Cloud Native – CMHK 2 Days

Trainer
Felix Tsang

擅長雲管理和雲計算技術 , IaaS 和 PaaS 及原生雲解決方案。
享受知識分享,也是國際開源基金會Linux Foundation APAC 的Open Source Evangelist, 推廣開源科技
CNCF認證 : LFAI , CKA, CKAD, CKS (Instructor , Developer , Security Specialist)
RedHat認證： RHCI (Instructor), RHCA (Architect)

Email : felix@linux.com
Blog: https://medium.com/@felixltd

Cloud Native Architecture

Cloud providers offer a variety of on-demand services, starting with simple (virtual-) servers, networking, storage, databases, and much more. Deploying and managing these services is very convenient, either interactively, or by using application programming interfaces (APIs).

In this chapter, you will learn about the principles of modern application architecture, often referred to as Cloud Native Architecture. We will discover what makes these applications *native* to cloud systems and how they differ from traditional approaches.

By the end of this section , you will learn the following:
•Discuss the basics of cloud native technologies.
•Understand the high level architecture of Kubernetes.
•Deal with the challenges of container orchestration.
•Discuss how container orchestration differs from legacy deployments.

Cloud Native Architecture - **CNCF Cloud Native Definition**

What is Cloud Native?

"Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

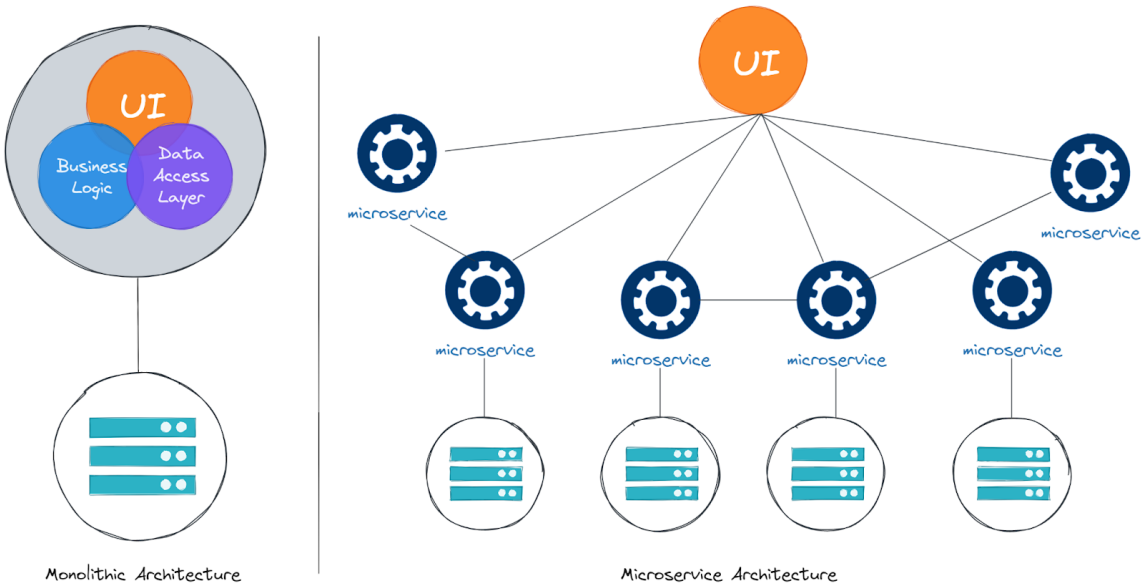
These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil."

"云原生技术有利于各组织在公有云、私有云和混合云等新型动态环境中，构建和运行可弹性扩展的应用。云原生的代表技术包括容器、服务网格、微服务、不可变基础设施和声明式API。

这些技术能够构建容错性好、易于管理和便于观察的松耦合系统。结合可靠的自动化手段，云原生技术使工程师能够轻松地对系统作出频繁和可预测的重大变更。"

Source from [CNCF (Cloud Native Computing Foundation)]

Cloud Native Architecture - **CNCF Cloud Native Definition**



How small the microservices?

a clearly defined scope of functions are often referred to as *microservices*.

Monolithic vs Microservices Architecture

- Increase complexity of applications
- Enable the change of the growing demand by users.
- Multiple teams, each holding ownership of different functions of application

Cloud Native Architecture Characteristics

- High level of automation
  - CI/CD , Building, testing and deploying applications as well as infrastructure with minimal human involvement

- Self healing
  - health checks which help monitor your application from the inside and automatically restart them if necessary.

- Scalable
  - starting multiple copies of the same application and distributing the load across them

- (Cost-) Efficient
  - scaling up your application for high traffic situations, scaling down your application

- Easy to maintain
- - Using *Microservices* allows to break down applications in smaller pieces and make them more portable, easier to test and to distribute across multiple teams.

- Secure by default
  - Cloud environments are often shared between multiple customers or teams, which calls for different security models.

Cloud Native Architecture Characteristics

A good baseline and starting point for your cloud native journey is the twelve-factor app. The twelve factor app is a guideline for developing cloud native applications, which starts with simple things like version control of your codebase, environment-awar

I. Codebase
One codebase tracked in revision control, many deploys

II. Dependencies
Explicitly declare and isolate dependencies

III. Config
Store config in the environment

IV. Backing services
Treat backing services as attached resources

V. Build, release, run
Strictly separate build and run stages

VI. Processes
Execute the app as one or more stateless processes

VII. Port binding
Export services via port binding

VIII. Concurrency
Scale out via the process model

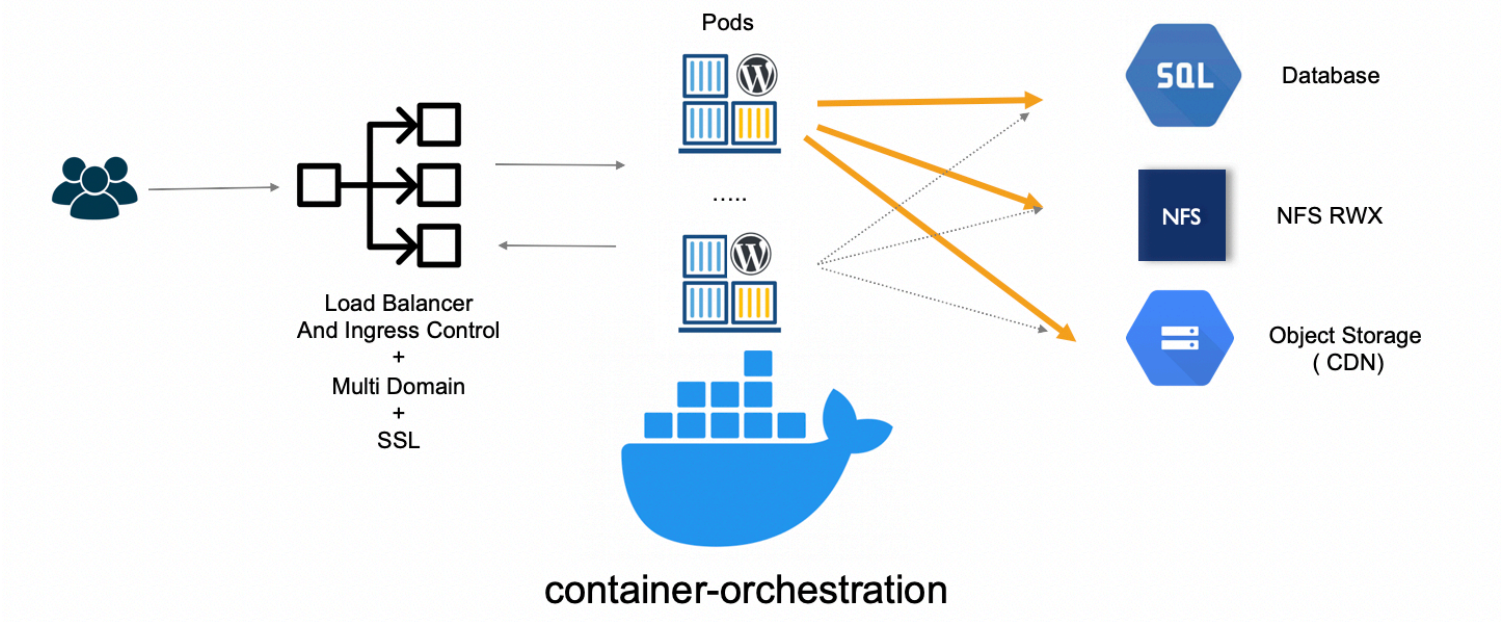
IX. Disposability
Maximize robustness with fast startup and graceful shutdown

X. Dev/prod parity
Keep development, staging, and production as similar as possible

XI. Logs
Treat logs as event streams

XII. Admin processes
Run admin/management tasks as one-off processes

Cloud Native Architecture Characteristics



Cloud Native Roles

- Cloud Architect
  Responsible for adoption of cloud technologies, designing application landscape and infrastructure, with a focus on security, scalability and deployment mechanisms.

