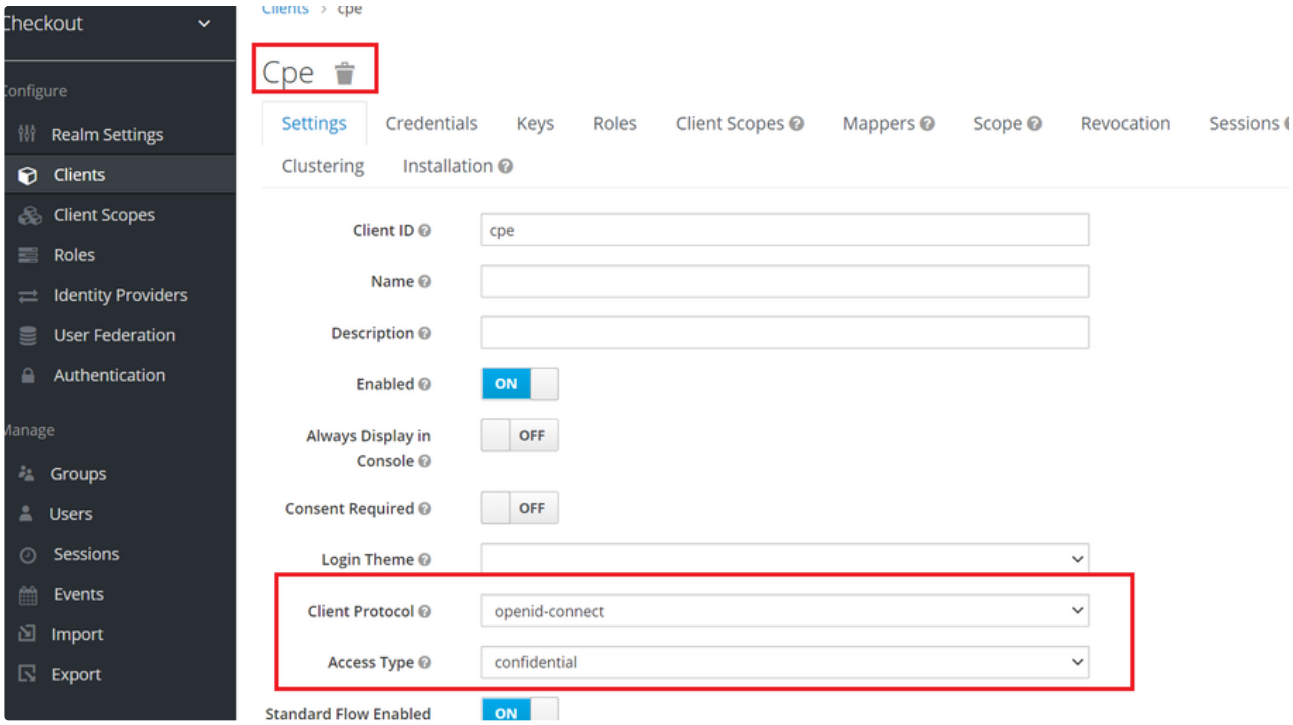
Create a realm:

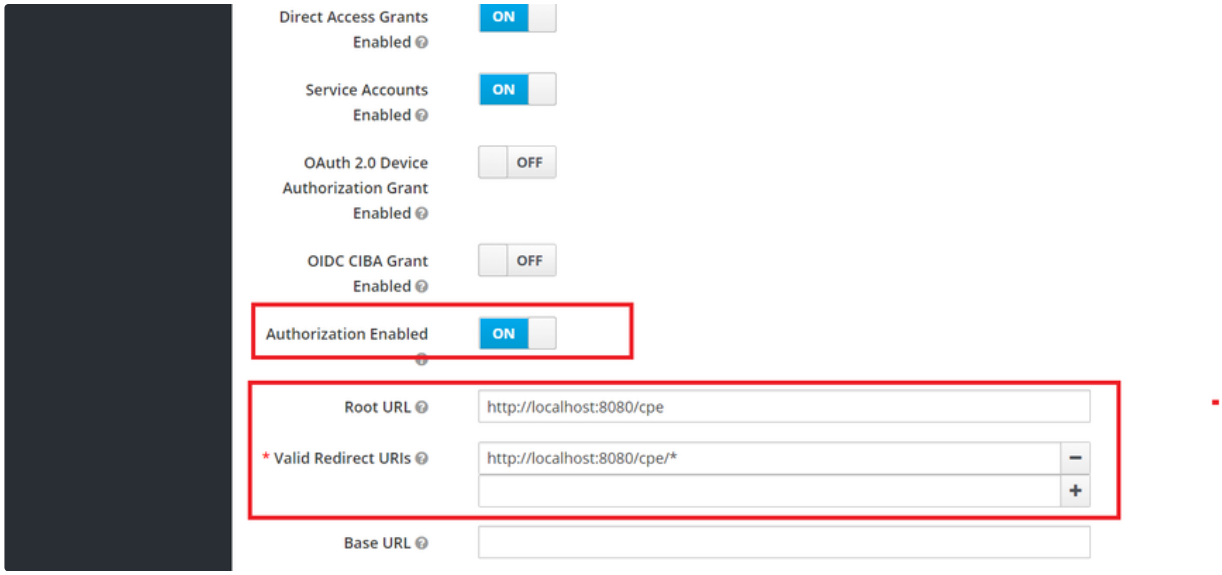
The realm maps to a function domain in the organization. There can be many protected resources under the realm.

