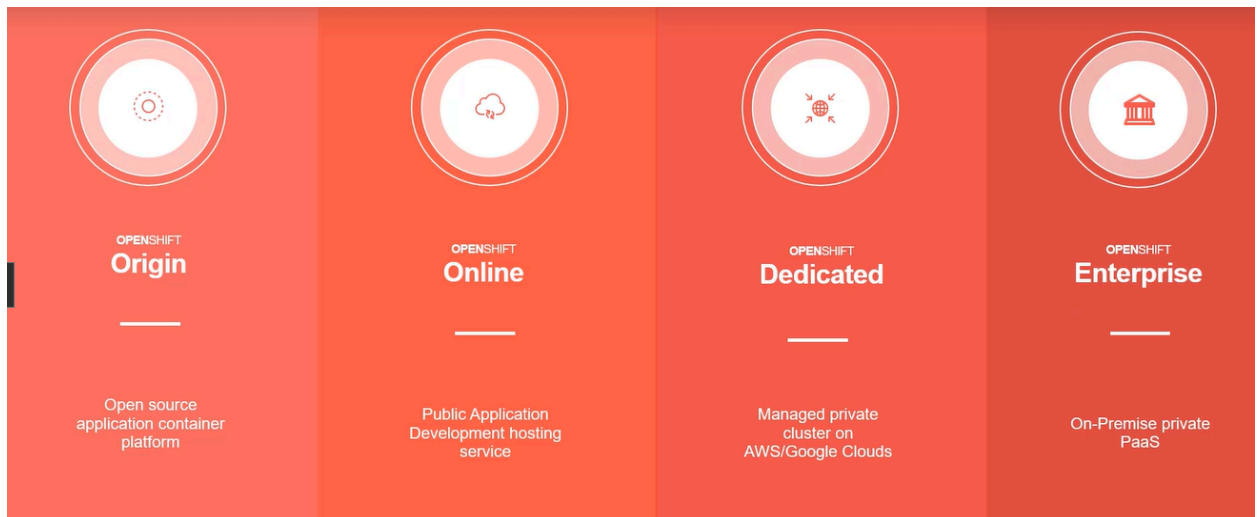


#Basics

Openshift is redhat's open source container application platform for developing and hosting enterprise grade applications

What's Difference between them?



1.

Openshift origin is based on top of docker containers and the kubernetes cluster manager, with added developer and operational centric tools that enable rapid application development, deployment and lifecycle management.

Kubernetes is a container orchestration system i.e. k8 is a software that orchestrates container

Container (Real life analogy):

It's like giving all our stuff to others . Without a bucket, giving them all at once is a tedious task. But if we put all our stuff in a bucket, then we can give it easily to others. Docker helps us do this. Container is like a venv in python (isolated environments).

We put the app in a container and the container runs on the host machine.

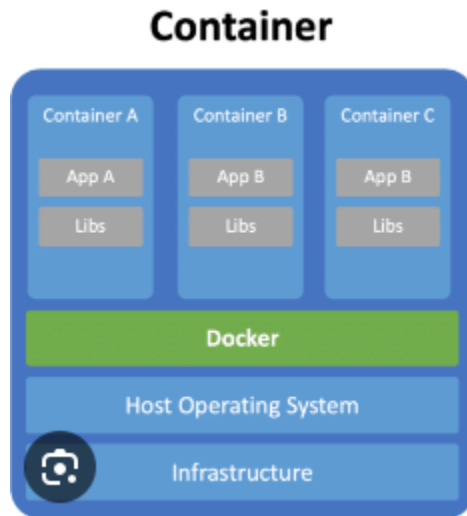
Instead of just having 1 container for the entire application, we can have different containers that do different jobs , but they all will work together as a single application. This is one way you can achieve microservices in software development. Example: container 1 for DB, container 2 for web server etc.

If your db requires linux and web server requires windows and db requires packages, libraries, dependencies with 1.8 and web server requires the same thing with different

versions . then it's an issue. If some new guy comes, setting up everything for him takes a longer time. Hence use docker

If you to run instance of ansible on docker container just do `docker run ansible` , likewise `docker run node js`

If u are running multiple api on same host and if one api cause some issue due to memory consumption etc, then all api will face issue. So container is needed



