Microservices-Coding

Sample Use Case- ECommerce Application

Ecommerce Application with features:

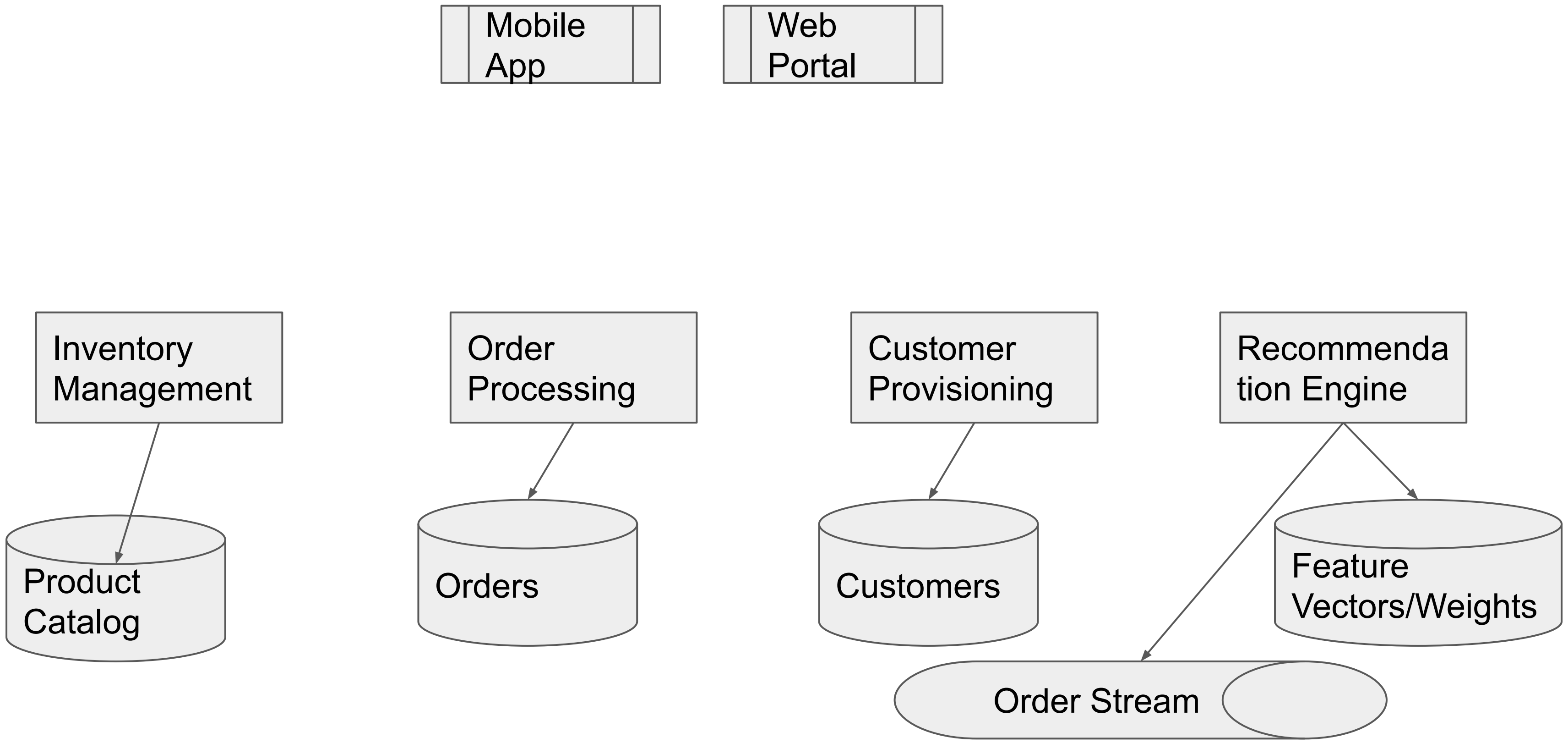
InventoryManagement

Order Processing

Recommendation Engine

Customer Provisioning

Microservices



Create Microservices

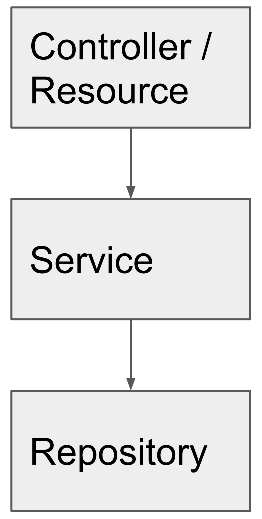
Create Microservices in Spring boot

InventoryManagement

OrderProcessing

Customer Provisioning

Recommendation Engine



Spring Boot

Makes it easy to create stand alone , production-grade applications.
Very little spring configuration required
Opinionated view of the spring platform and the 3rd party libraries required.