Create client:

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

Configure → clients → create:



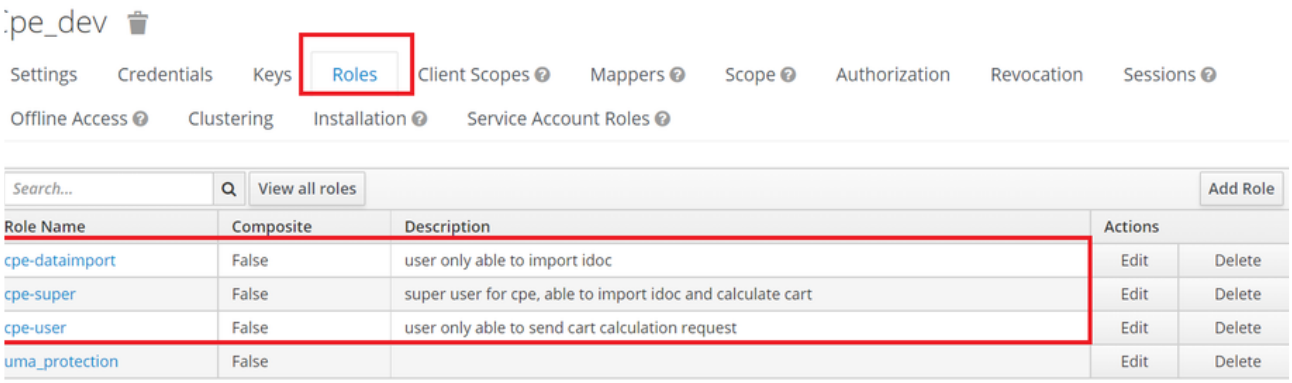


“Access Type” needs to be “confidential”, otherwise, there will be no client secret generated.

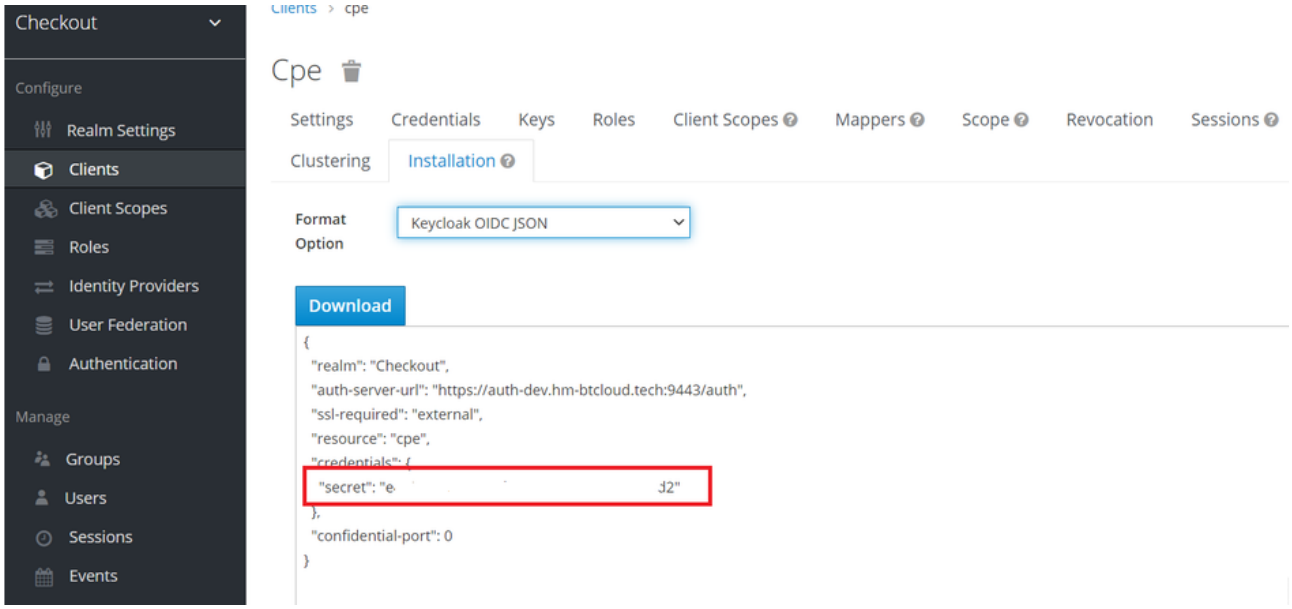
“Authorization Enabled” needs to be set to “on”

“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!