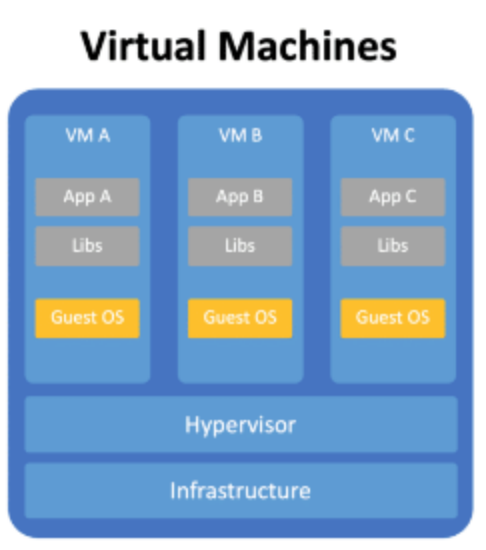
Ubuntu, fedora etc are different OS . but they have same OS kernel i.e. linux (the thing that interacts with hardware). OS is a combination of os-kernel and software

os-kernel + software 1 = ubuntu

os-kernel + software 2 = fedora

Os-kernel + software 3 = centos

Docker containers share the underlying kernel. According to the image , the host operating system = os software= ubuntu. If HOS = ubuntu, docker can run any flavor of OS on top of it as long as they are based on the same kernel. Here the kernel is linux. So it can run fedora,centos too. But if we want to run windows os, then it's not possible. For that we would have to run docker on a windows server. Docker doesn't act like a hypervisor (hypervisor allows different os on same physical servers with different users as guests).



Each vm has its own os unlike docker where all containers share same kernel. So utilization memory is higher in vm's. Booting up (starting) vm take more time than containers

Difference between VM and container

1. VM has its own os , container share underlying kernel
2. Boot up time is faster in container than VM
3. Containers are used for single process / tasks. VM is not like that
4. Due to all above containers are lightweight and VM's are not
5. And due to that it allows you to implement microservice architecture

Kubernetes

K8 is used to horizontally scale your software based on components that need scaling.

Upgrading your server by a bigger ram is vertical scaling .

Having multiple servers is horizontal scaling .

(Trick : you don't keep computer on computer, so u dont do vertical)

After packaging your application into your docker container, you want to productionize it. How do you do it? What if your application relies on other containers such as databases or messaging service or other back-end service? How do you scale up and down?

You need a platform that needs to orchestrate connectivity between different containers and automatically scale up/down based on load. The whole process of automatically deploying and managing containers is known as container orchestration. Kubernetes is that tool.

Docker swarm (docker), kubernetes(google), mesos (Apache).

Docker terminologies

Image is a template . Images are read-only templates that contain a runtime environment that includes application libraries and applications. Images are used to create containers. Images can be created, updated, or downloaded for immediate consumption.

Container is the running instance of the image. Containers are segregated user-space environments for running applications isolated from other applications sharing the same host OS.

Registries

Registries store images for public or private use. The well-known public registry is Docker Hub, and it stores multiple images developed by the community, but private registries can be created to support internal image development under a company's discretion.

Openshift architecture

Openshift is a platform that allows you to run containerized applications and workloads that are powered by kubernetes underneath it.

Containers are formed from images. Where do images come from ? you can create them or pull these images/ configure openshift to pull these images from a public docker repository like dockerhub or use openshift container registry(OCR) that comes builtin with openshift origin.

A collection of one or more containers from a pod and multiple pods form a deployment. We can then use services to expose the deployment to other applications or to the external world.

Docker basics

A Docker image is like a blueprint for a container. It contains everything a container needs to run, such as the operating system, application code, and other dependencies. Think of it as a layered cake (os-kernel->os-software>application>library>dependencies>)

Layers in Docker Images:

- Immutable layers: Each layer in a Docker image is unchangeable. Once a layer is created, it cannot be modified.
- New layers are added: When you modify a container, Docker doesn't change the existing layers. Instead, it adds a new layer on top to record the changes.

Now, let's understand how to create Docker images in two ways:

1. Using a Running Container

Steps:

1. Start from an existing image: You launch a container from an existing image, for example, a base image like Ubuntu or a pre-built image like Node.js
2. Make changes in the running container: You can install software or modify files. For example, you might install a package inside the container.
3. Commit the changes: After making the changes, you can save them into a new image using Docker's commit command. The process of creating creates new image.

2. Using a Dockerfile (Recommended Approach)

Steps:

1. Create a Dockerfile:
2. Build the image:
3. Run the image:

So Docker uses a client-server architecture, and the client will run our command-line tool, which is known as docker, so all of our requests that we're going to make are running on the client server.

