HOW CONTAINER ORCHESTRATION CAN STRENGTHEN YOUR MICRO-SERVICES

THE APPROACH OF KUBERNETES

Riccardo Poggi
HOW CONTAINER ORCHESTRATION CAN STRENGTHEN YOUR MICRO-SERVICES

THE APPROACH OF KUBERNETES

Riccardo Poggi

1. MICRO-SERVICES ARCHITECTURE
2. CONTAINERISED MICRO-SERVICES
3. CONTAINER ORCHESTRATION
HOW CONTAINER ORCHESTRATION CAN STRENGTHEN YOUR MICRO-SERVICES

THE APPROACH OF KUBERNETES

Riccardo Poggi

1. MICRO-SERVICES ARCHITECTURE
2. CONTAINERISED MICRO-SERVICES
3. CONTAINER ORCHESTRATION
MONOLITH APPLICATION

- Key aspects
  - Single code-base
  - Single build system
  - Single executable
MICRO-SERVICES

• Key aspects
  - Loosely coupled
  - Independently deployable
  - API service communication

MICRO-SERVICES

USER INTERFACE

POST

GET

GET

GET

SHIPMENT
DATABASE

ORDER

INVENTORY

ACCOUNT
DATABASE

DATABASE

DATABASE
MICRO-SERVICES

• Key aspects
  - Loosely coupled
  - Independently deployable
  - API service communication
FROM SOA TO MICRO-SERVICES

**CLOUD SERVICE CONSUMER**

**USER INTERFACE**

**DATABASE**

**ACCOUNT**

**ORDER**

**SHIPMENT**

**INVENTORY**

**POST**

**GET**

**ENTERPRISE SERVICE BUS**

**CONSUMER**

**C++**

**Java**

**Java**

**C#**

**C++**

**USER INTERFACE**
MICRO-SERVICES ARCHITECTURE

**BENEFITS**
- Highly scalable
- Resilient
- Easy to deploy
- Accessible
- More open

**CHALLENGES**
- Building
- Testing
- Deployment
- Logging
- Monitoring
- Connectivity
UNEXPECTED FAILURE

“Dealing with unexpected failures is one of the hardest problems to solve especially in a distributed system”
FAULT-TOLERANCE

• Fault-tolerance
  - System able to continue proper operation in the event of failure of one or more of its components

• Resilience

• Graceful degradation
  - The ability of maintaining functionality when portions of a system break down
FAULT-TOLERANCE

- Fault-tolerance
  - System able to continue proper operation in the event of failure of one or more of its components

- Resilience

- Graceful degradation
  - The ability of maintaining functionality when portions of a system break down
HIGH-AVAILABILITY

- Redundancy
  - Eliminate single points of failure
  - Failure of a component does not mean failure of the entire system

- Reliable crossover
  - Not to have crossover be a single point of failure

- Monitoring
  - Detection of failures as they occur
  - A user may never see a failure, but the maintenance activity must
HIGH-AVAILABILITY

• Redundancy
  – Eliminate single points of failure
  – Failure of a component does not mean failure of the entire system

• Reliable crossover
  – Not to have crossover be a single point of failure

• Monitoring
  – Detection of failures as they occur
  – A user may never see a failure, but the maintenance activity must
FAIL-OVER POLICY

• Fail-over policy
  – Failure as an unrecoverable critical issue
  – Implementing the behaviour a service follows in case of its own failure

• Last action before failing
  – Does the service holds important data which needs to be saved?
  – Does the service has a configuration or status which needs to be saved?