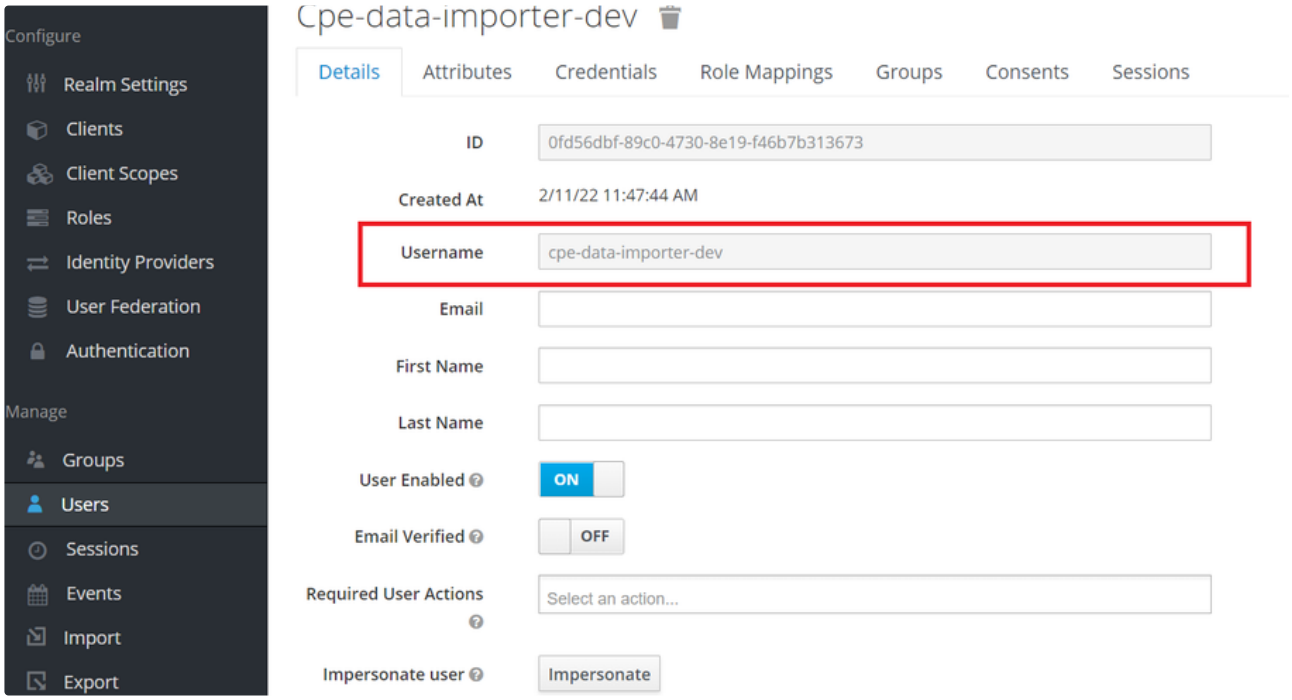
After saving, oauth configuration can be viewed and exported via Configurer → clients → installation



Create user:

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

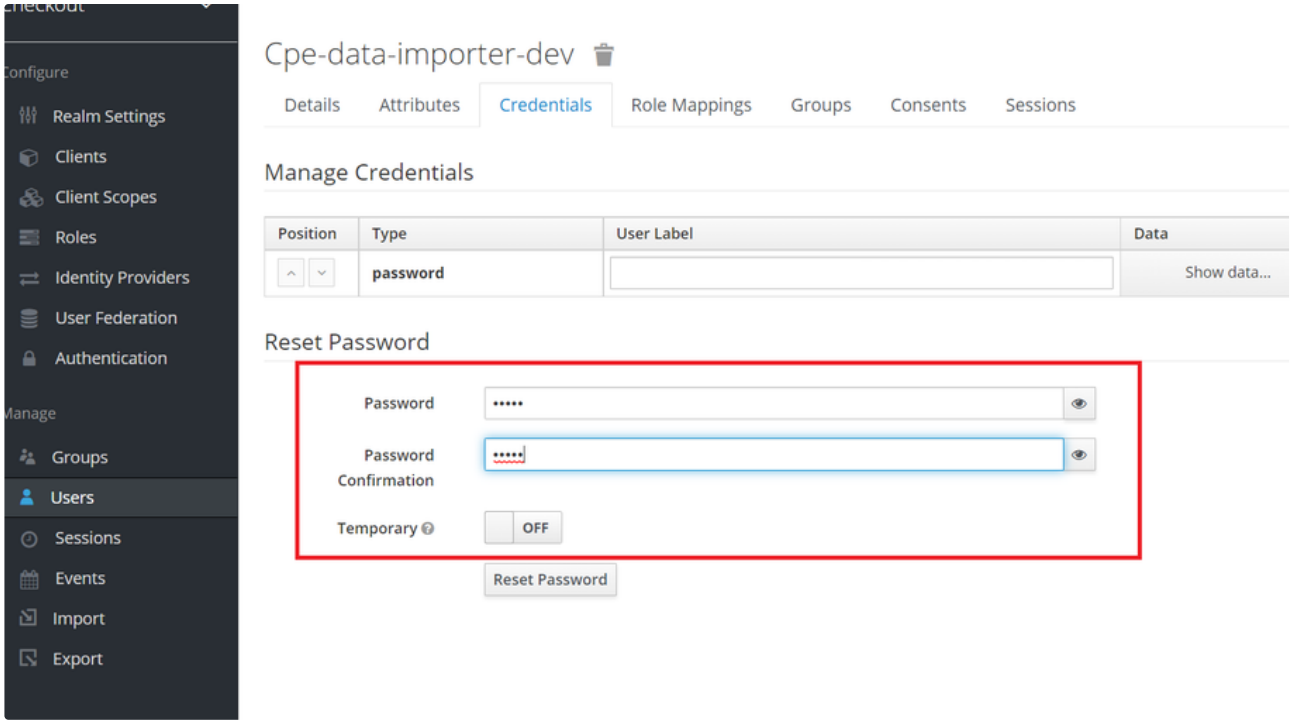
Manage → Users → create:



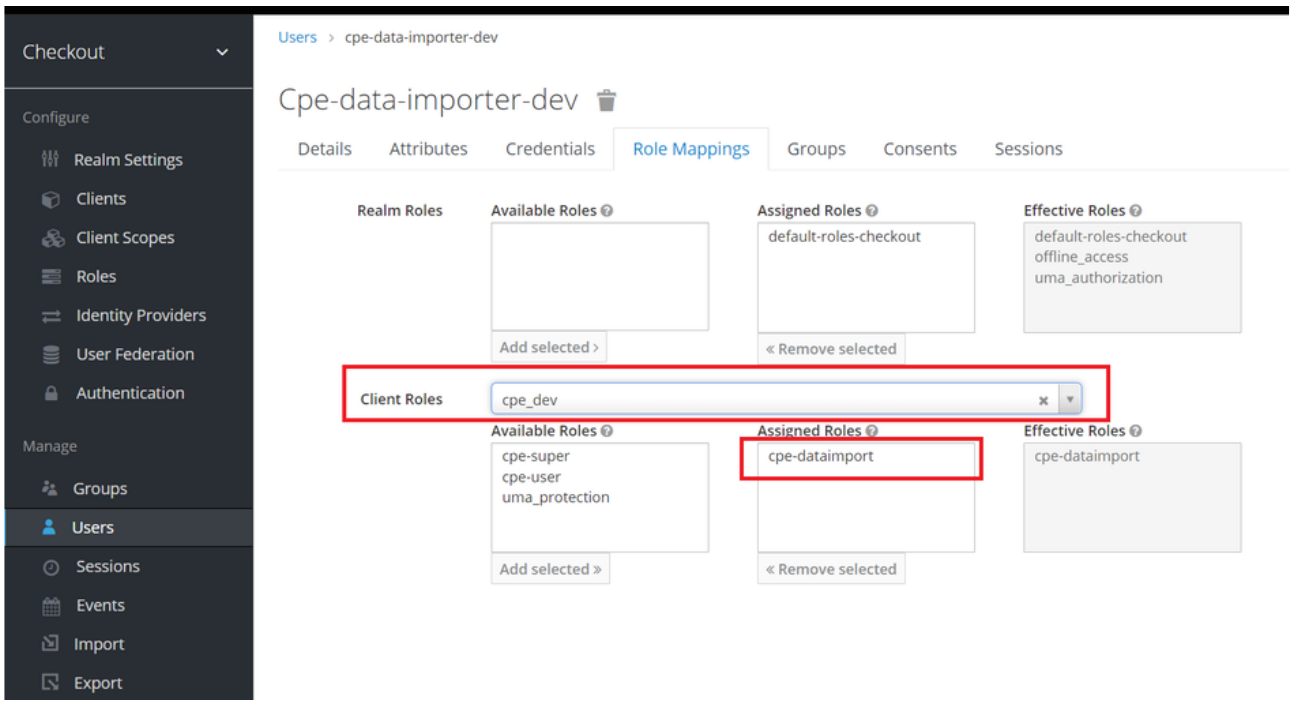
Only username is required here.

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

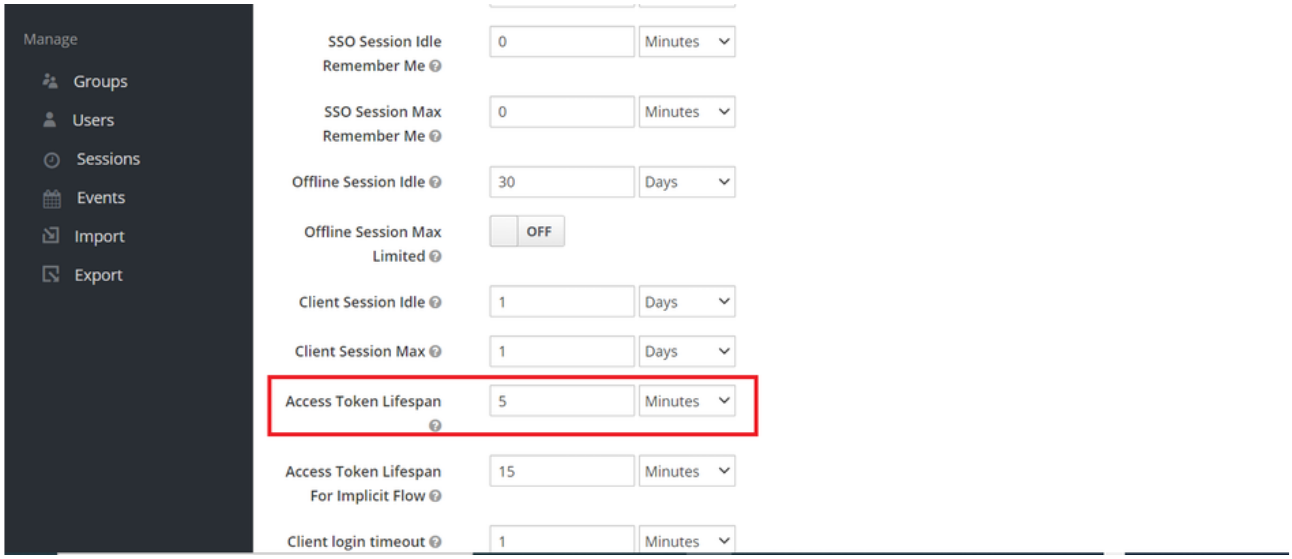


After creating the user, we also need to attach 1 or more client roles to it, so that the role information will be included in the generated oauth token.