Supported embedded servlet containers:

1. Tomcat 9
2. Jetty 9.4
3. Undertow 2.0

Softwares Supported

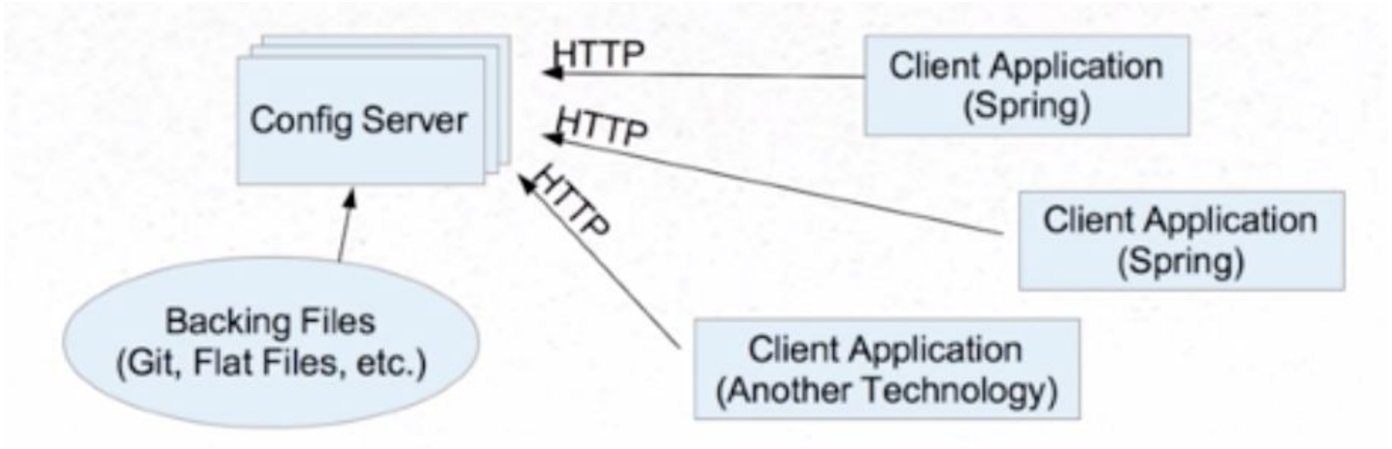
Minimum
Java 1.8 +
Maven 3.3+ or Gradle 4.4+

Spring Boot and REST

- REST capability is built into Spring MVC
- Create domain objects as parameters and return values
- Mark parameters with @RequestBody and return values with @ResponseBody
- Spring MVC automatically handles conversion into JSON

Spring cloud config-centralized config management

- Centralized server that serves configuration information
- Configuration can be backed by property files, database or Git repo
- Clients connect thru HTTP and load the config at startup
- Use Spring Cloud Bus for pushing config changes at runtime. AMQP messaging provider is required.


Config Server
HTTP
HTTP
HTTP
Client Application (Spring)
Client Application (Spring)
Client Application (Another Technology)
Backing Files (Git, Flat Files, etc.)