- DevOps Engineer
  Often described as a simple combination of developer and administrator, but that doesn't do the role justice. DevOps engineers use tools and processes that balance out software development and operations. Starting with approaches to writing, building, and testing software throughout the deployment lifecycle.

- Full-Stack Developer
  An all-rounder who is at home in frontend and backend development, as well as infrastructure essentials.

- Site Reliability Engineer (SRE)
- Security Engineer
- DevSecOps Engineer
- Data Engineer

Discussion

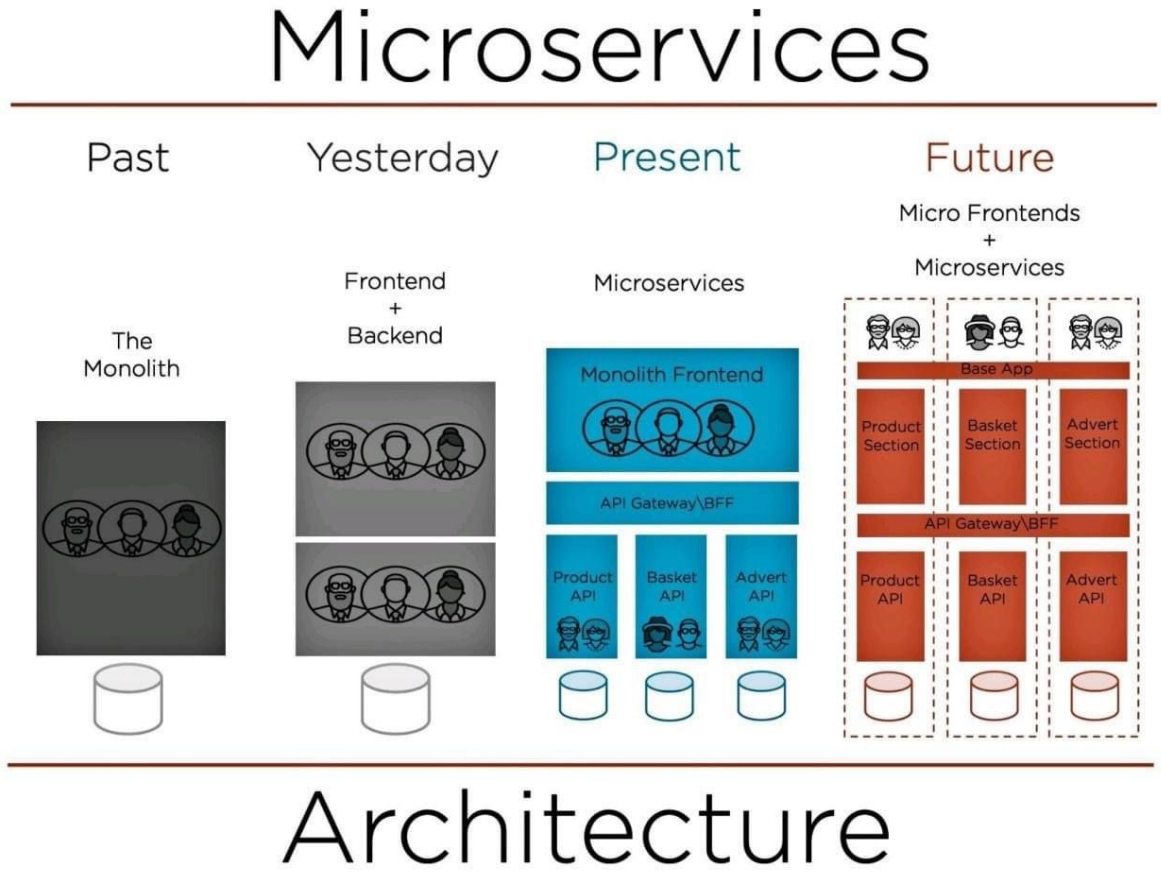
Traditional application delivery/deployment and problems…

Modern Application Development

Modern Application Development

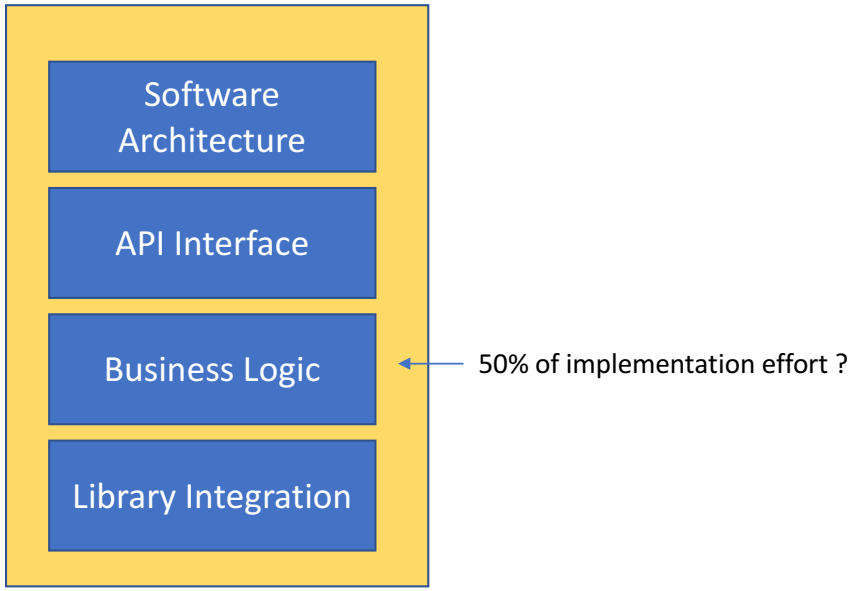
1. Short Development Cycle
2. Function change on demand
3. Highly integrated
4. PoC from market
5. Pilot Project model
6. Faster deployment
7. Multi-team

8. ……



Application Coding Element

Yesterday App Function Unit



”

Can we increate the ratio
of business logic section ?

Quarkus for Microservices Development

One of the first implementations of the MicroProfile specification was the Wildfly Swarm / Thorntail application server, based on the Wildfly server code, which failed to provide the startup performance required in the microservices environment due to the dynamic nature of the majority of Java EE specifications. To achieve the desired performance these implementations needed a rewrite that would reduce startup time by relying on a more complete build process, which performs activities normally done during startup or lazily at runtime, including configuration, bootstrapping and reflection preparation. Quarkus emerged as the collective effort from the different technologies that encompass the MicroProfile implementation.
Quarkus focuses on these key factors to provide the best development experience for Microservices:

Container Native

Quarkus is first and foremost a cloud native framework, which means that the main deployment platform is the container. To be a container first framework, special care was taken to have:

Quarkus for Microservices Development

One of the main drives in Quarkus is to provide the best support for cloud workloads, Serverless and FaaS use cases. In this environment size, memory footprint and startup time are more important than other metrics.
Despite these optimized performance gains over traditional cloud frameworks, an approximate 1 second startup time (depending on the plug-ins used), might still be too long (depending on business requirements). For this, Quarkus offers full support of the GraalVM's Ahead of Time (AoT) compilation.
GraalVM is a special JVM from Oracle, which among other features offers AoT compilation to executable, for Java applications, with some limitations. The most important requirement to support this native executable generation is the Closed World Assumption, which means the JVM needs to know at compilation time which classes the application is going to use. Quarkus achieves this by rewriting the implementations of the libraries that conform to the standard and assess the required classes at build time instead of at runtime through reflection.
During this AoT compilation GraalVM removes all unneeded classes in the application, the dependent libraries, and the JVM.
Startup time for native applications is usually around tens of milliseconds, and uses less resources because the application only loads the required classes.
This native support makes Quarkus the best suited framework for cloud deployments where startup time and memory usage are hard constraints. Also in Serverless and FaaS environments where requirements are similar, Quarkus provides the best alternative.

Advantages of Quarkus

Compared to other Java Cloud Native frameworks, Quarkus offers:

- Faster time to respond to the first request.
- Higher request per second ratio.
- Lower memory usage.
- Higher throughout in reactive routes.
- Full native image support.
- Passes MicroProfile conformance tests.
- Developer Centric

Traditional cloud native frameworks offer slower startup times

Let’s check it out !

