Kong THE CLOUD CONNECTIVITY COMPANY

Kong Gateway Operations for Kubernetes

OIDC Plugin

Course Agenda

1. Kong Gateway Installation on K8s
2. Intro to Kong Ingress Controller
3. Securing Kong Gateway
4. Securing Services on Kong
5. **OIDC Plugin**
6. Kong Vitals
7. Advanced Plugins Review
8. Troubleshooting
9. Test your Knowledge

Learning Objectives

1. Understand and explain how to integrate the OIDC plugin
2. Be able to administer OIDC plugin
3. Understand some basic OIDC workflows

Reset Environment

Before we start this lesson, be sure to reset your environment:

```
$ cd /home/labuser
$ git clone https://github.com/Kong/edu-kgac-202.git
$ source ./edu-kgac-202/base/reset-lab.sh
```

What is OIDC?

What is OIDC?

OpenID Connect/OAuth 2.0 authentication framework provides multiple ways to secure services for different scenarios, including User to Service and Service to Service scenarios.

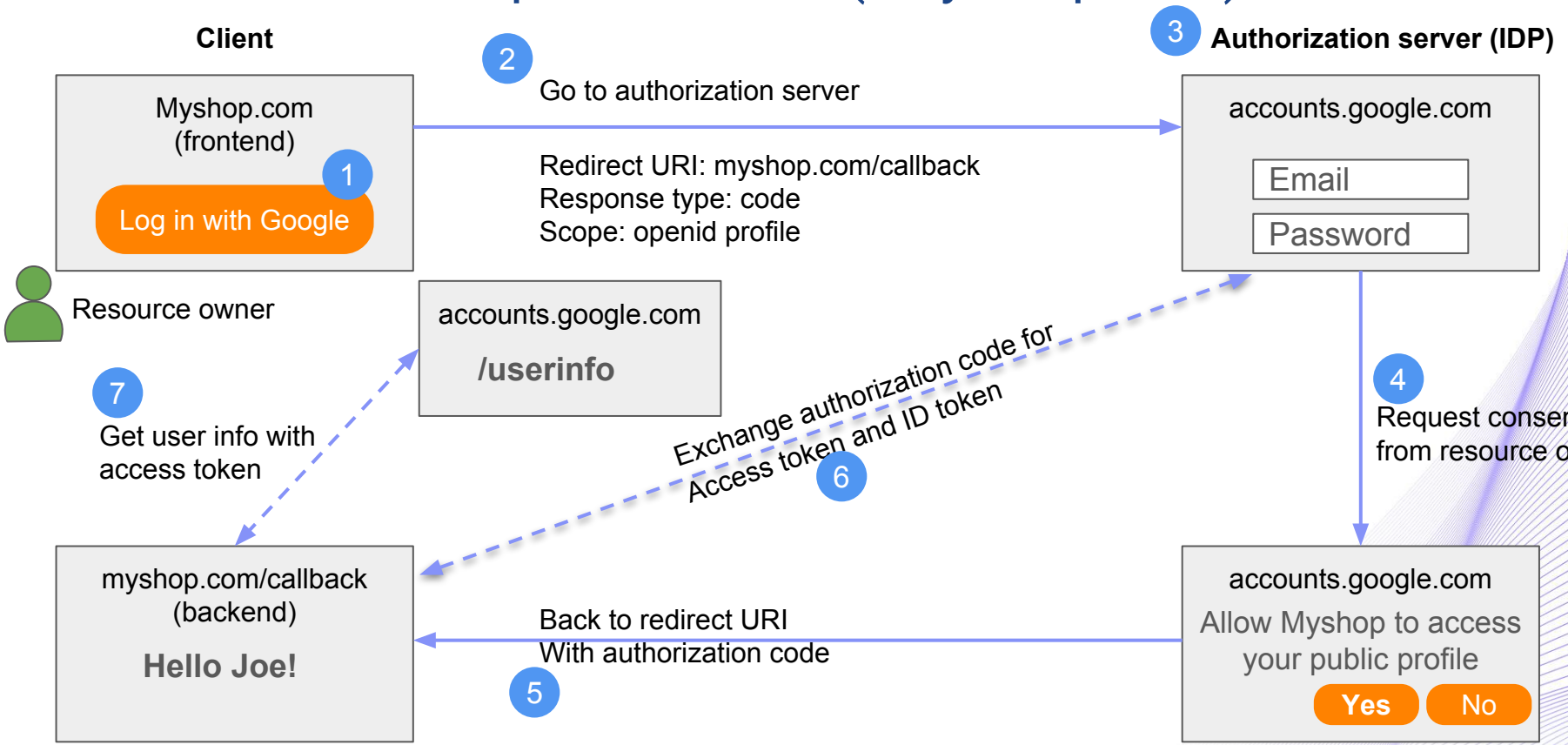
A detailed description of OIDC is beyond the scope of this course