Service Discovery Server and client-Eureka

Eureka provides a “lookup” server

Made HA by running multiple copies and replicating state of registered services

“Client” services register with Eureka

Client services send heartbeats to Eureka

Spring Cloud Ribbon for Client side load balancing

- Client side load balancing augments load balancing by allowing the client to choose a server based on some criteria specific to client
- Ribbon is an easy to use implementation of client side load balancing
- Low level implementation which introduces coupling between client and server IDs

Spring Cloud Feign: Declarative Rest Client

- Declarative way to call Rest Services
- Alternative to conventional RestTemplate
- Feign integrates with Eureka and Ribbon
  - Eureka gives client Ids for the registering clients
  - Ribbon automatically handles load balancing
  - Feign handles the code

```
@FeignClient(value = "MessageInquiryClient", url = "https://jsonplaceholder.typicode.com")
public interface MessageInquiryClient {

    @GetMapping(value="/posts",consumes= MediaType.APPLICATION_JSON_VALUE)
    List<Message> getMessages();

}
```

Spring Cloud Hystrix- Circuit breaker pattern

Easy to use circuit breaker

Detects failure conditions and “opens” to prevent further calls

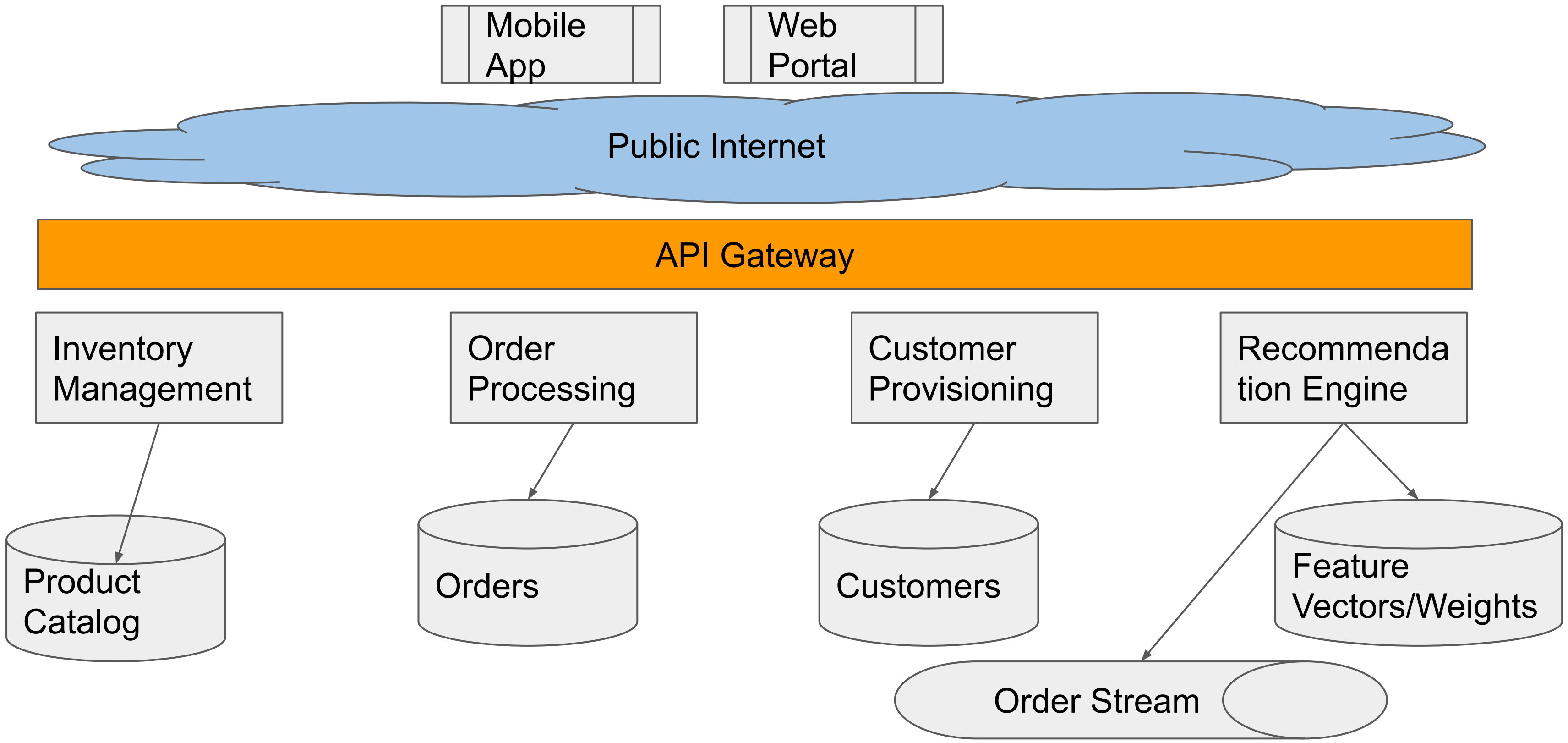
Identify fallback response for failure of service dependencies

```
@SpringBootApplication
@EnableFeignClients
@EnableCircuitBreaker
public class FeignclientwithhystrixApplication {

	public static void main(String[] args) {
		SpringApplication.run(FeignclientwithhystrixApplication.class, args);
	}

}
```