Lab0 Quarkus Sample Project Hello World

Cloud Native Training
Lab Environment Setup V1

System Requirements:

OS

1. Windows 10 or above
2. Mac OS 10.15 or above

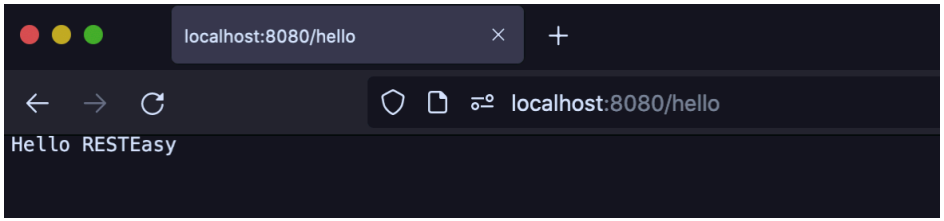
Hardware requirement

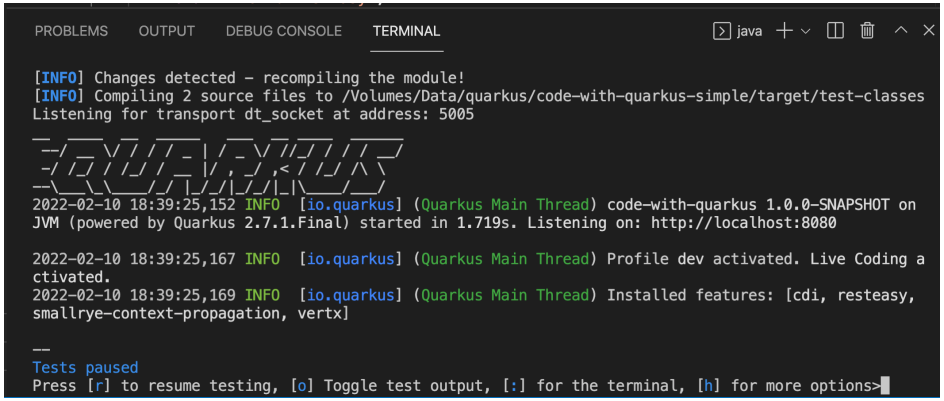
1. CPU i5 or above
2. RAM 8GB or above
3. Hard Disk 80GB free or above

Software List

1. JDK 17 or above
2. Apache Maven 3.8.3 or above
3. IDE (Visual Studio Code)
4. [Pre-Lab] Quarkus Sample Project Hello World
5. (Optional) VMware Player or Virtual Box (for Kubernetes Installation) for Ubuntu 18.04.5 (x2 VMs)
5. (Optional) Script for install Kubernetes Cluster(1 Master Node, 1 Workder Node) on Ubuntu 18.04.5

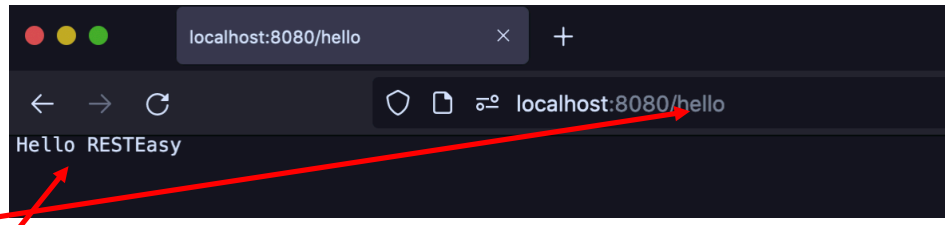
1) JDK Installation





Complete the Cloud-native lab Environment setup

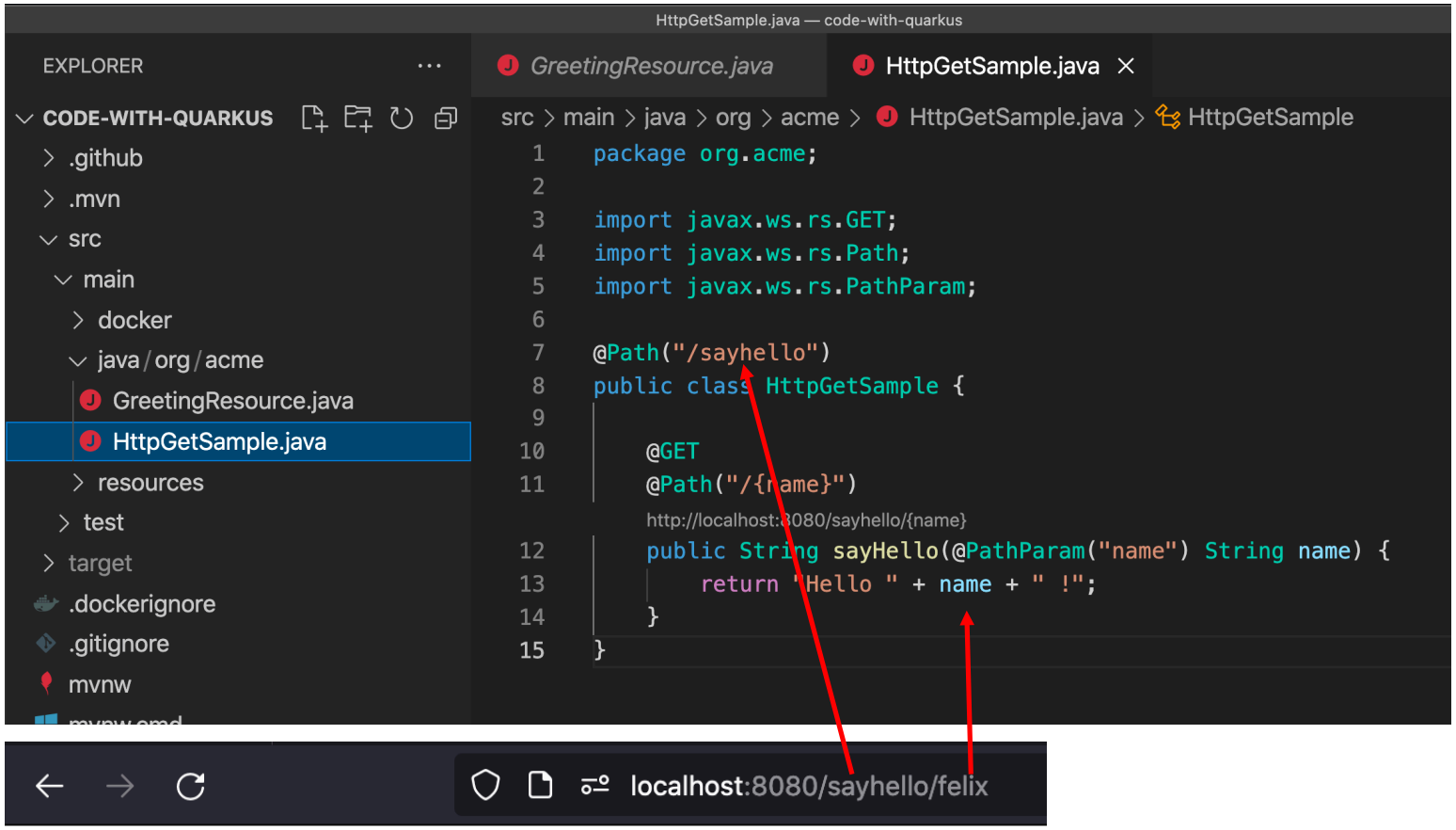
```java
package org.acme;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/hello")
public class GreetingResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return "Hello RESTEasy";
    }
}
```



Focus on your business logic ?

Lab1 Retrieve Parameters from API Call

Lab1 Retrieve Parameters from API Call



Hello felix !

Lab1 Retrieve Parameters from API Call

```
@Path("/{name}")
http://localhost:8080/sayhello/{name}
public String sayHello(@PathParam("name") String name)
```

@PathParam Binds to a segment of the URI. It is possible to include a placeholder with a matching name in the @Path value. An example URI that includes a path parameter is: http://www.example.com/currencies/USD

@QueryParam Binds to an HTTP query parameter using its name. An example URI that includes a query parameter is: http://www.example.com/rest?name=Test

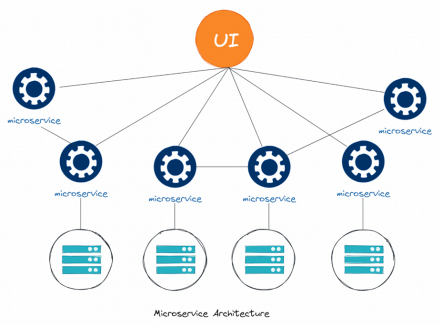
@HeaderParam Binds to a header in the HTTP request using its name.

@FormParam Binds to a form field using its name.