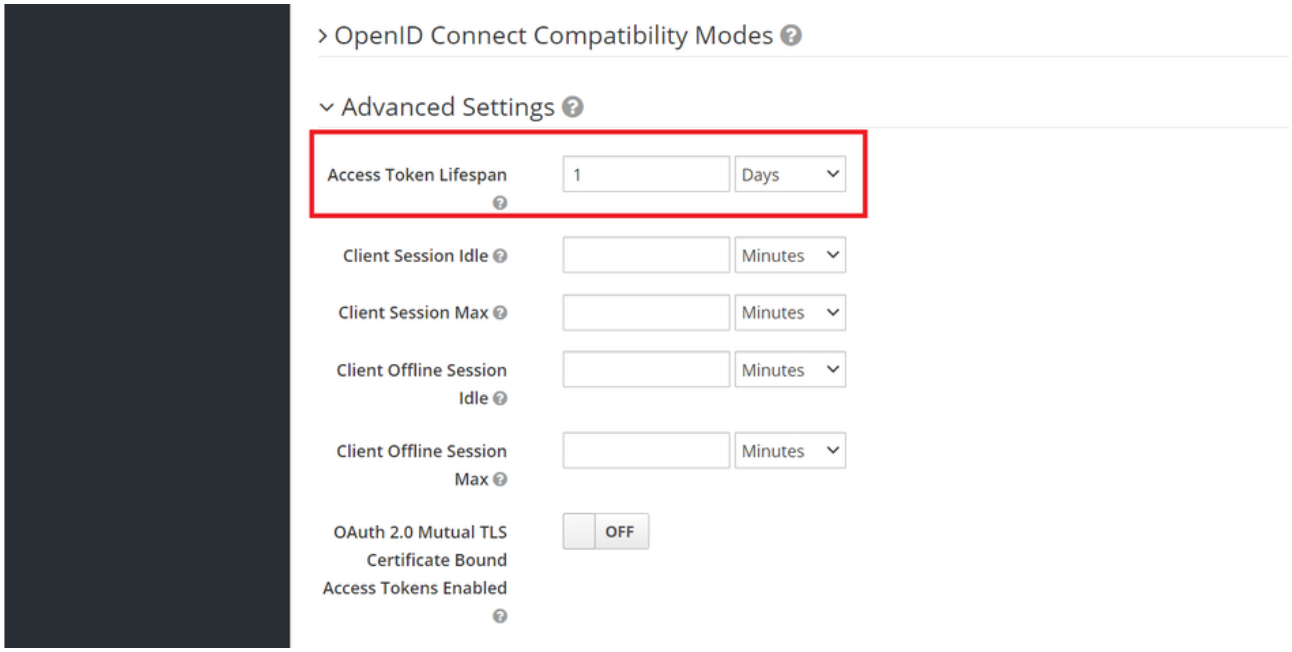


Configure token expiration time:

Global level: Realm Settings → Tokens → Access Token Lifespan

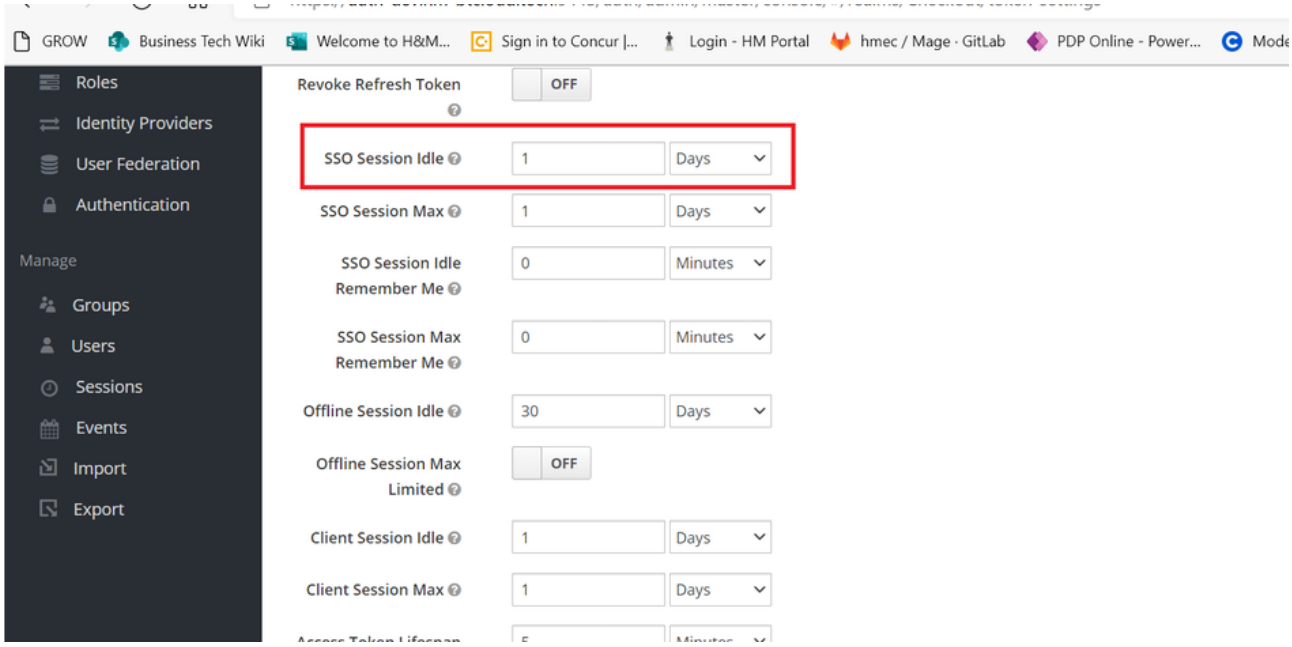


Per client override: Clients → [client name] → Settings → Advanced settings → Access Token Lifespan

