Kong Commands

Create a docker network

```
docker create network kong-net
```

Install Kong in DB-less mode

Docker run command for Kong DB-less

```
docker run -d --name kong-dbless \
  --network=kong-net \
  -v "$(pwd):/kong/declarative/" \
  -e "KONG_DATABASE=off" \
  -e "KONG_PROXY_ACCESS_LOG=/dev/stdout" \
  -e "KONG_ADMIN_ACCESS_LOG=/dev/stdout" \
  -e "KONG_PROXY_ERROR_LOG=/dev/stderr" \
  -e "KONG_ADMIN_ERROR_LOG=/dev/stderr" \
  -e "KONG_ADMIN_LISTEN=0.0.0.0:8001, 0.0.0.0:8444 ssl" \
  -p 8000:8000 \
  -p 8443:8443 \
  -p 127.0.0.1:8001:8001 \
  -p 127.0.0.1:8444:8444 \
  kong:latest
```

Test the Admin API base URL

http://localhost:8001/
https://localhost:8444/

Kill and remove the Container

```
docker ps
docker kill kong-dbless
docker rm kong-dbless
```

Install Kong with DB

1. Install Postgres DB

```
docker run -d --name kong-database --network=kong-net -p 5432:5432 -e
"POSTGRES_USER=kong" -e "POSTGRES_DB=kong" -e "POSTGRES_PASSWORD=kong"
postgres:9.6
```

```
docker ps
```

```
docker images
```

2. Prepare the DB for kong

```
docker run --rm --network=kong-net -e "KONG_DATABASE=postgres" -e
"KONG_PG_HOST=kong-database" -e "KONG_PG_USER=kong" -e
"KONG_PG_PASSWORD=kong" kong:latest kong migrations bootstrap
```

3. Start the Kong API Gateway

```
docker run -d --name kong --network=kong-net -e "KONG_DATABASE=postgres" -e
"KONG_PG_HOST=kong-database" -e "KONG_PG_USER=kong" -e
"KONG_PG_PASSWORD=kong" -e "KONG_PROXY_ACCESS_LOG=/dev/stdout" -e
"KONG_ADMIN_ACCESS_LOG=/dev/stdout" -e "KONG_PROXY_ERROR_LOG=/dev/stderr"
-e "KONG_ADMIN_ERROR_LOG=/dev/stderr" -e "KONG_ADMIN_LISTEN=0.0.0.0:8001,
0.0.0.0:8444 ssl" -p 8000:8000 -p 8443:8443 -p 127.0.0.1:8001:8001 -p 127.0.0.1:8444:8444
kong:latest
```

4. Test the Kong Admin API url

```
curl http://localhost:8001/
```

Kong default ports

By default, Kong listens on the following ports:

Kong api gateway proxy ports
8000: listens for incoming HTTP traffic from your clients, and forwards it to your upstream services.
8443: listens for incoming HTTPS traffic. This port has a similar behavior to 8000, except that it

Kong Admin API ports (Used for configuring and updating conf configuration of services, routes etc)
8001: Admin API listens for calls from the command line over HTTP.
expects HTTPS traffic only. This port can be disabled via the configuration file.
8444: Admin API listens for HTTPS traffic.

4.Lifecycle commands

Note: If you are using Docker, exec into the Docker container to use these commands.

```
docker exec -it 1f24c8e1a39a bash
```

Stop Kong Gateway using the stop command:

```
kong stop
```
Reload Kong Gateway using the reload command:

```
kong reload
```
Start Kong Gateway using the start command:

```
kong start
```

Configuring a Service

1. Add Service

```
curl -i -X POST --url http://localhost:8001/services/ --data name=example-service --data
url=http://mockbin.org
```

Verify

```
curl  http://localhost:8001/services/example-service
```

2. Add Route

```
curl -i -X POST http://localhost:8001/services/example-service/routes \
  --data 'paths[]=/mock' \
  --data name=mocking
```

Verify

```
curl http://localhost:8001/routes
```

3. Verify the api gateway endpoint using kong proxy port 8000

```
curl http://localhost:8000/mock/request
```

Day2 : Exercise 1

The backend Microservice end point is https://httpbin.org

We want an endpoint /httpmock configured in API gateway so that the request gets forwarded to the above backend Microservice.

End result https://localhost:8000/httpmock/get
Should return a valid json response from actual microservice

1. We have to create a new service pointing to https:/httpbin.org using Admin API
2. We need to create a route with above created service /httpmock using Admin API
3. Verify the https://localhost:8000/httpmock/get

4. Configure Plugin

1. Configure key-auth plugin for example-service

```
curl -i -X POST --url http://localhost:8001/services/example-service/plugins/ --data
name=key-auth
```

2. Verify

```
curl http://localhost:8000/mock/request
```

Should get error as No API key found

5. Adding Consumer

1. Create a consumer through Admin API

```
curl -i -X POST --url http://localhost:8001/consumers/ --data "username=consumer1"
```