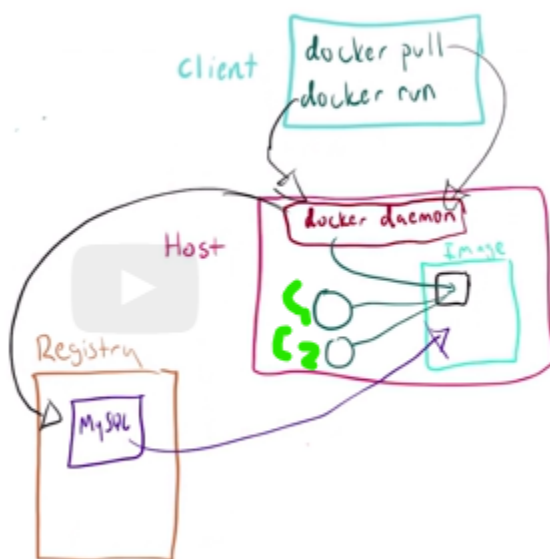
And then we'll also have another server, which is running the Docker service, and that has a daemon on it that's responsible for all of the building and running and downloading images.

essentially, we have these two separate servers (C n S). But they can actually be running on the same host. (what we do during development)

Docker daemon responsible for responding to clients.

When the client runs **docker pull mysql**, the request goes to host/daemon. Daemon says I need an image of mysql. So it'll first look at the local repository then if it doesn't have it, it'll go to a registry like docker hub and get it and drop it into our image repository in host. So that it does not go to the registry next time.

When the client runs **docker run mysql**, the request goes to host/daemon and says i want to run a container (mysql). Daemon checks for mysql images and creates a container based on that image. We can do it n time to create n containers. They are created on host



Container isn't new, but it makes things easy because it uses things already available in the Linux kernel like namespaces, control groups, and SELinux.

Namespaces are generally used to isolate processes and protect system resources. Each container runs in its own namespace, which means it has its own isolated view of system resources like processes, network interfaces, and file systems. Think of it as each container having its own mini-operating system environment while sharing the same underlying kernel with other containers. So unique IP

address(ipconfig),ports, hostname,root accounts (but as non privileged user on host),mnt namespace for each container.

Example of MNT Namespace in Containers:

When you create a Docker container, data stored inside a container is lost when the container is deleted, but volumes allow you to store data outside the container's filesystem, ensuring it persists even after the container is removed or restarted. Each container can mount or unmount file systems without affecting the host or other containers, giving it a private file system environment.

Imagine you have two containers running on the same host:

Container A mounts a directory /mnt/dataA that holds application data for App A.

Container B mounts a different directory /mnt/dataB for App B.

Control groups, or cgroups, are used to limit, monitor, and prioritize resource usage (such as CPU, memory, disk I/O, and network) for a group of processes, like a container. By using control groups you can set,track and prioritize the usage of resources by individual containers.

Security-Enhanced Linux (SELinux) is a security mechanism that enforces mandatory access control (MAC) policies on processes, including containers. It controls what resources (**e.g., files, network ports**) a container can access, ensuring strict security boundaries. In the context of containers, SELinux ensures that even if a process inside a container is compromised, it cannot access the host system or other containers in unauthorized ways. Labels: Every file, directory, and process has an SELinux label. Containers also get specific SELinux labels when they are run. So you can set RBAC for individual containers

Namespaces is what allows each container to act as mini -os with its own ip address,port,hostname,mnt name space etc

Cgroups is what allows us to track containers cpu usage,

and SELinux is there to implement Rbac for what files, network, ports each container can access.

Kubernetes Basics:

Containers are extremely lightweight and they're intended to fail, and that's not actually a bug but rather a feature.

Kubernetes takes care of 3 things :

Orchestration : kubernetes takes care of maintaining communication between containers/pods. If the container falls down, then the next spawned up one does not have the same ip address. Ip addresses are constantly changing. Kubernetes solves this So using services enables us to have a persistent IP address so that the pods can communicate with each other.

Scheduling: Kubernetes takes care of scaling up and down containers. It can make sure that a minimum number of containers are up and running. Replication controllers are objects that are used in order to help scale out the application.

Isolation : Kubernetes will help maintain the integrity of our solution by making sure that one container's issue does not affect the other. If a container is gone, all data associated with it is gone . Persistent volumes are used for us to persist data. And a persistent volume claim is just an object that we use to say I would like to have access to this data and we can use that to have multiple containers on different hosts all sharing the same persistent volumes.

Kubernetes architecture:

Kubernetes cluster is a set of node servers that run containers and they are managed by one master server. This master server is

responsible for running kubectl commands. Each node server can run multiple containers

Like docker uses containers as working objects, kubernetes uses pods. Pods are one or more containers responsible for sharing IP addresses.