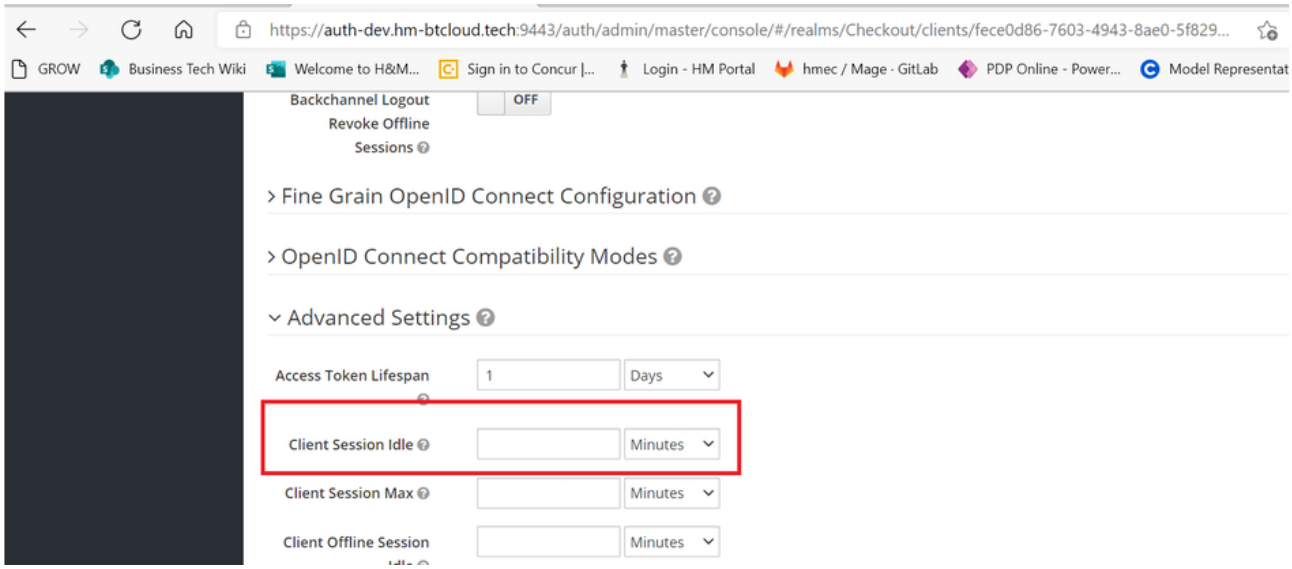


Refresh token expiration time:

Global level: Realm Settings → Tokens → SSO Session Idle

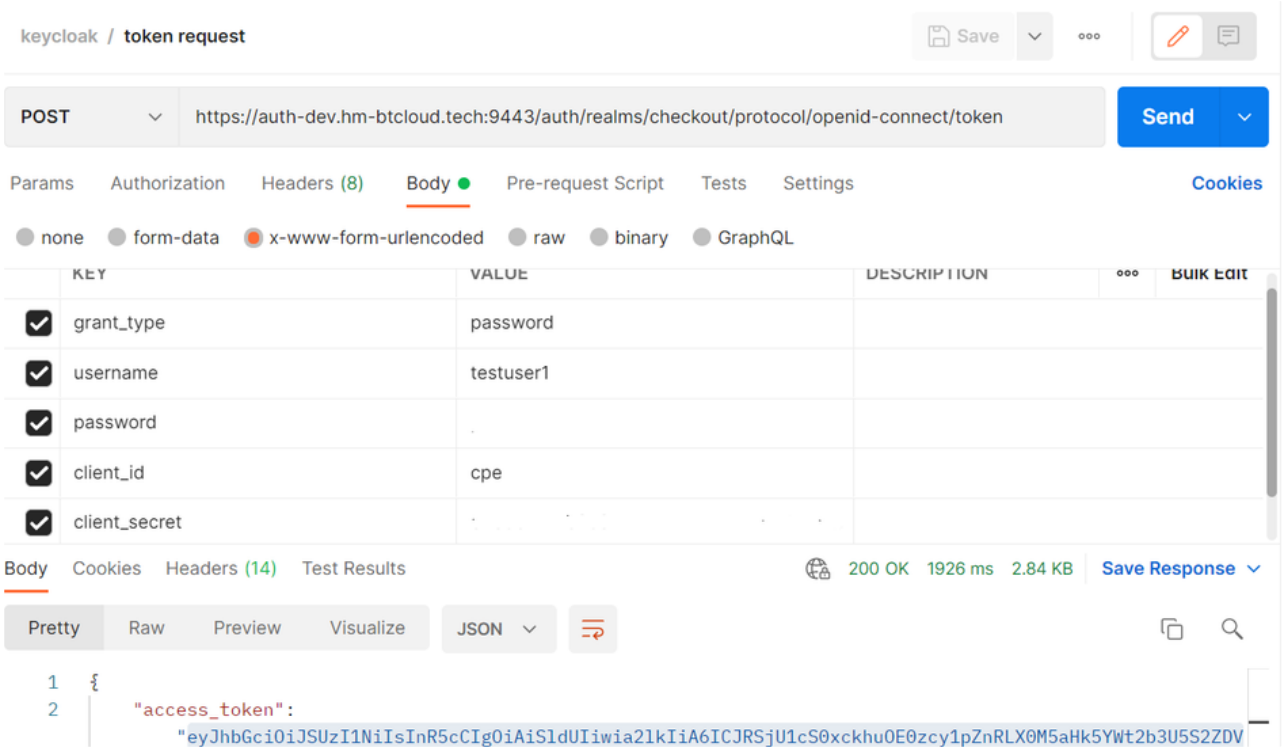


Per client override: Clients → [client name] → Settings → Advanced settings → Client Session Idle (this will only take effect if the configured value is **less than Tokens → SSO Session Idle**. Otherwise, the configuration will not take effect and token refresh period falls back to Tokens → SSO Session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