Kong supports many authentication frameworks and standards to properly govern and secure Services. For the OIDC authentication framework, Kong supports a variety of 3rd party providers such as Auth0, Okta, MS Azure ID, Google, Amazon AWS Cognito, etc. As long as a provider supports OpenID Connect standards, Kong should be able to work with it.

An identity provider (IdP) is a system entity that creates, maintains, and manages identity information for principals and also provides authentication services to relying applications within a federation or distributed network.

For this workshop, we will be using a 3rd party Identity Provider (IdP), Keycloak.

OIDC Sample Workflow (very simplified)



The Kong OIDC Plugin

The OpenID Connect plugin standardizes the integration with 3rd party identity providers. The plugin can be used to implement Kong as a (proxying) OAuth 2.0 Resource Server (RS) and/or as an OpenID Connect Relying Party (RP) between the client and the upstream service.

Kong's OIDC plugin is quite versatile, with around 200 parameter settings. It would be useful to know what combination of configurations are best fitted for your use-case. It supports a number of credential and grant types:

- Signed JWT access tokens (JWS)
- Opaque access tokens
- Refresh tokens
- Authorization code
- Username and password
- Client credentials
- Session cookies

Deploy an Identity Provider (IDP)

Task: Deploy Keycloak

Deploying and configuring an IDP is beyond the scope of this course, so we have deployed Keycloak by way of a container. To review the configuration for the container look at the keycloak section of the docker-compose.yaml file used in this course.

```
$ cd ~/edu-kgac-202
$ ./docker-containers/deploy.sh
```

Before we continue, it is worth noting that during the startup of keycloak, a number of users were provisioned via the configuration file `/data/kong_realm_template.json`:

```
$ cd ~/edu-kgac-202/exercises/oidc
$ cat ./kong_realm_template.json | jq '.users[].username'
```

```
"admin"
"employee"
"partner"
"service-account-kong"
```

Deploying the OIDC Plugin

Task: Add a Service to use with OIDC

```
$ cd /home/labuser/edu-kgac-202/exercises/oidc
$ cat << EOF > ./httpbin-oidc-plugin.yaml
---
apiVersion: v1
kind: Namespace
metadata:
  name: httpbin-demo
---
EOF
$
```

Task: Add a Service to use with OIDC

```
$ cat << EOF >> ./httpbin-oidc-plugin.yaml
apiVersion: configuration.konghq.com/v1
kind: KongPlugin
metadata:
  name: httpbin-oidc
  namespace: httpbin-demo
plugin: openid-connect
config:
  issuer: $KEYCLOAK_CONFIG_ISSUER
  client_id:
  - kong
  client_secret:
  - $CLIENT_SECRET
  response_mode: form_post
  redirect_uri:
  - https://$KEYCLOAK_REDIRECT_URI
  ssl_verify: false
EOF
```

Task: Add a Service to use with OIDC

```
$ kubectl apply -f ./httpbin-oidc-plugin.yaml
namespace/httpbin-demo created
kongplugin.configuration.konghq.com/httpbin-oidc created
$ kubectl apply -f ./httpbin-ingress-oidc.yaml
namespace/httpbin-demo unchanged
service/oidc-service created
ingress.networking.k8s.io/oidc-route created
$
```

OpenID Connect Plugin Configuration Parameters

We've pre-created variables with the following values the OpenID Connect plugin expects:

- `config.issuer` - This parameter tells the plugin where to find discovery information
- `config.client_id` - The client_id of the OpenID Connect client registered in OpenID Connect Provider
- `config.client_secret` - The client_secret of the OpenID Connect client registered in OpenID Connect Provider
- `config.redirect.uri` - This parameter defines the URL the IDP will redirect the user to after a successful authentication
- `config.response_mode` - This parameter specifies the response mode the IDP should respond with
- `config.ssl_verify` - This parameter is set to false for this environment since the Keycloak uses a self signed certificate

Task: Verify Protected Service

Let's make a call to the API that is now protected with the OpenID Connect plugin

```
$ http GET kongcluster:30000/oidc
```

```
HTTP/1.1 302 Moved Temporarily
...
```

The response should be `HTTP 302 Moved Temporarily`. This is due to all traffic hitting our route now needing to be properly authenticated. Now try

```
$ http GET kongcluster:30000/oidc -a user:password
```

```
HTTP/1.1 401 Unauthorized
...
```

The response should be `HTTP 401 Unauthorized` as '`user`' was not specified in the Keycloak configuration we loaded.