API Gateway with Zuul



Features of API Gateway

- Built for specific client requirements
- Reduces remote calls using composite services
- Routes calls to specific servers
- Handles caching
- Protocol translation

Zuul - Routing and filtering

JVM based router and load balancer

- Supports many API gateway features
- Routing to real server
- Basic usage:
    - Enable Eureka client
    - Enable Zuul proxy
    - Default behavior: Eureka Client Ids become URIs

Spring Cloud Recap

Config- External config management

Eureka- Service Discovery

Hystrix- Circuit Breaker for resiliency

Feign- Declarative service invocation with client side load balancing

Ribbon- Client load balancer

Zuul - Service Routing

Containerization

Docker Ecosystem

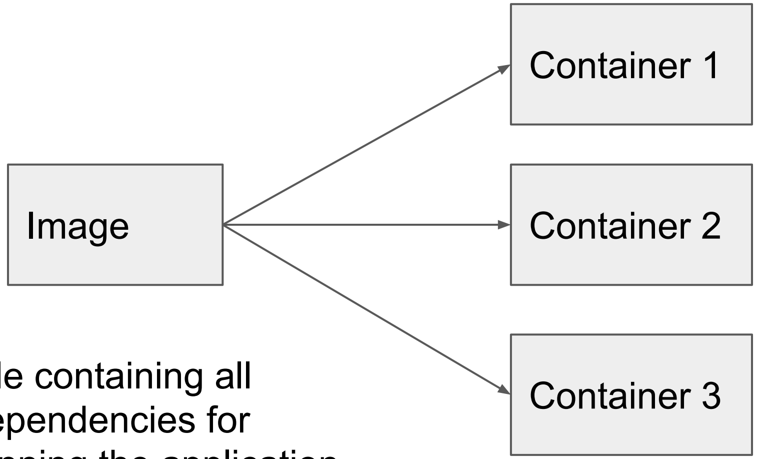
Docker Client

Image Registry

Docker Daemon

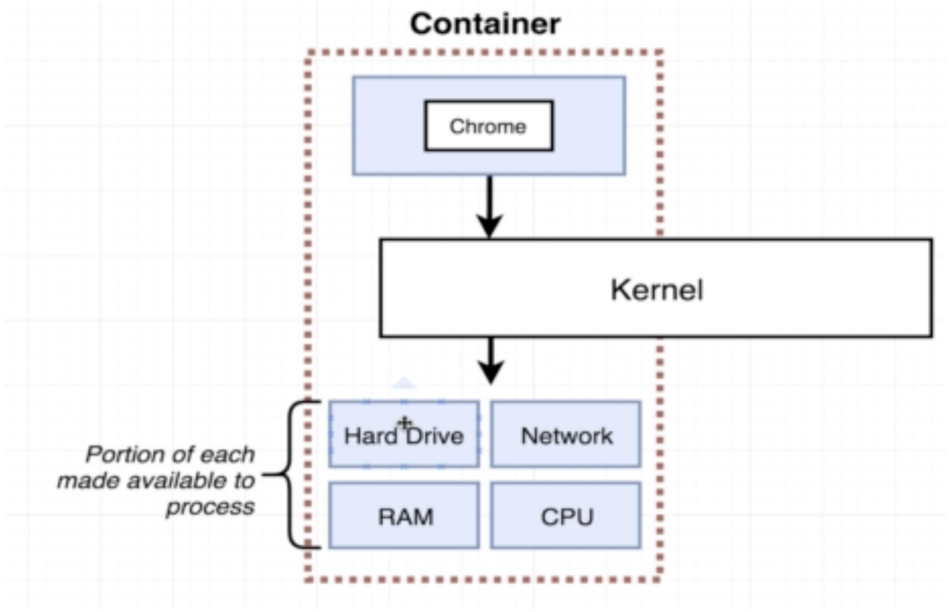
Docker Compose

Image and Container

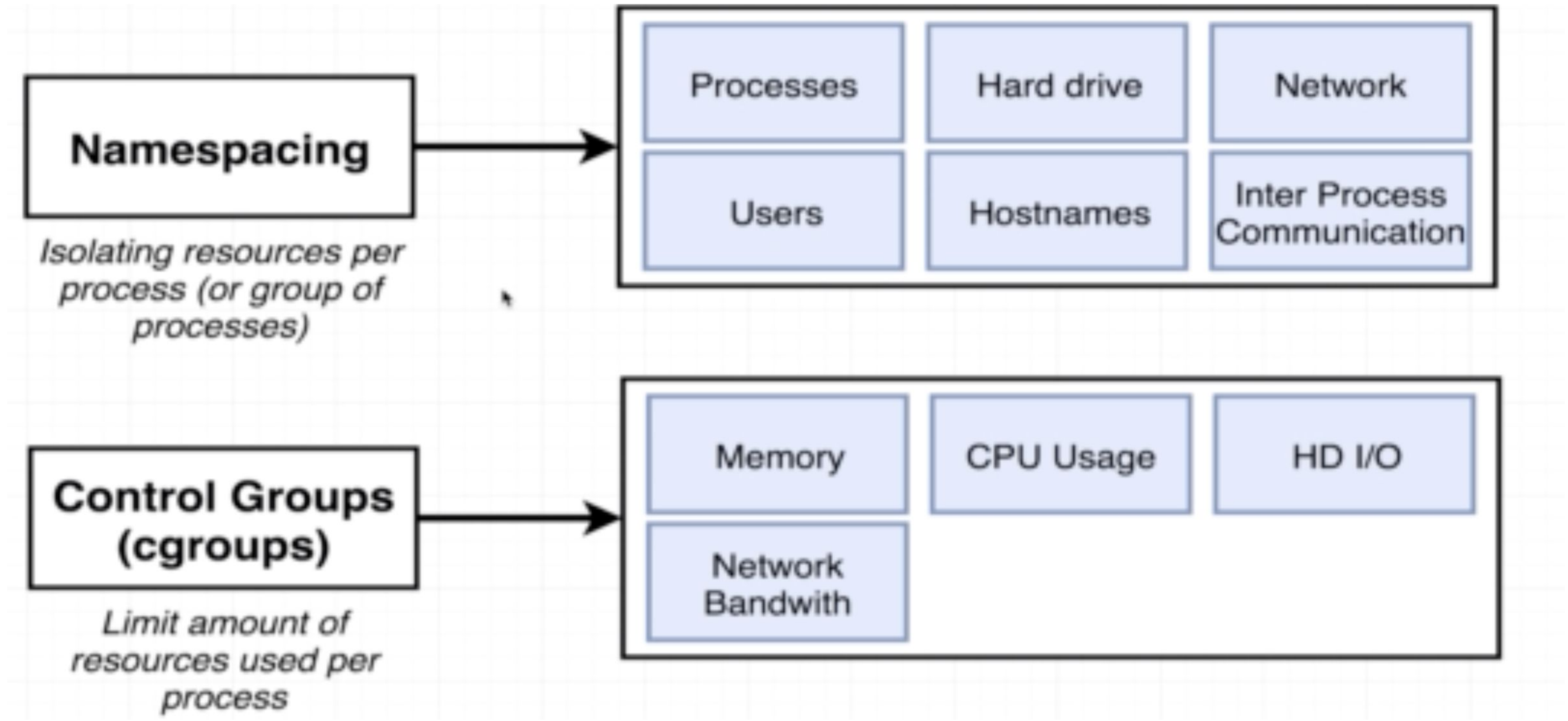


File containing all dependencies for running the application

Running Instance of an image



Docker- Core concept


Namespacing
Isolating resources per process (or group of processes)
Processes
Hard drive
Network
Users
Hostnames
Inter Process Communication
Control Groups (cgroups)
Limit amount of resources used per process
Memory
CPU Usage
HD I/O
Network Bandwith