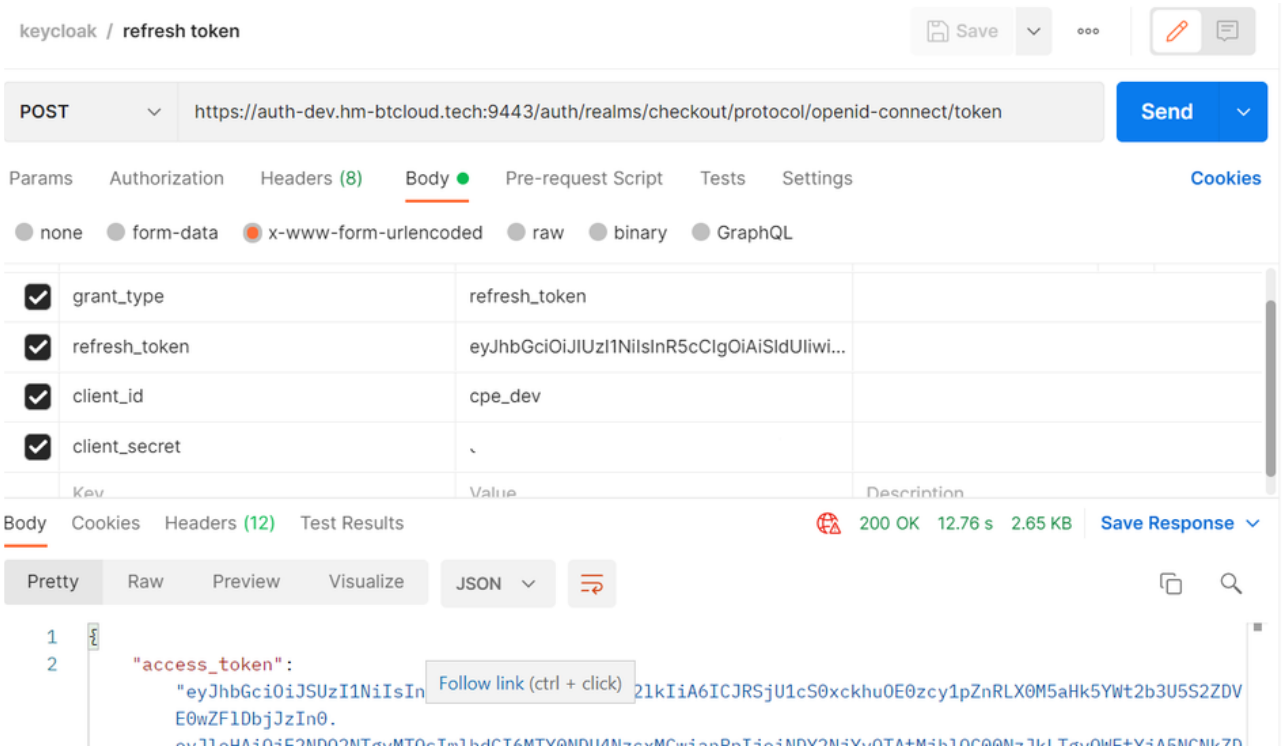
Testing oauth token acquisition:

Request oauth token via password flow:




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

Refresh token:



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

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.keycloak.bom</groupId>
      <artifactId>keycloak-adapter-bom</artifactId>
      <version>16.1.1</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependency>
    <groupId>org.keycloak</groupId>
    <artifactId>keycloak-spring-boot-starter</artifactId>
</dependency>
```

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

```
keycloak:
  realm: Checkout
  auth-server-url: "http://keycloak-http.tech-foundation-dev/auth"
  resource: "cpe_dev"
  credentials:
    secret: [match secret in keycloak client's installation]
  ssl-required: external
  use-resource-role-mappings: true
  public-client: true
```

```
  security-constraints:
    - auth-roles:
        - "cpe-super"
        - "cpe-dataimport"
      security-collections:
        - patterns:
            - "/checkout/cpeinbound/v1/*"
          methods:
            - "POST"
            - "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:

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

```
    @Bean
    open fun tomcatConnectorCustomizer(): TomcatConnectorCustomizer {
        return TomcatConnectorCustomizer { connector ->
            connector.maxSavePostSize = 2097152
        }
    }
```

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.

```
{
 "listen": "0.0.0.0:30000",
 "upstream-url": "http://127.0.0.1:8080",
```

```
 "discovery-url": "https://auth-dev.hmcn.tech/auth/realms/checkout",
 "openid-provider-proxy": "http://keycloak-http.tech-foundation-dev",
 "client-id": "cpe_dev",
 "client-secret": "xxxx",
 "resources":[
   {"uri": "/restapi",
    "methods": ["POST"],
    "roles": ["cpe-super"]
   },
   {"uri": "/test/*",
    "white-listed": true
   }
 ]
}
```