Docker installation

Docker installation can be done on:

1. Windows 10 pro
2. Windows 10 education
3. Windows 10 enterprise

64 bit cpu required

Virtualization should be enabled in bios

In task manager -> performance . Virtualization should be enabled

Docker and other virtualization software such as VM ware or virtual box cannot run on same host

Follow below pdf to install properly

■ Exercise-Installing-Docker-CE-on-Windows.pdf

<https://desktop.docker.com/win/stable/amd64/Docker%20Desktop%20Installer.exe>

When you install it will ask for installation of wsl 2 . install that. Docker Desktop for Windows uses Windows Subsystem for Linux 2 (WSL 2) to support Linux containers. WSL 2 is essentially a lightweight virtual machine (VM) that runs a Linux kernel on top of the Windows operating system. Docker commands are same irrespective of what os you host is running on . Run in administrator mode

Docker commands

1. echo FROM nginx:alpine > Dockerfile (to create docker file)
2. docker run -dit debian

- a. -d : detach , -i : interactive , t :terminal
- b. Run will create a container from image. You can type many time to create many containers .
- c. Detach allows the container to run in the background. Successful Run will give a container id. If it doesn't something went wrong. use contained id for communication with container . detach means the process runs in the background, and the container doesn't stay attached to your current terminal. If you use only -it. Then if you close terminal. Then container stops.
- d. Interactive allows stdin (standard input) to be open. This allows you to type commands into the container.
- e. -t is closed linked with -i. Provides a pseudo terminal. To interact with a shell you need a terminal. Bot i and t usage is way to tell docker that allow interacting through terminal

3. docker ps

- a. Will give you list of running containers.

```
C:\Windows\System32>docker ps
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS | NAMES |
|--------------|--------|---------|----------------|---------------|-------|------------------|
| ccbe3604e975 | debian | "bash" | 6 seconds ago | Up 5 seconds | | exciting_pasteur |
| e0fd721bb701 | debian | "bash" | 11 minutes ago | Up 11 minutes | | sad_pare |

- b. You can use some starting characters from container id or Names to interact with container
- c. Docker ps -a will give you list of all containers after you started docker at that time

4. docker stop eof // docker stop sad_pare

- a. Stops the container either using container id starting digits or names

5. Docker kill eof

- a. A. doesnt do graceful stop . be really careful , you can put it in such a state that it cannot be restarted

6. Docker rm eof

- a. Removes a container

7. docker run debian

- a. Doesn't give anything . docker ps will give nothing because we din't use detach. So it immediately starts a container and stops it . evidence can be seen in ui

8. `docker run -d -i -t debian` or `docker run -it -d debian` is equally valid.
 - a. If you want to see the full container id use `docker ps --no-trunc`
 - b. If you want to provide a custom name then use `docker run -dit --name=myname debian`
 - c. `--` means its optional
 - d. `-rm` tells to delete container once its stops
 - e.
9. Docker restart 345
 - a. Restarts container
10. `docker images`
 - a. Will give a list of images downloaded from registry and are present in docker host.
 - b. Tag will tell you the version of the image. If you don't specify the version. Default tag:latest is used. How ever you can provide your own tag. Its like git project version.
11. `docker inspect [container_id]`
 - a. Will give file/mounts , networks , ports details.
 - b. You can see the proof control groups idea here. Your pc ip config and containers are different
12. `Docker --help` , `docker images --help` , `docker image prune --help`
 - a. You can use help at any section to know what commands can be used there and what it does.
13. `Docker pull nginx`
 - a. Downloads nginx image
 - b. Use `docker images` command to verify if image has been downloaded
14. `Docker history nginx`
 - a. Gives an idea of how that image was created. Each line is a layer and corresponding command executed is shown
15. `Docker tag nginx:latest nginx:stable`
 - a. to change the name or tag to something else
 - b. `Docker tag imagename:current tag imagename: new tag name`