Task: View Kong Discovery Information from IDP

You can view that Kong has loaded the discovery information from the IDP

```
$ http GET kongcluster:30001/openid-connect/issuers
```

```
HTTP/1.1 200 OK
...
```

You can filter the response to get to specific information:

```
$ http -b GET kongcluster:30001/openid-connect/issuers | jq -r .data[].issuer
```

```
https://8080-1-ebf469fd.labs.konghq.com/auth/realms/kong/.well-known/openid-configuration
```

Opening the above URL in a browser will show you the current configuration of the plugin.

OIDC Examples

Now that we have OIDC configured, let's look at a few examples.

There are many different authentication scenarios to choose from, for example.

- password grant
- client credentials
- authorization code
- bearer token

and a number of authorization scenarios relating to for example

- Roles
- Audience Required

In this course, we will cover a selection of authentication and authorization scenarios

Authentication with Password Grant

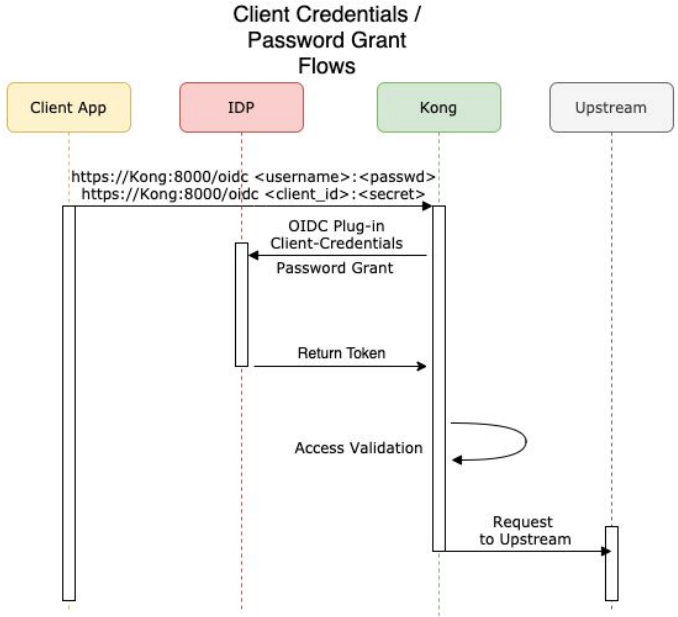
Password Grant Authorization Flow

In this section, we will use Password Grant. With this grant, Kong will call the token endpoint of the IDP on behalf of the end user to obtain short-lived tokens.

In Keycloak, a `user=employee` with `credentials=test` has already been created. Let's utilize this user for the password grant.

Note: The Password Grant flow is used in legacy use cases because the client application has to collect and send a user's credentials to the IDP to authenticate the user. The latest OAuth 2.0 Security Best Current Practice disallows the password grant entirely.

A more modern, secure, approach is to use the Authorization Code flow.



Task: Confirm Keycloak is configured for Password Grant

Before getting an access token for user, let's confirm Keycloak is configured for Password Grant, and that password can be passed in request body/header/query:

```
$ OIDC_PLUGIN_ID=$(http GET \
    kongcluster:30001/routes/httpbin-demo.oidc-route.00/plugins/ \
    | jq -r '.data[] | select(.name == "openid-connect") | .id')
```

```
$ http -b GET kongcluster:30001/plugins/$OIDC_PLUGIN_ID \
    | jq .config.auth_methods
```

```
  "Password", . . .
```

```
$ http -b GET kongcluster:30001/plugins/$OIDC_PLUGIN_ID \
    | jq .config.password_param_type
```

```
  "header",
  "query",
  "body"
```

Task: Provide credentials to Kong and retrieve Access Token

Use employee/test credentials to get an access token for the user:

```
$ http GET kongcluster:30000/oidc -a employee:test
```

The response should be a HTTP 200 OK. Take note of the bearer token and session cookie in the headers. This is the short-lived token that can be used by the client application to access protected endpoints. The service we're using simply echoes back what's been sent. You can decode this token at https://jwt.io or from CLI to inspect the contents of it.

```
$ AUTHORIZATION_INFO=$(http kongcluster:30000/oidc -a employee:test | jq -r
'.headers.Authorization' | cut -c 7-)

$ jwt -d $AUTHORIZATION_INFO | jq
```

As you can see, the main disadvantage of password grant is that user credentials are exposed to the client application.