• Termination
  – “Failure” can also be externally induced
  – Graceful kill (close)

```
def sigterm_handler(signal, frame):
    # save the state here or do whatever you want
    print('booyah! bye bye')
    sys.exit(0)

signal.signal(signal.SIGTERM, sigterm_handler)
```
SCALABILITY & ELASTICITY

- Requirement as a function of time
  - Resource allocation and server instantiation
- Scalability
  - Increasing the capacity
  - The available resources match the current and future usage plans
  - Scaling up: increasing the ability of an individual server
  - Scale out: adding multiple servers
- Elasticity
  - Increasing or reducing the capacity based on the load
  - The available resources match the current demands as closely as possible
**SCALABILITY & ELASTICITY**

- Requirement as a function of time
  - Resource allocation and server instantiation

- Scalability
  - Increasing the capacity
  - The available resources match the current and future usage plans
  - Scaling up: increasing the ability of an individual server
  - Scale out: adding multiple servers

- Elasticity
  - Increasing or reducing the capacity based on the load
  - The available resources match the current demands as closely as possible
CONTINUOUS DELIVERY

- Independent deploy
- Without service interruption
  - No downtime!
- Rebuild and redeploy
  - only one or a small number of services
CONTINUOUS DELIVERY

• Independent deploy
• Without service interruption
  - No downtime!
• Rebuild and redeploy
  - only one or a small number of services
STATEFUL VS. STATELESS

• Stateful
  - Possess saved data in a database that they read from and write to directly
  - If it shares DB with other micro-services less decoupled
  - When it terminates it has to save its state (fail-over policy)

• Stateless
  - Handle request and return responses
  - All necessary information supplied on the request and can be forgot after the response
  - No permanent data
  - Nothing to save when it terminates
HOW CONTAINER ORCHESTRATION CAN STRENGTHEN YOUR MICRO-SERVICES

THE APPROACH OF KUBERNETES

Riccardo Poggi

1. MICRO-SERVICES ARCHITECTURE
2. CONTAINERISED MICRO-SERVICES
3. CONTAINER ORCHESTRATION
VMs have their own OS kernel, while containers share it with the host OS
CGROUPS & NAMESPACES

**CGROUP**

Cpu, memory, I/O, ...

**NAMESPACE**

Cgroup, IPC, network, mount, PID, User, ...
CONTAINER RUNTIME

• Container Runtime
  – In a OCI/CNI compatible version is a daemon process
  – Creates and executes a container

• To fully create a container:
  1. Creates the rootfs filesystem.
  2. Creates the container
     • Set process namespaces and cgroups
  3. Connects the container to a network
  4. Starts the user process
The most widely known container runtime is Docker.

But there are also others:
- rkt, containerd, lxd, singularity, etc..
DOCKER IMAGE

- Docker images are built from a base image
- Base images are built up using instructions
  - Run a command
  - Add a file or directory
  - Create an environment variable
  - What process to run when launching a container from this image
# Dockerfile
FROM ubuntu:latest

RUN apt-get update
RUN apt-get install -y python python-pip
RUN pip install Flask

COPY . /app

CMD python /app/order_service.py

$ docker build -t my-image .
$ docker run my-image
CONTAINERISED MICRO-SERVICES

- Apply containers to microservices architecture
  - One-to-one map for single independent services container
  - Decoupling inside/outside container
- Questions still to be solved
  - Tightly coupled processes inside one container?
  - Everything running on one single node
  - Redundancy and scalability
CONTAINERISED MICRO-SERVICES

• Apply containers to micro-services architecture
  – One-to-one map for single independent services container
  – Decoupling inside/outside container

• Questions still to be solved
  – Tightly coupled processes inside one container?
  – Everything running on one single node
  – Redundancy and scalability
CONTAINERISED MICRO-SERVICES

• Apply containers to micro-services architecture
  – One-to-one map for single independent services container
  – Decoupling inside/outside container

• Questions still to be solved
  – Tightly coupled processes inside one container?
  – Everything running on one single node
  – Redundancy and scalability
HOW CONTAINER ORCHESTRATION CAN STRENGTHEN YOUR MICRO-SERVICES

THE APPROACH OF KUBERNETES

1. MICRO-SERVICES ARCHITECTURE
2. CONTAINERISED MICRO-SERVICES
3. CONTAINER ORCHESTRATION
ORCHESTRATION

• Orchestration
  - Automated arrangement
  - Coordination
  - Management

• Useful tool for
  - Service Discovery
  - Load Balancing
  - Health checks
  - Auto-scaling
  - Zero-downtime deploys
  - (And much more...)
PLATFORM OVERVIEW