Understanding JSON Serialization and Deserialization in Quarkus

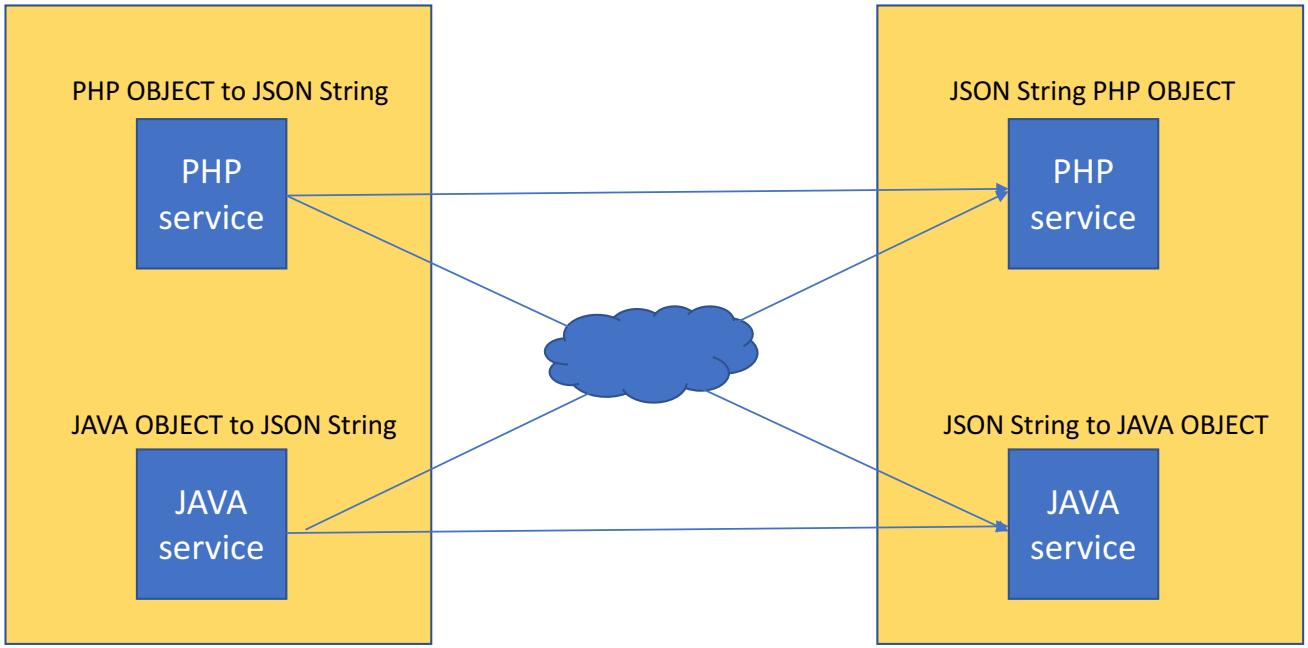
Traditional Java EE was dependent on JSON-P for all kinds of JSON manipulation. Quarkus, following JAX-RS 2.1 specification, uses JSON-B, and for the common cases, you can rely on Quarkus' JSON capabilities for parsing and generating the input and outputs of your services. If you define your endpoints with a @GET, @POST, @PUT, or @DELETE annotation, then Quarkus will attempt to serialize the returned object.

How function exchange data each others ?



REST API

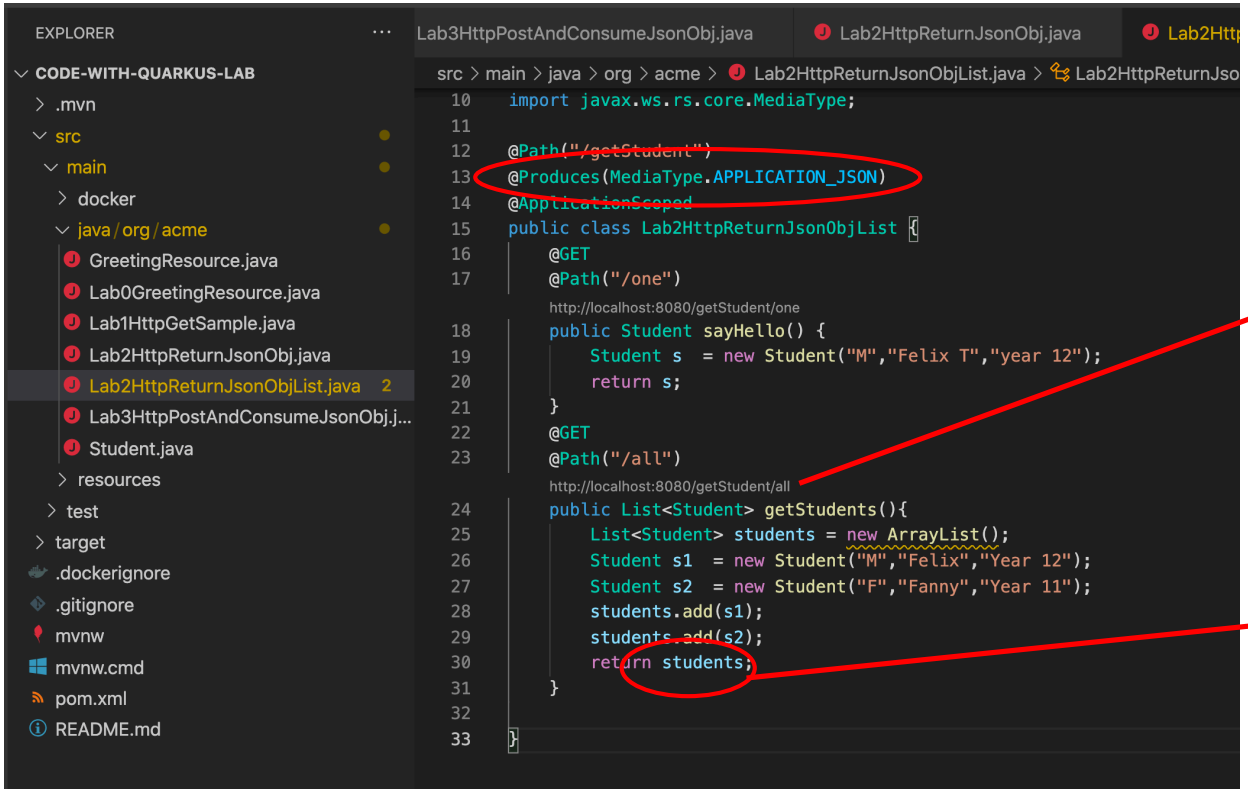
....STRING ....