Authenticate with Bearer Token Grant

Authenticate with Bearer Token Grant

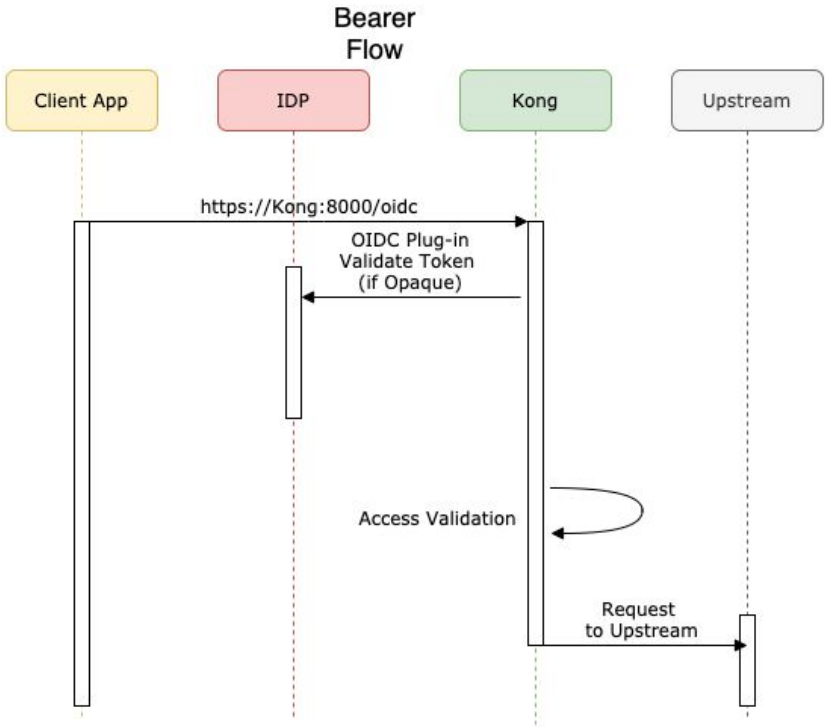
Token Authentication (also called Bearer authentication) is an HTTP authentication scheme that involves security tokens called bearer tokens.

In this flow, the client or application has a token.

Kong can validate the token using signature when it is JWT, or validate with Introspect endpoint when it is an opaque token.

With the token validated, Kong can leverage claims in JWT tokens or claims in introspection results to drive access decisions such as consumer mapping, group mapping etc.

In this section, you will use the bearer token generated by the authentication server for the employee user to authenticate to the Service.



Task: Get a token and authenticate with it

Let's obtain a bearer token for the user from the IDP directly using credentials:

```
$ BEARER_TOKEN=$(http -f POST $KEYCLOAK_URL/auth/realms/kong/protocol/openid-connect/token \
                  grant_type=password \
                  client_id=kong \
                  client_secret=$CLIENT_SECRET \
                  username=employee \
                  password=test \
                  | jq -r .access_token)
```

Use the bearer token to authenticate to the Service:

```
$ http GET kongcluster:30000/oidc authorization:"Bearer $BEARER_TOKEN"
```

Since the bearer token is valid and associated with the employee user, the response is an `200 OK`

```
HTTP/1.1 200 OK
```

Task: Decoding the Bearer Token

You can easily decode the bearer token from the CLI as noted before or using http://jwt.io. Here is an explanation for the displayed fields:

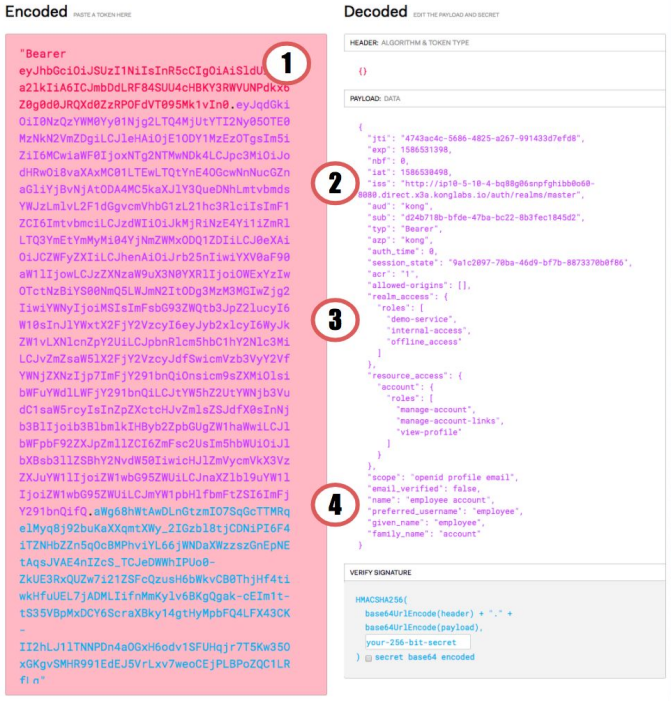
Bearer Token: This the encoded bearer token.

Issuer: This is the `config.issuer` you specified in during the initial Keycloak configurations.

Realm Access: This shows the Realm Access and the roles associated with the user.

Token Information: This shows the scope, email, name, etc. that is associated with this user. Notice the given name and family name that you specified in the Keycloak configurations.

As we move into the Authorization section, you will notice how we can use the claims and scopes in the token to drive authorization decisions.



Authorization

Authorization with OpenID Connect Overview

To this point we have focused on authentication for various application use cases

In this section, we will focus on authorization to control access to Services.

There are a number of ways to do this with Kong. There are a number of popular approaches to authorizing API access with Kong and the OpenID Connect plugin, including:

- Authorization with Consumer Mapping
- Authorization with Roles
- Authorization with Audience Required
- Authorization with Scopes Required

We will look at authorization with roles as well as Rate Limiting Different Consumers

Authorization with Consumer Mapping

Authorization with Consumer Mapping

One way to authorize access to services with Kong is to require external IDP users to map to a Kong consumer.

Kong Consumer objects are not necessarily singular users/consumers but they can also be constructs of a "Consumer package".

For example, if you have a Consumer named "gold", this could represent all the consumers that have a "gold" tier account with your organization.

Consumer objects let you control who can access your APIs, report on traffic using logging and allow applying additional restrictions like rate limits..

In this section, you'll authorize access to the employee user via consumer mapping.

Task: Configure a consumer & modify OIDC plugin to require preferred_username

Configure a consumer with `username=employee` in Kong using the CLI.

```
$ kubectl apply -f ./oidc-consumer.yaml
```

Now patch OIDC plugin to require a consumer with the `preferred_username` claim

```
$ cp httpbin-oidc-plugin.yaml httpbin-oidc-plugin-claim.yaml
$ cat << EOF >> httpbin-oidc-plugin-claim.yaml
  consumer_claim:
  - preferred_username
EOF
$ kubectl apply -f ./httpbin-oidc-plugin-claim.yaml
```

```
namespace/httpbin-demo unchanged
kongplugin.configuration.konghq.com/httpbin-oidc configured
```

Task: Verify authorization works for a user mapped to a Kong consumer

Let's verify a user whose name matches a Kong consumer can successfully authenticate and is authorized. Use the employee consumer to verify.

```
$ http GET kongcluster:30000/oidc -a employee:test
```

The employee user can still proxy but now the headers `x-consumer-ID` and `x-consumer-username` are added

Also, if you inspect the Bearer token and paste it in the https://jwt.io/ website, you will see that the claim `preferred_username=employee`.

```
HTTP/1.1 200 OK
...
        "X-Consumer-Id": "41d7c210-5f8e-4eca-9285-20133ef36ea3",
        "X-Consumer-Username": "employee",
...
```

Task: Verify authorization is forbidden for a user not mapped to a consumer

Let's verify a user whose name does not match a Kong consumer is denied access. Use the partner consumer to verify.

```
$ http GET kongcluster:30000/oidc -a partner:test
```

This user is forbidden because the partner user is not mapped to the consumers in Kong.

```
HTTP/1.1 403 Forbidden
```

Task: Add & Verify Rate Limiting

Now that a Kong Consumer for the user employee exists, apply a Rate Limiting policy to the Consumer

```
$ kubectl apply -f ./oidc-consumer-rate-limiting.yaml
namespace/httpbin-demo unchanged
kongplugin.configuration.konghq.com/employee-rate-limiting created
kongconsumer.configuration.konghq.com/employee configured
```

Execute the same request 5 times in less than a minute to see rate limiting plugin in action:

```
$ for ((i=1;i<=5;i++)); do http -h GET kongcluster:30000/oidc -a employee:test; done
```

```
HTTP/1.1 429 Too Many Requests
...
X-RateLimit-Limit-Minute: 3
X-RateLimit-Remaining-Minute: 0
...
```

The last 2 requests trigger the rate limit as seen above.