Containerize with Docker

Build the docker image

```
docker build --file=Dockerfile --tag=inventorymanagement:latest --rm=true .
```

Run the container

```
docker run  --publish=<hostport>:<exposed port>
--volume=/Users/fab/Documents/pratik/tmp:/tmp inventorymanagement:latest
```

Container Orchestration

Kubernetes

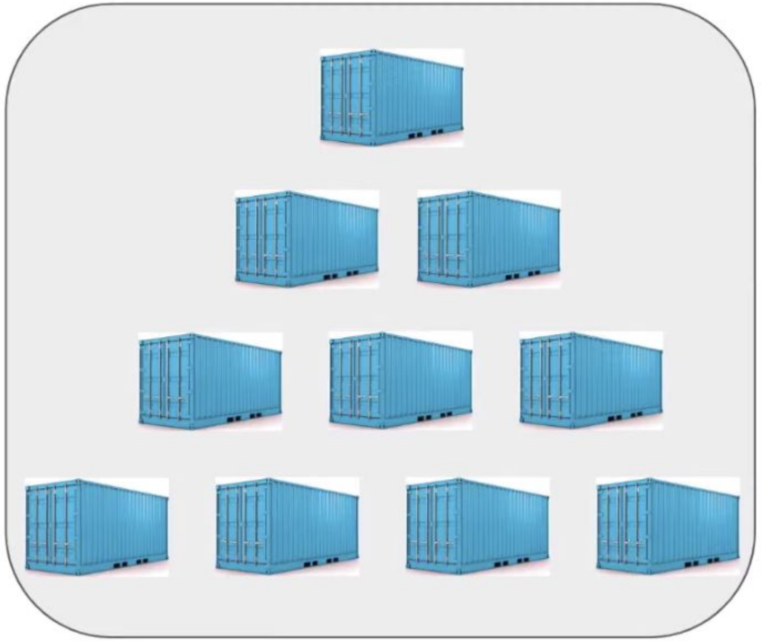
Docker Swarm

Apache Mesos
Marathon

⇨

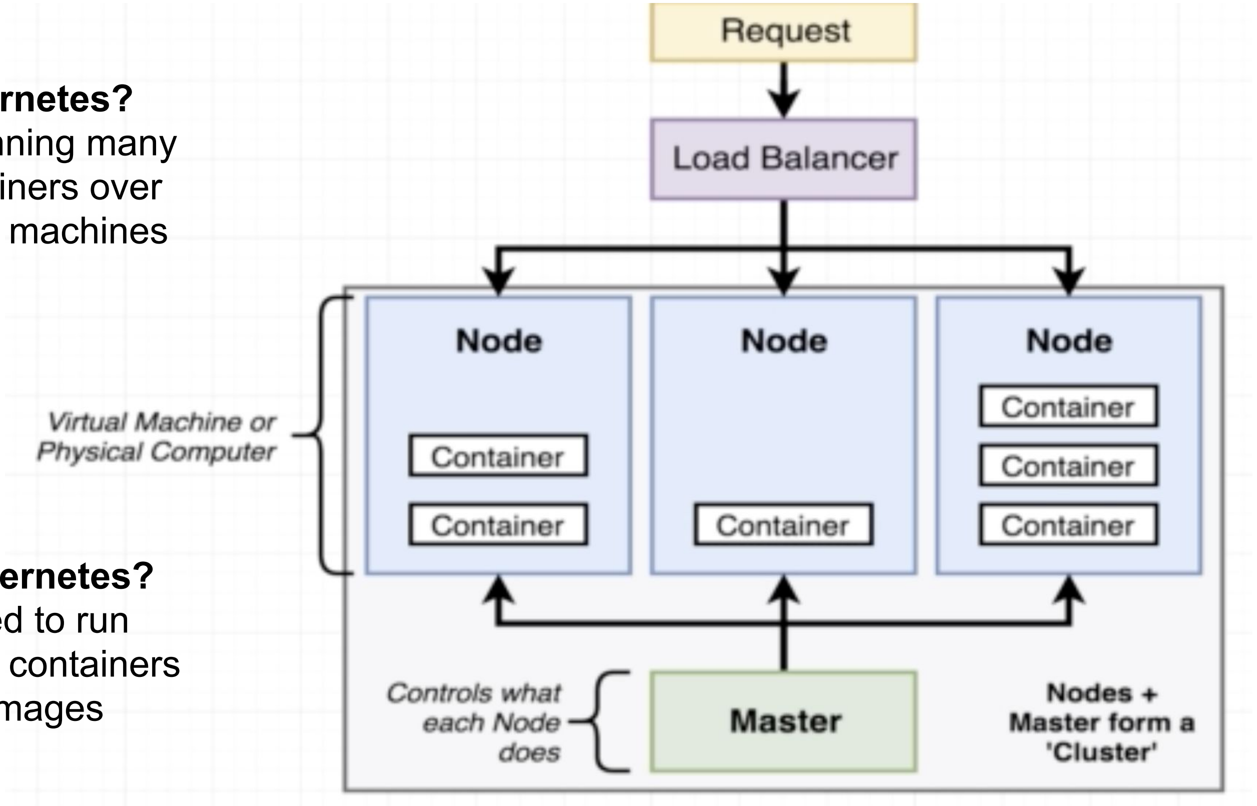
- deploying
- scheduling
- scaling
- load balancing
- batch execution
- rollbacks
- monitoring