```
C:\Users\DELL>docker tag --help
```

```
Usage:  docker tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]
```

- c.
 - d. Something inside colon (:) means its optional, if you don't provide default ones are assumed
16. When you download an image from docker you can have a specific tag which will say the version of images you want to download. <https://hub.docker.com/> go here and see the tag name for that particular image.
17. You can click on any tag in above link and see corresponding docker file / code
18. Docker build -t mycustomnode .
- a. Build : builds the image
 - b. -t : allows you define custom tag name , here its mycustomnode.
 - c. (dot) . at the last line tells that there is docker file in current directory and use it to build image
 - d. Later you use docker run mycustomnode . to check if its working.
19. Docker rmi mycustomnode / docker rmi -f mycustomnode
- a. Removes that image / untags it . always refer to iamgename:tagname for correct deletion
 - b. Sometimes you will also get deleted . this means it removed data from disk
20. docker image prune
- a. If you run docker images and get <none> <none>. No name, no tag. It means it's a dangling image. They are Untagged and unused layers . You can use docker image prune to remove those.
21. docker system prune -a
- a. it will only keep images used by running container
 - b. Stop unused containers
 - c. All networks unused by at least one container
 - d. Will remove dangling images and all build cache
 - e. If you dont use -a it will remove dangling images and unused build cache.
22. Docker system df
- a. Stands for disk free. Gives details about disk space . to decide on if you want to delete any stuff

- b. Provides details total number of images, containers, local volumes, build caches, . How many of them are active, their sizes and how much of it is reclaimable.
 - c. If you want to uninstall docker and get disk space , delete `var/lib/docker` . do this after you get a back up
- 23. `Docker run -dit --restart=always --name=alwayson debian`
 - a. Restarts container with same name if its gets stopped or host server gets restarted
- 24. `Docker investigate alwayson | grep -A3 RestartPolicy`
 - a. Grep for regex
 - b. -A matching context . A3 three lines after matching content
 - c. RestartPolicy: matching pattern
 - d. Instead of always , you can use `ondash_failure` and unless stopped
- 25. `Docker logs containerid`
 - a. Gives list of activities performed related to container
 - b. `Docker logs -f 82` : follows and logs as any new activity is performed . lets say you do something involving container. That will be logged here.
 - c. `Docker logs -n =3 82` : displays last 3 lines in log
 - d. `Docker logs -t 82` : log with timestamps
- 26. `Docker run -dit -rm imagename`
 - a. -rm tells to delete container once its stops
- 27. `Docker run -dit -p=8081:80 --name=containername imagename`
 - a. Exposes port of your pc and routes it to port 80 of container
 - b. You can also export multiple ports for same container
 - c. `Docker run -dit -p=8081:8081 -p=8082:8082 --name=containername imagename`
- 28. `Docker exec -it container name /bin/sh` or `/bin/bash`
 - a. Allows you enter container and run cmd commands there
 - b. You run any commands like `ls,pwd` etc
 - c. Use `ctrl +d` to exit container or type `exit`
 - d. You can also enter terminal while creating a container. Ex:
`Docker run -it container name /bin/sh`
- 29. `apt update > apt install -y vim`

- a. First enter contain , then type apt update , then apt install -y vim
 - b. This is useful if you are using debian based containers and some commands are not available inside container . ex: vi , uptime etc
 - c. After executing A , type vim to verify
30. Docker top containername
- a. Will give list of process running inside container
31. To push a image to private hub
- a. docker login -u username
 - b. In password provide password or docker token
 - c. docker push username/imagename
 - d. Sometimes you will have to change tag

file commands:

- 1. FROM
 - a. Usually used in the first link. Refers to the base image.
- 2. LABEL
 - a. Description of image. You can write anything by writing it as a variable
 - b. Ex: LABEL maintainer="shiva"
- 3. COPY
 - a. copies a file from local file system to image to build a image you have specific in your code at build time (build time means when you run docker build)
- 4. ENV
 - a. Allows you declare Environment variables that can be used later on in your build
- 5. RUN
 - a. Used to run a shell command
 - b. Always use && to run multiple shell commands if in sequence , this prevents creating another layer just for single command

```

95      && rm -rf "$GNUPGHOME" \
96      && mkdir -p /opt \
97      && tar -xzf yarn-v$YARN_

```

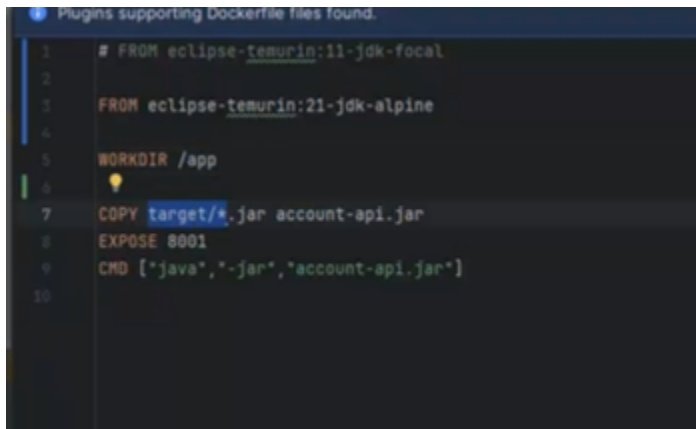
6. EXPOSE

- a. To open a port

7. CMD

- a. used to specify the default command to run when a Docker container is started from the image
- b. Usually commands are given in array ex: ["node", "daemon off"] etc
- c. daemon off says that keep running container in background , if you say on that it will start and immediately stop.

Exercises:



```

1  # FROM eclipse-temurin:11-jdk-focal
2
3  FROM eclipse-temurin:21-jdk-alpine
4
5  WORKDIR /app
6
7  COPY target/*.jar account-api.jar
8  EXPOSE 8001
9  CMD ["java", "-jar", "account-api.jar"]
10

```

Docker push username/imagename

clone repo

build image of accounts api

run that image

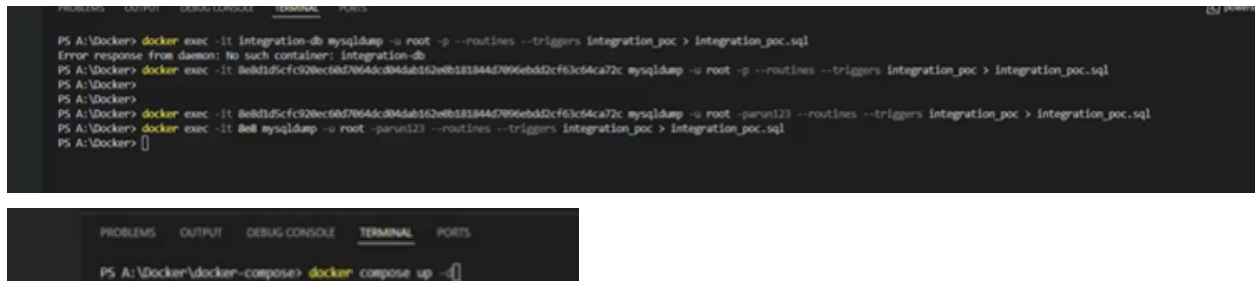
pull mysql image

run it

Docker management for docker compose

1. Install Dbeaver
2. Connect container (local host port) and dbeaver
3. Running stored procedure (dbscript.sql)
4. Load csv

#



```
PS A:\Docker> docker exec -it integration-db mysqldump -u root -p --routines --triggers integration_poc > integration_poc.sql
Error response from daemon: No such container: integration-db
PS A:\Docker> docker exec -it 8edd1d5cf920ec68d7064dc046ab162eb181844d7096ebdd2cf63c6ca72c mysqldump -u root -p --routines --triggers integration_poc > integration_poc.sql
PS A:\Docker>
PS A:\Docker> docker exec -it 8edd1d5cf920ec68d7064dc046ab162eb181844d7096ebdd2cf63c6ca72c mysqldump -u root -parun123 --routines --triggers integration_poc > integration_poc.sql
PS A:\Docker> docker exec -it 808 mysqldump -u root -parun123 --routines --triggers integration_poc > integration_poc.sql
PS A:\Docker>
PS A:\Docker\docker-compose> docker compose up -d
```

Procedures for docker swarm;

1. docker tag imagename:tag username/properimagename:latest
2. docker login -u username
3. In password provide password or docker token
4. docker push username/imagename
5. docker system prune -a
6. docker images
7. docker swarm init
8. Go to directory container docker compose yaml file (edit to to include sql source file)
9. docker stack deploy -c docker-compose.yaml integration_stack
10. docker stack services integration_stack
11. docker images
12. docker ps
13. docker swarm leave --force

To enter nginx container and reload it
docker exec -it f582 /bin/sh

nginx -s reload

Exit

```
curl http://account-api-container:8001/api/v1/accounts/112233/list
```

```
docker-compose restart <nginx_service_name>
```

```
docker network connect integration_stack_integration-net b9f
```

```
docker exec -it --user root <APISIX_CONTAINER_NAME> /bin/sh
```

```
curl http://172.31.224.1:8001/api/v1/accounts/496273/list
```

```
/etc/nginx/scripts/tokenValidation.js    inside container
```

```
docker-compose -p docker-apisix up -d
```

```
docker compose -p docker-apisix down
```

Docker registries

Docker hub is the default registry that docker client is configured to connect to.

Registry is a server that holds images. From here you can push pull images

A repository is in within a registry and it is a collection of images stored where each images is stored as tags . in other words repository is The path to a directory of images on a registry server

Local image is stored in the format : registry address / repository/imagename:tag

If your own private

You can have username/imagename:tag

There are different registries:

docker hub

Amazons ECR

QUAY (redhat)

gcr.io (google's container registry)

Apache apex built on top of nginx . opensource . code available.

<https://github.com/apache/apisix/blob/master/apisix/plugins/authz-keycloak.lua> they have a custom plugin whose code is written in lua . search for “ local function “ to see different written methods .

<https://www.keycloak.org/2021/12/apisix> . search for `client_secret_post` , this is used for introspection . may be out of date

<https://apisix.apache.org/blog/2021/12/10/integrate-keycloak-auth-in-apisix/>

<https://apisix.apache.org/docs/apisix/3.7/plugins/openid-connect/> can check this

<https://docs.api7.ai/hub/authz-keycloak/> use this as reference

Keywords uma