Task: Cleanup

Disable this consumer mapping and remove the rate limiting. This is important to progress with the next section.

```
$ kubectl apply -f ./httpbin-oidc-plugin.yaml
```

```
namespace/httpbin-demo unchanged
kongplugin.configuration.konghq.com/httpbin-oidc configured
```

```
$ kubectl delete kongplugin employee-rate-limiting -n httpbin-demo
kongplugin.configuration.konghq.com "employee-rate-limiting" deleted
$ kubectl apply -f ./oidc-consumer.yaml
namespace/httpbin-demo unchanged
kongconsumer.configuration.konghq.com/employee configured
```

Authorization with Roles

Authorization with Roles

In this section we cover how to drive access to services via Roles from the IDP using the ACL plugin in conjunction with the OpenID Connect plugin.

This scenario is useful to drive user access based on user roles managed in the IDP with no need to define or manage identities in Kong.

Another scenario this is useful for is when different types of users authenticate differently, for example external users authenticate via Key-Auth while internal users authenticate via LDAP.

To accomplish this, we will setup the OpenID Connect plugin to read the user's roles as defined in the IDP, then configure the ACL plug-in to require users to have specific roles before they can access the services.

Task: Modify the OIDC plugin to search for user roles in a claim

Shown is a snippet of the JWT token

We will patch the OpenID Connect plugin to search the array `realm_access→roles` for user roles.


```
  "jti": "d28100f2-e7d2-4cd1-a985-5c75b40e560b",
  "exp": 1586957698,
  "nbf": 0,
  "iat": 1586956798,
  "iss": "http://ip10-0-25-4-bqbfgnd51hr0hcprqds0-
8080.direct.x4a.konglabs.io/auth/realms/master",
  "aud": "kong",
  "sub": "d24b718b-bfde-47ba-bc22-8b3fec1845d2",
  "typ": "Bearer",
  "azp": "kong",
  "auth_time": 0,
  "session_state": "f78d2a4e-1634-44a7-99c9-51f68a8ef0ab",
  "acr": "1",
  "allowed-origins": [],
  "realm_access": {
    "roles": [
      "demo-service",
      "internal-access",
      "offline_access"
    ]
  },
```

Task: Modify the OIDC plugin to search for user roles in a claim

Run the following commands to patch the OpenID Connect plugin

```
$ cp ./httpbin-oidc-plugin.yaml ./httpbin-oidc-plugin-realm.yaml
$ cat << EOF >> ./httpbin-oidc-plugin-realm.yaml
  authenticated_groups_claim:
  - realm_access
  - roles
EOF
$ kubectl apply -f httpbin-oidc-plugin-realm.yaml
namespace/httpbin-demo unchanged
kongplugin.configuration.konghq.com/httpbin-oidc configured
```

Task: Configure the ACL plugin and whitelist access to users with the admins role

```
$ kubectl apply -f ./httpbin-ingress-oidc-acl.yaml
namespace/httpbin-demo unchanged
kongplugin.configuration.konghq.com/httpbin-acl created
service/oidc-service configured
ingress.networking.k8s.io/oidc-route configured
```

The ACL plug-in is now applied and requires users to be a member of the internal role to access the service.

```
$ http GET kongcluster:30000/oidc -a employee:test
```

```
HTTP/1.1 403 Forbidden
```

This user is forbidden because the user is not a member of the internal role.

Task: Modify the ACL plugin to require users being members of the role demo-service to access the service

```
$ cat << EOF | kubectl apply -f -
apiVersion: configuration.konghq.com/v1
kind: KongPlugin
metadata:
  name: httpbin-acl
  namespace: httpbin-demo
plugin: acl
config:
  allow:
  - admins
  - demo-service
EOF
kongplugin.configuration.konghq.com/httpbin-acl configured
```

Task: Modify the ACL plugin to require users being members of the role demo-service to access the service

Access the route with user employee

```
$ http GET kongcluster:30000/oidc -a employee:test
```

```
HTTP/1.1 200 OK
```

Notice the user is now able to access the service. This method allows the identity management process to drive which services secured by Kong the user has access to.