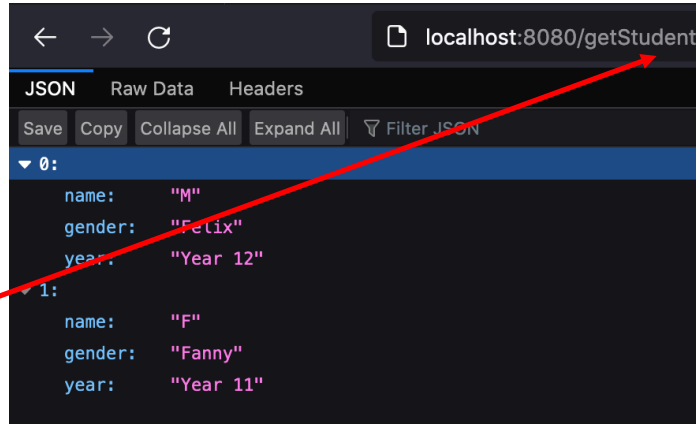
Lab2 Response with JSON Obj

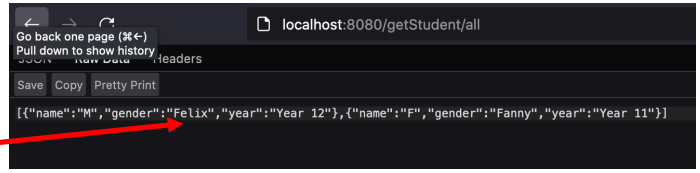
Convert Object to JSON String

Lab2 Response with JSON Obj



Output

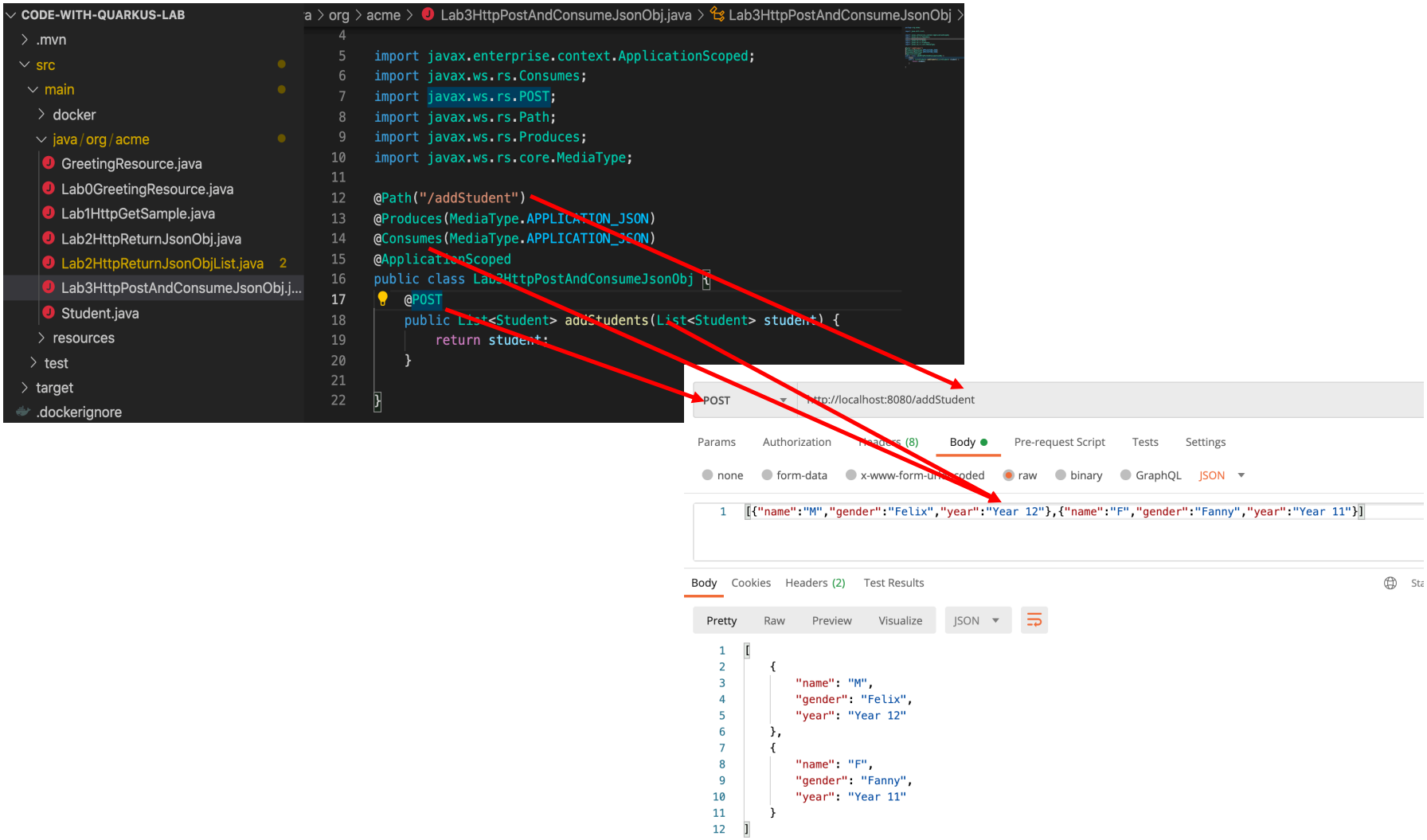


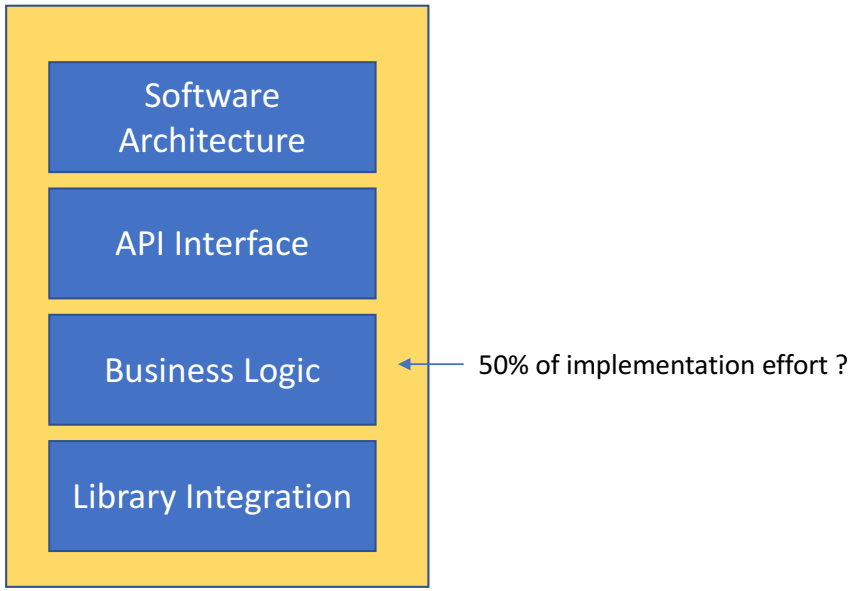


LAB3 HTTP Post Consume JSON Obj

Convert JSON String to Object

LAB3 HTTP Post Consume JSON Obj





After Lab1 -3
Think what you need to focus again?

?

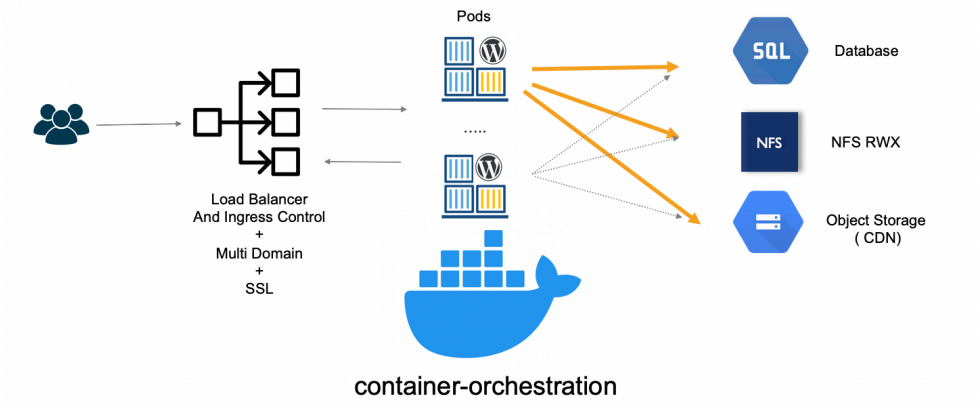
1. Expose Function as REST API interface
2. REST API Communication
3. Business Logic

App Characteristics

1. Auto scaling
2. Multiple instances

→ Stateless →

1.Authentication
2.Persistent Data



App Characteristics

1.Authentication

Maintaining Security in Microservices
Maintaining identity and access management through a series of independent services can be a real challenge in microservice-based applications. Requiring every service call to include an authentication step is not ideal. Fortunately, there are a number of possible solutions, including:

Single Sign-On
A common approach for authentication and authorization that permits the client to use a single set of login credentials to access multiple services.

Distributed sessions
A method of distributing identity between microservices and the entire system.

Client-side token
The client requests a token and uses this token to access a microservice. The token is signed by an authentication service. A microservice validates the token without calling the authentication service. JSON Web Token (JWT) is an example of token-based authentication

Client-side token with API Gateway
API gateways cache client-side tokens. The validation of tokens is handled by the API gateway

JSON Web Token (JWT)

What is JSON Web Token?

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA** or **ECDSA**.

Although JWTs can be encrypted to also provide secrecy between parties, we will focus on *signed* tokens. Signed tokens can verify the *integrity* of the claims contained within it, while encrypted tokens *hide* those claims from other parties. When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it.

When should you use JSON Web Tokens?

Here are some scenarios where JSON Web Tokens are useful:

•**Authorization**: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.

•**Information Exchange**: JSON Web Tokens are a good way of securely transmitting information between parties. Because JWTs can be signed—for example, using public/private key pairs—you can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

Client-side token - JSON Web Token (JWT)

JWT

Session, Cookies

透過 Session 和 Cookie 實作驗證機制

JSON Web Token (JWT)

What is the JSON Web Token structure?
In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:

1. Header
2. Payload
3. Signature

Therefore, a JWT typically looks like the following.

Base64URL(Header). Base64URL(Payload).Signature
Eg:
xxxxx.yyyyy.zzzzz

Let's break down the different parts.

JSON Web Token (JWT)

Header

The header *typically* consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

For example:

```
{
  "alg": "HS256",
  "type": "JWT”
 }
```

Then, this JSON is **Base64Url** encoded to form the first part of the JWT.