```
<dependency> <groupId>org.keycloak</groupId> <artifactId>keycloak-authz-client</artifactId>  
<version>8.0.1</version> </dependency>
```

Set authorization on for client

For /anything end point its says unaunthorized if lazy load is false and forbidden if true

<https://github.com/apache/apisix/issues/10708> focus on the end point

<https://github.com/apache/apisix/issues/10708>

<https://keycloak.discourse.group/t/keycloak-behind-api-gateway-invalid-bearer-token/10114>
current issue : well logged

<https://github.com/apache/apisix/issues/11033>

https://docs.redhat.com/en/documentation/red_hat_single_sign-on_continuous_delivery/4/html/authorization_services_guide/service_overview#service_authorization_api

So there is no point of authentication when you use it as bearer. Its only authorization . thats why you have select authorization on in client . this will add uma privilege and add/check service account for that

<https://github.com/keycloak/keycloak/issues/32326>

<https://groups.google.com/g/keycloak-user/c/6lyzjtKlwq4>

The learning curve :

Scopes : where we can activity they can perform. It can be any custom name: read ,write , access etc

This is available in authorization > scopes

Resource : refers to a collection of paths and scopes . You can define several end points and define what activity they can perform. Ex : resource name : account financial info : /statement , /transaction/* , /balance . scope : access , get etc .

Available in authorization resource

Client scopes: represents the set of permissions or claims that a client application can request when accessing resources. So for the same client id , you can request some or more permission when requesting a token and you will get a token with those permissions. You can create a policy associated with it as well.

Create a client scope and add it to client . it's just a name it doesn't make sense yet.

Policy > collection of client scopes. You can add multiple client scopes and say which one is definitely required for the keycloak to say if it should provide permission.

Go to policies > client scope based policy . this represents the set of permissions or claims that a client application can request when accessing resources. Select multiple client scopes if needed and give a name to the policy .

Create a permission : you can either do resource based or scope based

If resource based : provide a name to the permission . add resource (refer to that point) and policy (collection of client scopes)

If scope based : provide a name to the permission , select resource , select scopes (individual scopes) , policy collection of (client scopes).

When requesting a token , in permissions section , they will define resourcename#individualscopes in scope based permission evaluation .

It may do following checks

1. Whether the individual scopes is defined in resource
2. Whether the token has permission
 - a. It does this by checking client scope in token. Then it checks permission defined for that client id. There it checks if the token has client scopes defined in permission for it to provide access

To evaluate a permission “A” the token should contain client scope
“B”.

Httpbin-access : Client scope

DMZ
CMS

Client specific routes
Multiple clients per route

Create a client in keycloak via apex [try in service section]

- 1/. Create a upstream
2. Create a service
3. Create a consumer whose job is to create client id and secret
4. subscribe to the service /upstream#

Creating a consumer should create a client in keycloak

Ssl try

| URL | Method | Permission | Resource | Scope | Roles |
|-----------|--------|----------------|-------------|---------------|---------------------------|
| /accounts | POST | account-create | res:account | scopes:create | admin |
| /accounts | GET | account-view | res:account | scopes:view | admin, agent, super-admin |
| /bots | POST | bot-create | res:bot | scopes:create | admin |
| /bots | GET | bot-view | res:bot | scopes:view | admin, agent, super-admin |
| /reports | POST | report-create | res:report | scopes:create | super-admin |
| /reports | GET | report-view | res:report | scopes:view | admin, agent, super-admin |

In routes advanced : i can import openapi

In consumers . i can not add keycloak

In route : plugin config : i can do orchestration

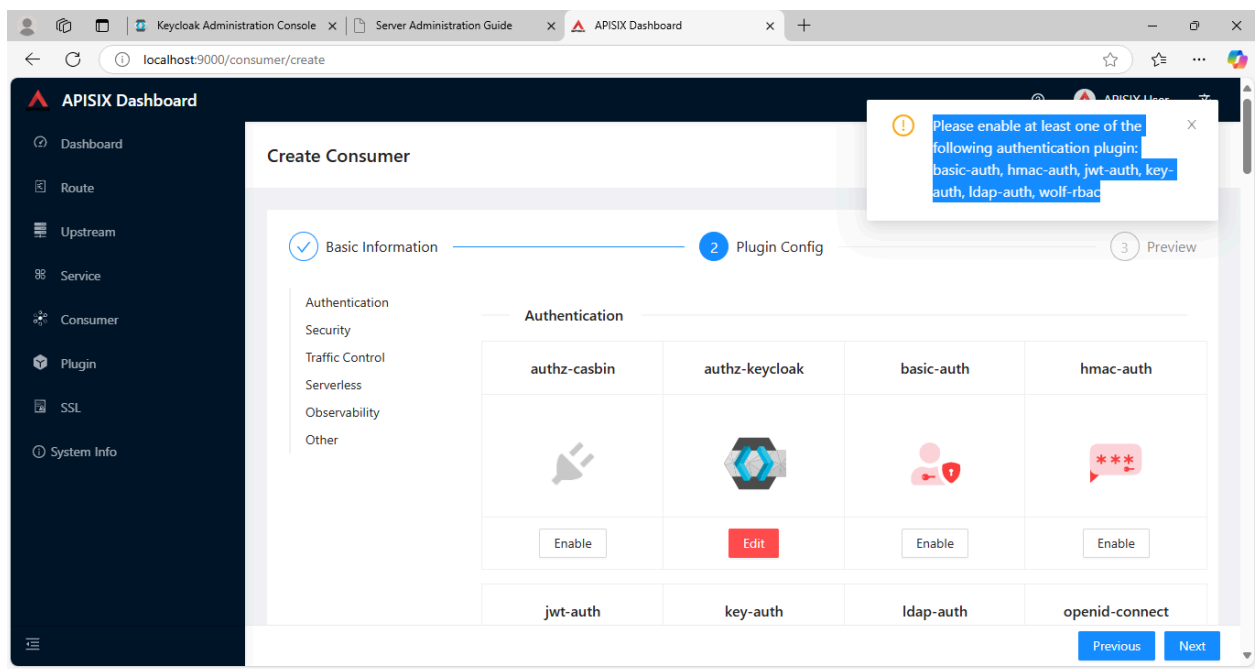
In plugin : i can enable a plugin and scope will be global

A Plugin configuration can be bound directly to a [Route](#), [Service](#), [Consumer](#) or [Plugin Config](#). You can refer to [Admin API plugins](#) for how to use this resource.

Uoi can do consumer restriction using consumer plugin in routes

You can label a route when defining route and use it later for grouping . in route you can clock on expand and use label to filter

In route you can also click advanced and use plugin template to apply authentication technique to bunch of it for example associate a label with custom and do whitelist based on that



| | |
|--|--|
| <pre>{ "uri": "/hy", "name": "hy-route", "methods": ["GET", "POST", "PUT", "DELETE", "PATCH",] }</pre> | <pre>{ "uri": "/hello", "name": "hello route", "methods": ["GET", "POST", "PUT", "DELETE", "PATCH",] }</pre> |
|--|--|