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

```
    spec:
      containers:
        - name: main container
          image: [main container image name]
        - name: keycloakgateway
          image: >-
            registry-vpc.cn-beijing.aliyuncs.com/bt-apac-service/keycloak-gatekeeper:latest
          command:
            - /keycloak-gatekeeper
          args:
            - '--config=/opt/bitnami/keycloak-gatekeeper/data'
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 30000
              protocol: TCP
          volumeMounts:
            - mountPath: /opt/bitnami/keycloak-gatekeeper
              name: volume-1647882528665
      volumes:
        - configMap:
            name: keycloak-gatekeeper-config
          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

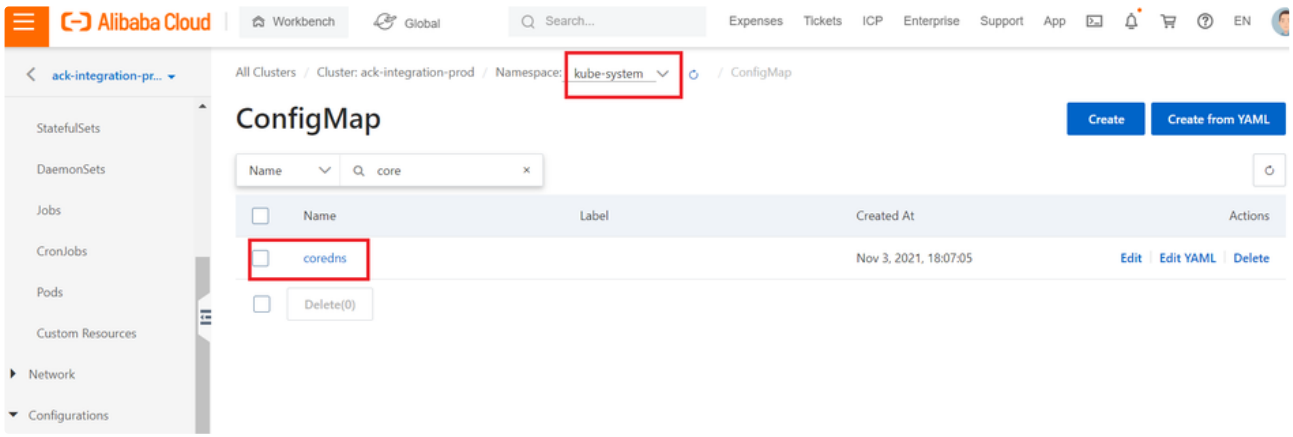
Cons:

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:

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

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

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	.: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
	}

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.

```
http_address = "0.0.0.0:30000"
upstreams = [
    "http://127.0.0.1:8080"
]
provider = "keycloak-oidc"
client_id = "cpe_dev"
client_secret = "e4e06214-b258-4b9d-a37d-16bd261914d2"
redirect_url = "https://myapp.com/oauth2/callback"
oidc_issuer_url = "https://auth-dev.hmcn.tech/auth/realms/Checkout"
email_domains = "*"
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

```
    spec:
      containers:
        - name: main container
          image: [main container image name]
        - name: oauth-proxy
          image: 'quay.io/oauth2-proxy/oauth2-proxy:latest'
          args:
            - '--config=/opt/oauth-proxy/data'
          imagePullPolicy: IfNotPresent
          ports:
```

```
            - containerPort: 30000
              protocol: TCP
          volumeMounts:
            - mountPath: /opt/oauth-proxy
              name: volume-1647869399928
      volumes:
        - configMap:
            name: oauth-proxy-config
          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.

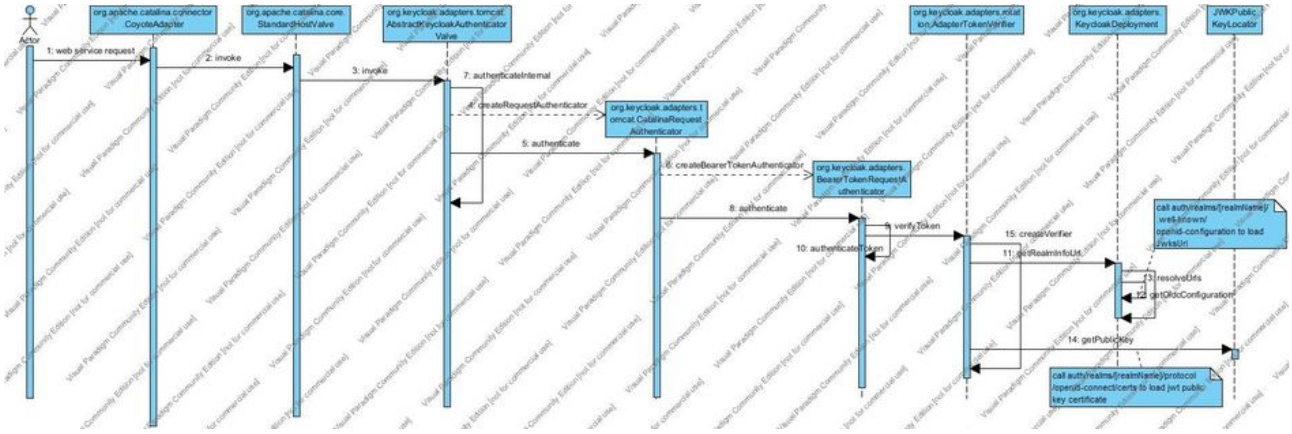
Pros:

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 securityContraints 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