JSON Web Token (JWT)

Payload

The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: *registered*, *public*, and *private* claims.

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

Then, this JSON is **Base64Url** encoded to form the first part of the JWT.

Official suggested attributes

- iss (issuer)：簽發人
- exp (expiration time)：過期時
- sub (subject)：主題
- aud (audience)：受眾
- nbf (Not Before)：生效時間
- iat (Issued At)：簽發時間
- jti (JWT ID)：編號

JSON Web Token (JWT)

Signature
To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.
For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

```
HMACSHA256(
base64UrlEncode(header) + "." + base64UrlEncode(payload),
secret)
```

The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.

The following shows a JWT that has the previous header and payload encoded, and it is signed with a secret.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

LAB4 JWT

Using JWT RBAC

LAB4 JWT

Using JWT RBAC

This guide explains how your Quarkus application can utilize SmallRye JWT to verify JSON Web Tokens, represent them as MicroProfile JWT org.eclipse.microprofile.jwt.JsonWebToken and provide secured access to the Quarkus HTTP endpoints using Bearer Token Authorization and Role-Based Access Control.

Quarkus OpenID Connect quarkus-oidc extension also supports Bearer Token Authorization and uses smallrye-jwt to represent the bearer tokens as JsonWebToken, please read the Using OpenID Connect to Protect Service Applications guide for more information. OpenID Connect extension has to be used if the Quarkus application needs to authenticate the users using OIDC Authorization Code Flow, please read Using OpenID Connect to Protect Web Applications guide for more information.

LAB4 JWT **Using JWT RBAC**

Validate JWT

LAB4 JWT **Using JWT RBAC**

Generate JWT

LAB4 JWT **Using JWT RBAC**

https://jwt.io/

```
String token =
    Jwt.issuer("https://felixtsang.com/issuer")
       .upn("felix@linux.com")
       .preferredUserName("felix")
       .claim("name", "nathan")
       .groups(new HashSet<>(Arrays.asList("User", "Admin")))
       .claim(Claims.birthdate.name(), "2001-07-13")
       .claim("userid", "123")
    .sign();
```

Encoded PASTE A TOKEN HERE


```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.ey
Jpc3MiOiJodHRwczovL2ZlbGl4dHNhbmcuY29tL
2lzc3VlciIsInVwbiI6ImZlbGl4QGxpbnV4LmNv
bSIsInByZWZlcnJlZF91c2VybmFtZSI6ImZlbGl
4IiwibmFtZSI6Im5hdGhhbiIsImdyb3VwcyI6Wy
JVc2VyIiwiQWRtaW4iXSwiYmlydGhkYXRlIjoiM
jAwMS0wNy0xMyIsInVzZXJpZCI6IjEyMyIsImlh
dCI6MTY0NDkyMzg1MywiZXhwIjoxNjQ0OTI0MTU
zLCJqdGkiOiI0ZjZlOTU2Yy1hNWJkLTQ4ZDUtYT
gzNy02OWYyMzBiY2I2ZTIjfQ.Lw2XOyu2UJF4Xz
zx43e6fvlKCrgJLBCdrZKJxfg6IyCn5fPedbkIf
EKuHJ5xQiqOFJakYqBdNdh6G1ZGSDxme33Q611S
6n3fEoKBVW1WtSYZRxT1kTFfeKicSJNCg9yO6v4
waAMmcu_m7fTHp0ffAqktQt_VfSSdqmf1DYVr2J
w
YTzhhpBjrFlil8XivF5vVYTrTdGaShz6MPD41T9
sBfNzchxA9g04MZQJdcuHY-_-
QuEu9kmRsi3vbxMaKykWF7fBecX6fREEI6Uwtx8
xXEGToSrS9f_XIf5wKcYYiQjfarnLr8LBFDbQMr
Zztlx29agV-nmOYfJBrqWG6kEDTw
```


Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE


```
{
  "typ": "JWT",
  "alg": "RS256"
}
```


PAYLOAD: DATA


```
{
  "iss": "https://felixtsang.com/issuer",
  "upn": "felix@linux.com",
  "preferred_username": "felix",
  "name": "nathan",
  "groups": [
    "User",
    "Admin"
  ],
  "birthdate": "2001-07-13",
  "userid": "123",
  "iat": 1644923853,
  "exp": 1644924153,
  "jti": "4f6e956c-a5bd-48d5-a837-69f230bcb6e2"
}
```


VERIFY SIGNATURE

```
RSASHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  Public Key in SPKI, PKCS #1,
  X.509 Certificate, or JWK stri
  ng format.
)
  Private Key in PKCS #8, PKCS #
  1, or JWK string format. The k
  ey never leaves your browser.
```

Lab summary

Recap Container *Orchestration Tool*

7 June 2014 By Google

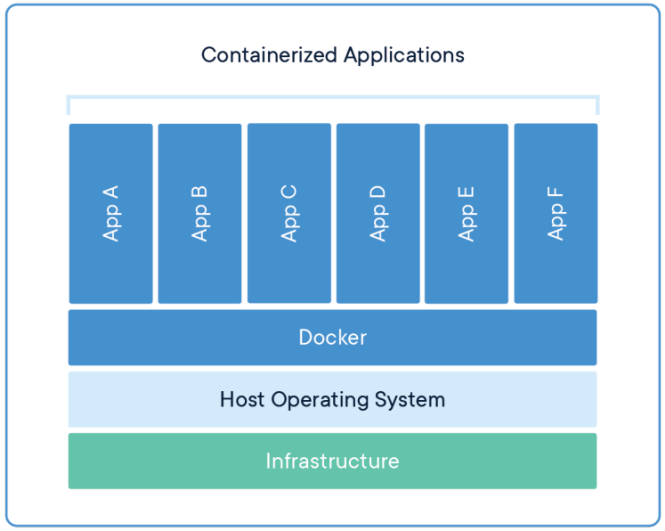
2015 (CNCF)

(CNCF) is a Linux Foundation project

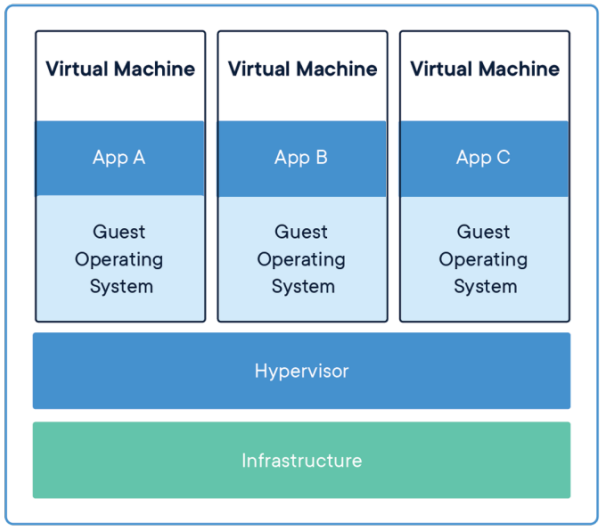
Kubernetes (commonly stylized as **K8s**[4]) is an open-source container-orchestration system for automating computer application deployment, scaling, and management.[5]
It was originally designed by Google and is now maintained by the Cloud Native Computing Foundation. It aims to provide a "platform for automating deployment, scaling, and operations of application containers across clusters of hosts".[6] It works with a range of container tools, including Docker.[7]
Many cloud services offer a Kubernetes-based platform or infrastructure as a service (PaaS or IaaS) on which Kubernetes can be deployed as a platform-providing service. Many vendors also provide their own branded Kubernetes distributions.