```

"HEAD",
"OPTIONS",
"CONNECT",
"TRACE"
],
"host": "127.0.0.1",
"plugins": {
  "basic-auth": {
    "disable": false
  },
  "response-rewrite": {
    "body": "bello",
    "disable": true,
    "status_code": 200
  }
},
"upstream": {
  "nodes": [
    {
      "host": "127.0.0.1",
      "port": 8080,
      "weight": 1
    }
  ],
  "timeout": {
    "connect": 6,
    "send": 6,
    "read": 6
  },
  "type": "roundrobin",
  "scheme": "http",
  "pass_host": "pass",
  "keepalive_pool": {
    "idle_timeout": 60,
    "requests": 1000,
    "size": 320
  }
},

```

```

"HEAD",
"OPTIONS",
"CONNECT",
"TRACE"
],
"host": "127.0.0.1",
"plugins": {
  "basic-auth": {
    "disable": false
  }
},
"upstream": {
  "nodes": [
    {
      "host": "127.0.0.1",
      "port": 9080,
      "weight": 1
    }
  ],
  "timeout": {
    "connect": 6,
    "send": 6,
    "read": 6
  },
  "type": "roundrobin",
  "scheme": "http",
  "pass_host": "pass",
  "keepalive_pool": {
    "idle_timeout": 60,
    "requests": 1000,
    "size": 320
  }
},
"status": 1
}

```

| | |
|--------------------------|--|
| <pre>"status": 1 }</pre> | |
|--------------------------|--|