User workloads

Distributed container management

Local container management

Container agnostic infrastructure

APPs

JOBS

SERVICES

ORCHESTRATOR

CONTAINER RUNTIME

INFRASTRUCTURE
KUBERNETES

• Master
  - The machine that controls Kubernetes nodes

• Node
  - The machines that perform the requested and assigned tasks

• Pod
  - A group of one or more containers deployed to a single node

• kubectl
  - Command line configuration tool for Kubernetes
A Pod is the basic building block of Kubernetes
- The smallest and simplest unit

“one-container-per-Pod”
- Most common Kubernetes use case
- Pod as a wrapper around a single container

Encapsulate multiple co-located containers
- Tightly coupled
- Need to share resources
REPLICASET

- Kubernetes Controller
  - Changes the system to move it from the current to the desired state

- ReplicaSet
  - Ensures that a specified number of pod replicas are running at any given time

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: demo
spec:
  replicas: 4
  selector:
    matchLabels:
      # this replicaset will apply to every template
      app: demo

# pod template spec
template:
  metadata:
    labels:
      app: demo
  spec:
    containers:
    - name: ubuntu
      image: ubuntu:18-demo
```

$ kubectl create -f replicaset.yaml
replicaset "demo" created
$ kubectl get replicaset demo
NAME  DESIRED  CURRENT  AGE
demo  2        2        40s
$ kubectl scale --replicas=4 replicaset/demo
replicaset "demo" scaled
$ kubectl delete replicaset demo
replicaset "demo" deleted
DEPLOYMENT

- Deployment controller
  - Declarative update for Pods and ReplicaSet
- Rollout
  - Ensure max unavailable/surge
  - e.g. at least 75% are up (25% max unavailable)
- Roll back

```bash
$ kubectl create -f deployment.yaml
deployment "demo" created

$ kubectl get deployment demo
NAME  DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
demo  2        2         2           2          40s

$ # alternatives: 'kubectl edit' or 'kubectl apply -f'
$ kubectl patch deployment -p {"spec": [...] "value": "v2"}
"demo" patched
$ kubectl rollout undo deployment/demo
$ kubectl delete deployment demo
deployment "demo" deleted
```
SERVICE

• Service
  – Abstraction to functionally group Pods
    – e.g. Front-end Pods, back-end Pods
• Consistent front for a set of Pods to offer a given service
• Possible to scale up and down Pods
ORCHESTRATED MICRO-SERVICES
SUMMARY OF OUR JOURNEY
SUMMARY OF OUR JOURNEY

MONOLITH APPLICATION

- APPLICATION
- USER INTERFACE
- SHOPPING CART
- ORDER STATUS
- ACCOUNT ACTIVATION
- INVENTORY
SUMMARY OF OUR JOURNEY

MONOLITH APPLICATION

APPLICATION
- USER INTERFACE
- SHOPPING CART
- ORDER STATUS
- ACCOUNT ACTIVATION
- INVENTORY

MICRO-SERVICES ARCHITECTURE

USER INTERFACE

SHIPMENT

DATABASE

ORDER

INVENTORY
SUMMARY OF OUR JOURNEY

MONOLITH APPLICATION

CONTAINER

MICRO-SERVICES ARCHITECTURE
SUMMARY OF OUR JOURNEY

MONOLITH APPLICATION

MICRO-SERVICES ARCHITECTURE

CONTAINER

ORCHESTRATION
THE END

Exercise session

Today @16:00

513-1-024 (CERN)

Thank You!
ACKNOWLEDGEMENTS

Many thanks to all those who helped shaping this lecture and exercises!

• iCSC Mentors:
  – Sebastian Lopienski
  – Enric Tejedor Saavedra

• CERN IT Support
  – Ricardo Brito Da Rocha

• Beta tester:
  – Luca Gardi
  – Marco Valente