https://en.wikipedia.org/wiki/Kubernetes

Container vs VM

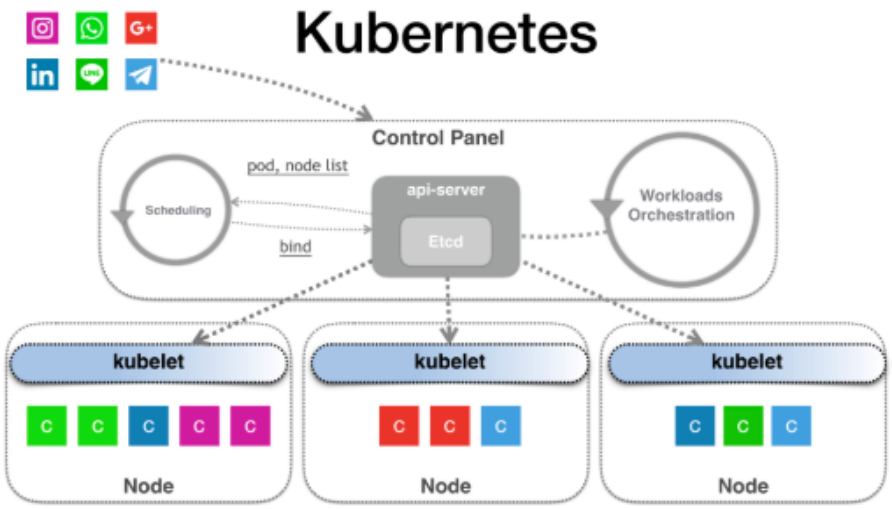


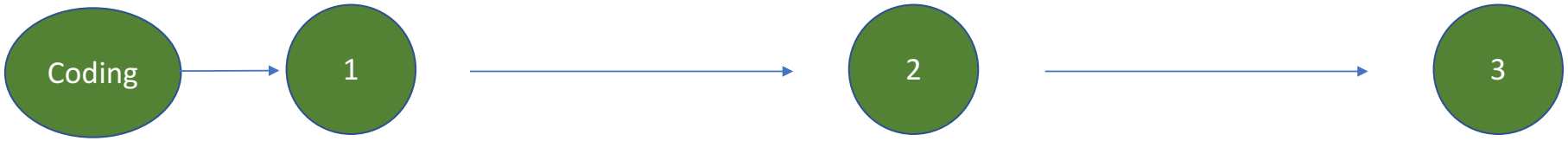
Abstract of OS kernel



Abstract of Hardware

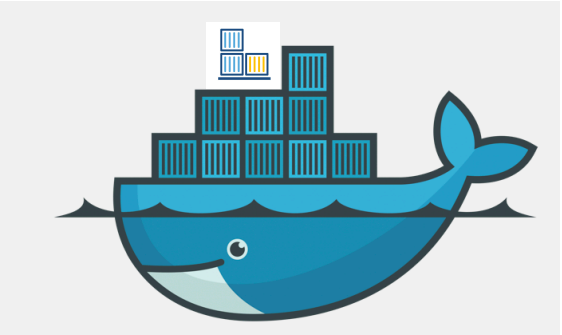
- *Container is a Process*
- *Kubernetes is an Orchestration tool*