2. Verify

```
curl http://localhost:8001/consumers
```

3. Provision key credentials for your Consumer

```
curl -i -X POST  --url http://localhost:8001/consumers/consumer1/key-auth/ --data
"key=myconsumersecurekey123"
```

4. Verify that your Consumer credentials are valid by accessing the /mock endpoint using the above key

```
curl -i http://localhost:8000/mock/request --header "apikey: myconsumersecurekey123"
```

Should get valid response

```
curl -i http://localhost:8000/mock/request --header "apikey: wrongkey"
```

Should get 401 unauthorized

Day2 : Exercise 2

Enable key-auth plugin for the previously created service for this ms https://httpbin.org

And create a new consumer and assign a valid key to him

Access the /httpmock endpoint using key authentication standard by passing above create key

http://locahost:8000/httpsmock/get

Task: Verify all Items with the Admin API

```
curl http://localhost:8001/services
curl http://localhost:8001/routes
curl http://localhost:8001/consumers
curl http://localhost :8001/plugins
```

How to DELETE a route

```
curl -X DELETE http://localhost:8001/services/example-service/routes/<id-of-the-route>
```

Setting Up Request Transformer Plugin

1. Configure the service to use request transformer plugin

```
curl -X POST http://localhost:8001/services/example-service/plugins \
--data "name=request-transformer" \
--data "config.add.headers=x-my-header:myvalue"
```

2. Verify

```
curl -i http://localhost:8000/mock/request --header "apikey:myconsumersecurekey123"
```

The header should be seen in the response body as the mockbin ms will return back the same header values

Setting Up Rate Limiting Plugin

1. Configure Rate limiting plugin at global level

```
curl  -X POST http://localhost:8001/plugins \
--data name=rate-limiting \
--data config.minute=5 \
--data config.policy=local
```

2. Verify

```
curl -i http://localhost:8000/mock/request --header ‘apikey: myconsumerseucrekey123”
```

If you want to validate Client IP based remove the key-auth plugin and make the below request

```
curl -i http://localhost:8000/mock/request
```

In both cases we should get 429 error after 5 request in a given minute.

```
{
"message": "API rate limit exceeded"
}
```

Setting Up Proxy Cache Plugin

1. Configure the Proxy Cache plugin at global level

```
curl -X POST http://localhost:8001/plugins \
--data name=proxy-cache \
--data config.content_type="application/json; charset=utf-8" \
--data config.cache_ttl=30 \
--data config.strategy=memory
```

2. Validating Proxy Caching

```
curl http://localhost:8000/mock/request
```

In particular, pay close attention to the values of
X-Cache-Status
X-Kong-Upstream-Latency
X-Kong-Proxy-Latency

No Cache invoked actual backend api
X-Cache-Status : Bypass
X-Kong-Upstream-Latency: >200ms
X-Kong-Proxy-Latency : 10ms

Cached scenario
X-Cache-Status : Hit
X-Kong-Upstream-Latency: 0s
X-Kong-Proxy-Latency : less than previous value

If you want to clear the cache manually you can do it using Admin API

```
curl -X DELETE http://localhost:8001/proxy-cache
```

Configure Upstream Services

1. Create Upstream

```
curl -X POST http://localhost:8001/upstreams \
--data name=example-upstream
```

2. We need to update the example-service with above created upstream

```
curl -X PATCH http://localhost:8001/services/example-service \
--data host='example-upstream'
```

3. We need to create targets for the given upstream

```
curl -X POST http://localhost:8001/upstreams/example-upstream/targets \
  --data target='mockbin.org:443'

curl -X POST http://localhost:8001/upstreams/example-upstream/targets \
  --data target='httpbin.org:443'
```

4. Verify the actual endpoint

```
curl http://localhost:8000/mock
```

It should load balance between the mockbin and httpbin webpage response

Day 3: Exercise 1

Yesterday as part of Exercise a route /httpmock and a service was created and key-auth plugin was enable now the client has request to change the authentication to Basic-authentication
So, the you have to remove the key-auth authentication on service and enable Basic-authentication on the route.

Also the client has requested to send back these 2 below header values as part of RESPONSE HEADER

my-custom-header1:value1
my-custom-header2:value2

Demo:

1. Send valid credentials and get the response
2. Response should have the 2 headers
3. Send invalid credentials and need to get 401 unauthorized error

Disable the plugin

find plugin id and copy

```
curl -X http://<admin-hostname>:8001/routes/mocking/plugins/
```

Disable the plugin

```
curl -X PATCH http://<admin-hostname>:8001/routes/mocking/plugins/{<plugin-id>} \
  --data enabled=false
```

Custom Plugins

1. Check if docker and docker-compose is available

by running them
If docker and docker-compose is not available it needs to be installed

```
sudo snap install docker
```

or

```
sudo apt install docker-compose
```

If it asks for unix password(rps) : rps

2. Create a folder ”labuser” and cd to it

```
mkdir labuser
cd labuser
```

3. Clone the kong-pongo repository

git clone https://github.com/Kong/kong-pongo.git

4. Setup the pongo

```
export PATH=$PATH:~/.local/bin
mkdir -p ~/.local/bin
ln -s $(realpath kong-pongo/pongo.sh) ~/.local/bin/pongo
```

5. Download the custom plugin template code from kong repository

```
git clone https://github.com/Kong/kong-plugin.git
cd kong-plugin
```

6. Test pongo command

```
pongo --help
KONG_VERSION=3.0.0.0 pongo run
```

Useful Pongo Commands

click on the command to see the description :

`pongo build`
This builds the Kong test image. You will not use this explicitly in this workshop but pongo run and related commands will run this command automatically under the hood.