Task: Cleanup

We need to clean up this set of plugins. This is important to progress with the next section.

```
$ kubectl delete kongplugin httpbin-oidc -n httpbin-demo
kongplugin.configuration.konghq.com "httpbin-oidc" deleted
$ kubectl apply -f ./httpbin-oidc-plugin.yaml
namespace/httpbin-demo unchanged
kongplugin.configuration.konghq.com/httpbin-oidc created
```

Authorization and Rate Limiting Different Consumers

Authorization and Rate Limiting Different Consumers

In a previous section, we covered how Consumers can be authorized and rate limited.

In this section, we will expand on that example further to implement a scenario where internal users, say employees, are rate limited at 5 requests per minute but external users, say partners, are rate limited at 1000 requests per minute.

This scenario is useful when you have a service accessible by both free and paid users.

Both user types are accessing the same service but we want to provide our paid subscribers preferential rate limits.

Another use-case scenario is when multiple subscriptions levels exist for a service, say Silver, Gold and Platinum.

We can enforce different rate limits for different subscription levels, for example, providing better rate limits to our higher paying consumers.

Task: Modify & verify the plugin to require a scope of admins

To enable this scenario we will configure the OpenID Connect plugin to map the `preferred_username` claim to a Kong Consumer but optionally require the Kong Consumer to exist.

Essentially, this is saying, if a Kong Consumer doesn't exist, allow the user to access but allow Kong to track who the user is.

```
$ cp ./httpbin-oidc-plugin.yaml ./httpbin-oidc-plugin-preferred.yaml
$ cat << EOF >> ./httpbin-oidc-plugin-preferred.yaml
  consumer_claim:
  - preferred_username
  consumer_optional: true
EOF
$ kubectl apply -f ./httpbin-oidc-plugin-preferred.yaml
namespace/httpbin-demo unchanged
kongplugin.configuration.konghq.com/httpbin-oidc configured
```

Task: Configure Rate Limiting Plugins

To implement our desired scenario, we're going to apply a rate limit to our route of 5 requests per minute and apply a rate limit of 1000 requests per minute to our consumer. Another way to look at it is, any user without a Kong Consumer who is rate limited differently, will be rate limited at 5 requests per minute.

```
$ kubectl apply -f ./httpbin-ingress-oidc-rate-limiting.yaml
namespace/httpbin-demo unchanged
kongplugin.configuration.konghq.com/httpbin-rate-limiting created
service/oidc-service configured
ingress.networking.k8s.io/oidc-route configured
```

```
$ sed -i "s/minute: 3/minute: 1000/g" ./oidc-consumer-rate-limiting.yaml
$ kubectl apply -f ./oidc-consumer-rate-limiting.yaml
namespace/httpbin-demo unchanged
kongplugin.configuration.konghq.com/employee-rate-limiting created
kongconsumer.configuration.konghq.com/employee configured
```

Task: Verify Rate Limits

Verify the rate limit of the partner. The partner user is allowed to access the Service but is rate limited at 5 requests per minute.

```
$ for ((i=1;i<=6;i++)); do http -h GET kongcluster:30000/oidc -a partner:test; done
```

```
HTTP/1.1 429 Too Many Requests
...
Date: Thu, 30 Jun 2022 19:53:53 GMT
RateLimit-Limit: 5
RateLimit-Remaining: 0
```

```
$ for ((i=1;i<=12;i++)); do http GET kongcluster:30000/oidc -a employee:test; done
```

```
HTTP/1.1 200 OK
...
X-Kong-Upstream-Latency: 21
X-RateLimit-Limit-Minute: 1000
X-RateLimit-Remaining-Minute: 988
```

Summary

In this lesson we've looked at

- What is OIDC?
- Deploying Keycloak as OIDC provider
- Deploying the OIDC Plugin
- Authenticate with Password Grant
- Authenticate with Bearer Token Grant
- Authorization with Consumer Mapping
- Authorization with Roles
- Authorization and Rate Limiting Different Consumers

Questions?

What's next?

In the next section we will look at monitoring your Kong Gateway.

Thank You