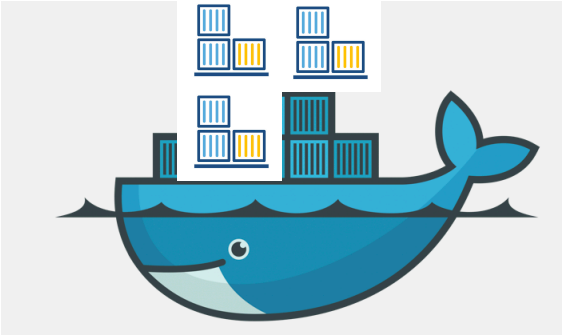




Containerized Application

> Source / Docker image

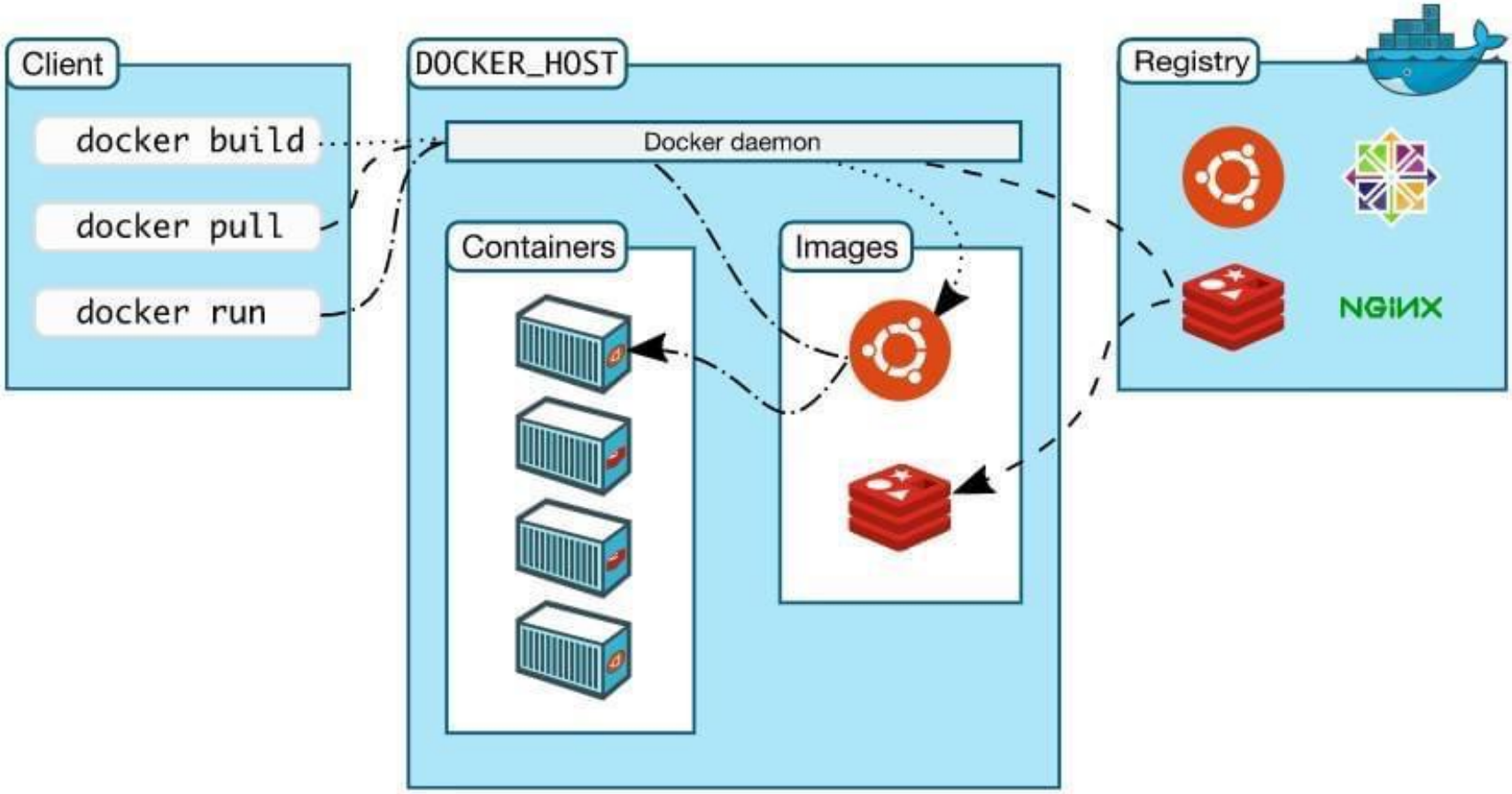


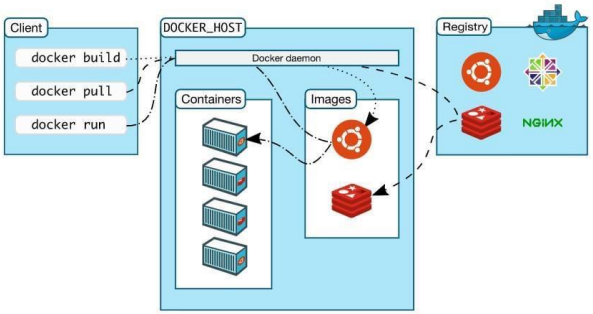


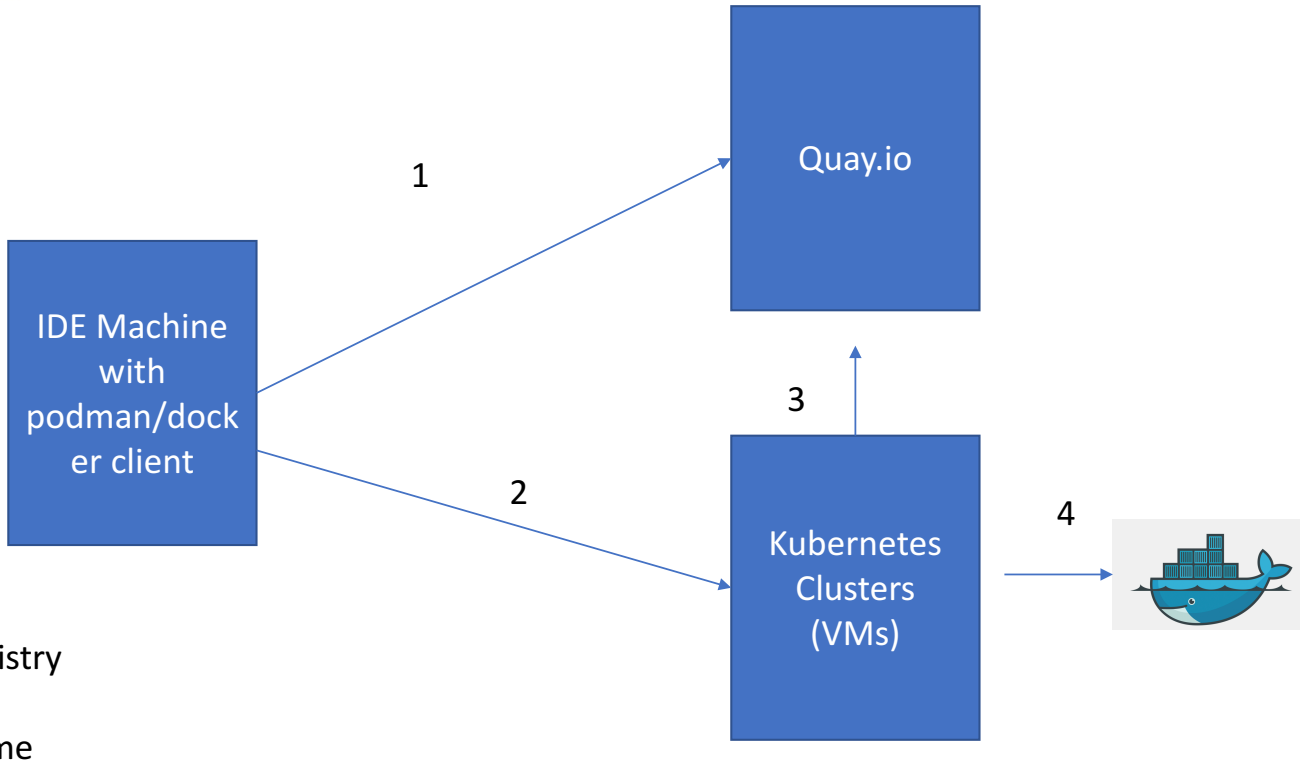
Application + Running ENV (container)

up and run

container-orchestration

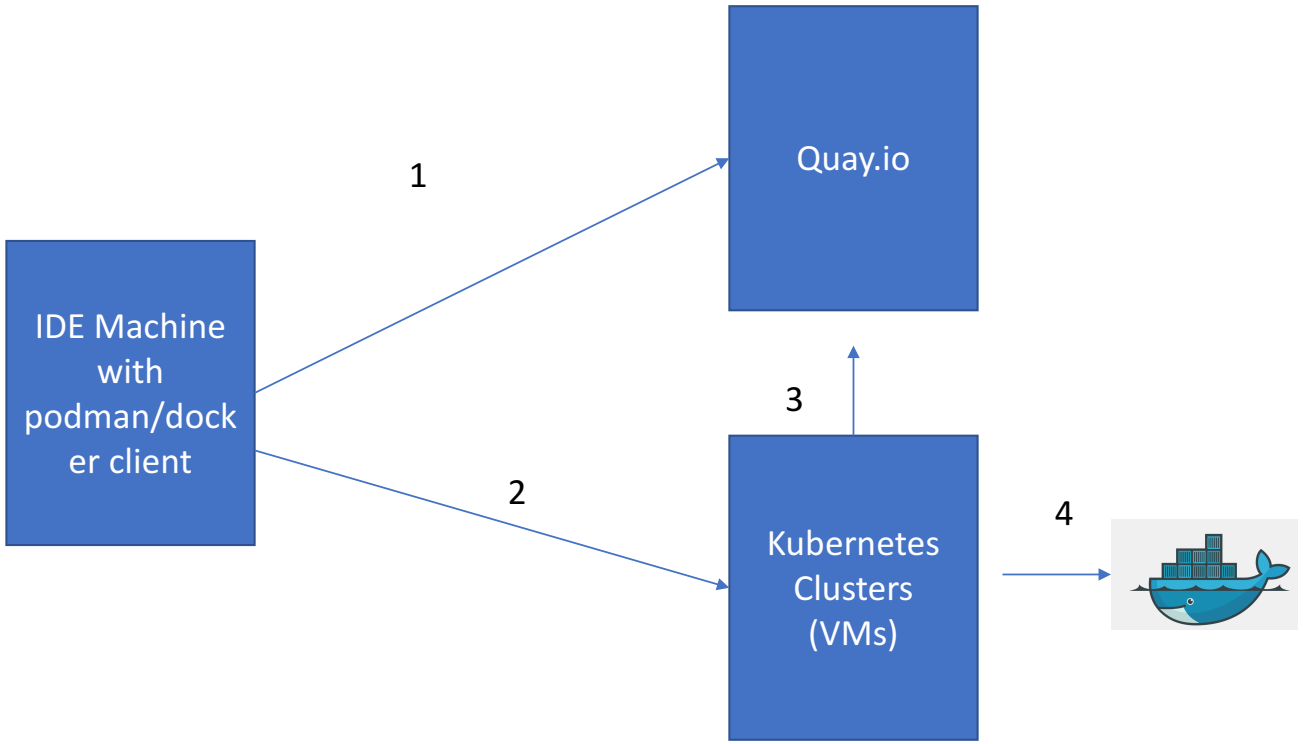

Client
docker build
docker pull
docker run
DOCKER_HOST
Docker daemon
Containers
Images
Registry
NGINX





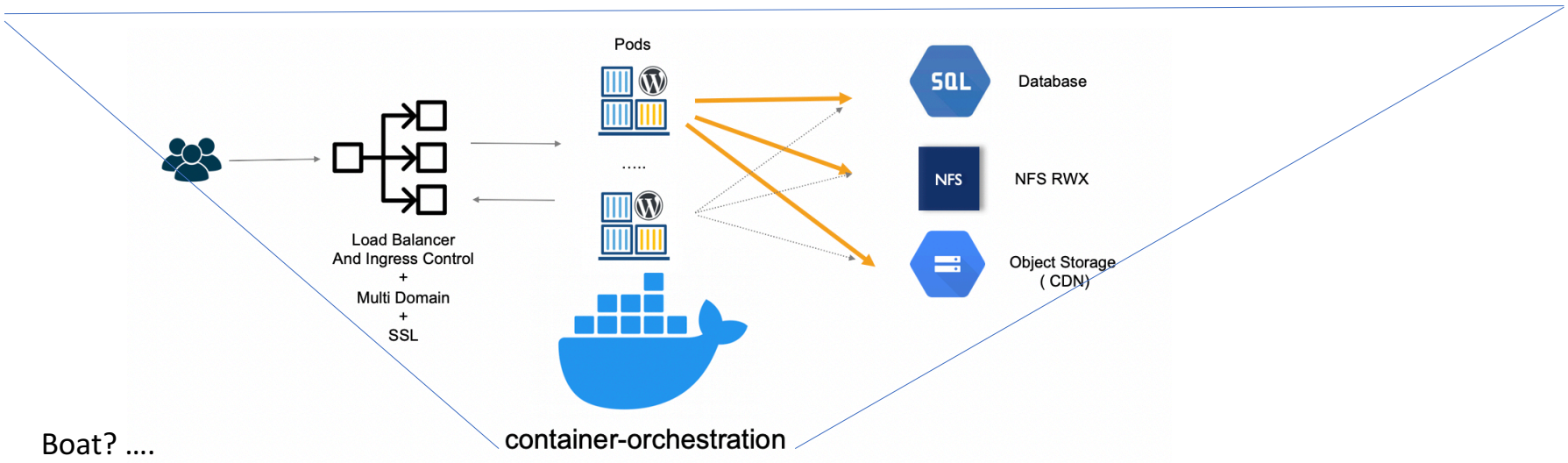
1.Build and upload image to quay.io registry
2.Create deployment on k8s cluser
3.Get registry image for createing runtime
4. Run time up and run

What Resources Created of the output yaml ?



Resources Created
1.Deployment
2.Pod
3.Service (NodePort)

Evaluation QR code



Boat? ....

More , challenge yourself

Hello World with NodePort Service on kubernetes cluster

https://kubernetes.io/docs/tasks/access-application-cluster/service-access-application-cluster/

Deploy app image to quay.io and apply to kubernetes cluster

https://redhat-developer-demos.github.io/quarkus-tutorial/quarkus-tutorial/kubernetes.html

0.kubernetes installation script https://github.com/lftsang/cloudnative

1.Create quay.io ac , and create the login token under setting
2.Create organization
3. mvn clean package -DskipTests -Dquarkus.container-image.push=true
4. Go to quay.io to set the repo public

```
kubectl run nginx --image=nginx
```

kubeadm token create --print-join-command