`pongo up`
This starts the required dependency containers for testing. This will pick up the configuration from .pongo/pongorc, environment variables, and/or the standard output.

This command will run automatically when pongo run is executed.

pongo run
This runs spec files and applies busted options set in .busted file. This will automatically run several commands as needed, such as pongo build , pongo up , for example.

Busted is a unit testing framework with a focus on being easy to use.

pongo down
This will stop the environment and remove all dependency containers (eg. postgres, etc.). This also means that any state in those containers will be lost. Also note that this command will NOT be run after the pongo run command has completed

pongo shell
This will attach a shell to a Kong test container (started from an image created by the pongo build command) and enbales you to run commands inside of it. This is a good tool when you want to test your plugin code manually.

Note: the container will be destroyed when you exit the shell.

pongo lint
This will run the luacheck linter on your plugin and test code. Since Lua is a dynamic language it is very easy to make typos that go unnoticed and create hard to find bugs. We strongly recommend to run this command while testing and add it to your CI setup.

Structure

A Kong plugin in its simplest form consists of a directory structure with at least two lua files:

```
simple-plugin (directory)

├── handler.lua (file)

└── schema.lua (file)
```

schema.lua defines the schema of the plugin like configuration options to assure users can only enter valid configuration values when configuring the plugin.

handler.lua determines which phase the plugin will be running during the network interactions of Kong with the client upstream service.

The plugins interface allows you to override any of the following methods in your handler.lua file to implement custom logic at various points of the execution life-cycle of Kong. Custom plugins can be written to execute on a number of methods, which are shown in the table below. However, most custom plugins will be written to execute logic in the access() handler (on the request) or header_filter() handler (on the response).

To help users get started with developing a custom plugin, Kong makes available a template plugin found on Github here to start developing with. The template plugin sets headers and logs messages at various phases of the request/response lifecycle.

Modify the handler.Lua file to make the integration test case fail

Modify the request flow and response flow to return a different value

Then run
KONG_VERSION=3.0.0.0 pongo lint

Check if any typos or syntax errors are there

KONG_VERSION=3.0.0.0 pongo run

This will setup the kong environment and run integration test

The test cases will fail

Now fix the spec files pointed in the error to match the asserts and re run pongo run

KONG_VERSION=3.0.0.0 pongo run

Deploy the customplugin to kong and test manually

```
KONG_VERSION=3.0.0.0 pongo shell
```

If migration scripts are there

```
kong migration bootstrap
```

```
kong start
```

```
http localhost:8001
```

Response

Let's add a service, a route and your plugin to the Kong configuration so that we can test the plugin's behavior manually.

Add service

Add a service to the Kong with the command shown below. The mock service in http://httpbin.org will echo the request along with providing the response.

```
http POST :8001/services/ name=mock-service url=http://httpbin.org
```

Add a route

Add a route as well by the command.

```
http POST :8001/services/mock-service/routes name=mock-route hosts:='["httpbin.org"]'
paths:='["/mock"]'
```

Add your plugin

Add your plugin to the service by the command.

```
http POST :8001/services/mock-service/plugins name=myplugin
```

Response

Verify your plugin is working

Let's verify the plugin is working as you expect by the command.

```
http :8000/mock/anything Host:httpbin.org
```

Check the response header section and notice the modified header
“Bye-World:” this is response by custom plugin”.
Also checkout the echoed request and see the request header set by the plugin
"Hello-World": "this is on a request by custom plugin".

Enabling Monitoring in kong (OSS) with prometheus plugin and prometheus server

Prerequisite is to have the kong up and running in the docker environment

1. Enable the prometheus plugin

```
curl -s -X POST http://localhost:8001/plugins/ \
 --data "name=prometheus"
```

2. Create a prometheus config yml file in local folder

```
prometheus.yml
scrape_configs:
  - job_name: 'kong'
    scrape_interval: 5s
    static_configs:
      - targets: ['kong:8001']
```

3. Run the Prometheus server in same docker network as kong

```
docker run -d --name kong-prometheus \
   --network=kong-net -p 9090:9090 \
   -v $(pwd)/prometheus.yml:/etc/prometheus/prometheus.yml \
   prom/prometheus:latest
```

4. Kong metrics that will be shared and available in prometheus can be viewed using kong admin url http://localhost:8081/metrics

5. Access the prometheus server using below link

http://localhost:9090

Access http://localhost:9090/graph and in the search tab search for kong_node_info you will see the information about kong server version

6. You can also access it using command line with below command

```
curl -s http://localhost:9090/api/v1/query?query=kong_node_info
```