⇨



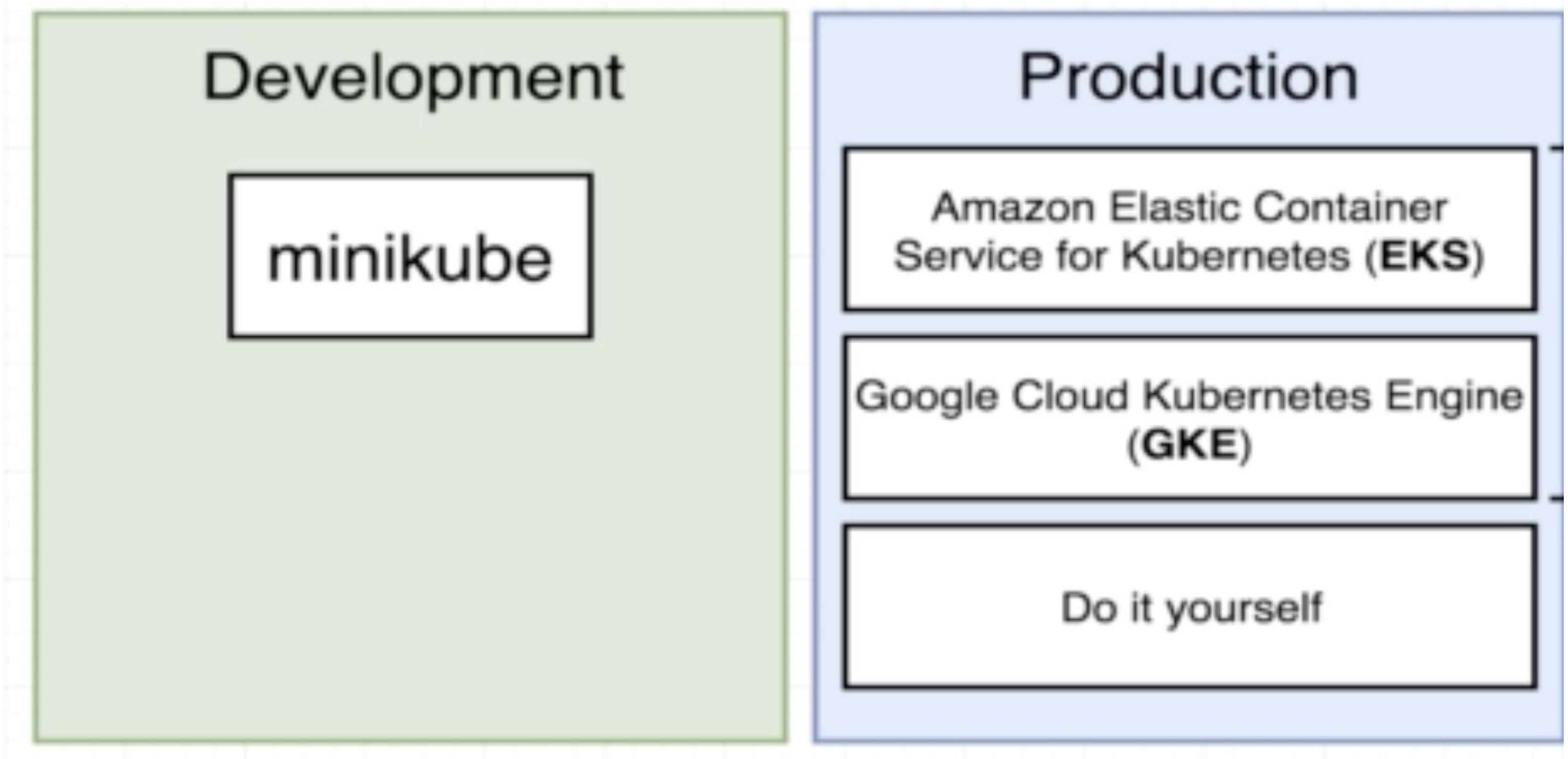
Kubernetes - Cluster of containers

What is Kubernetes?
System for running many different containers over many different machines

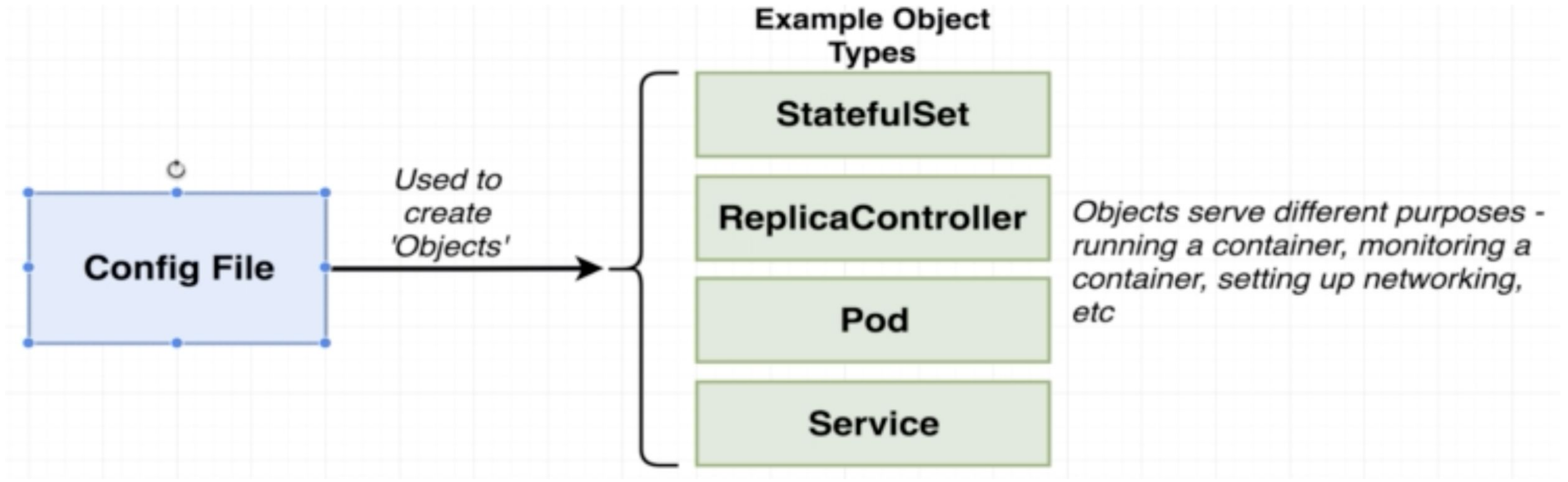

Request
Load Balancer
Node
Node
Node
Container
Container
Container
Container
Container
Container
Virtual Machine or Physical Computer
Controls what each Node does
Master
Nodes + Master form a 'Cluster'

Why use Kubernetes?
When you need to run many different containers with different images

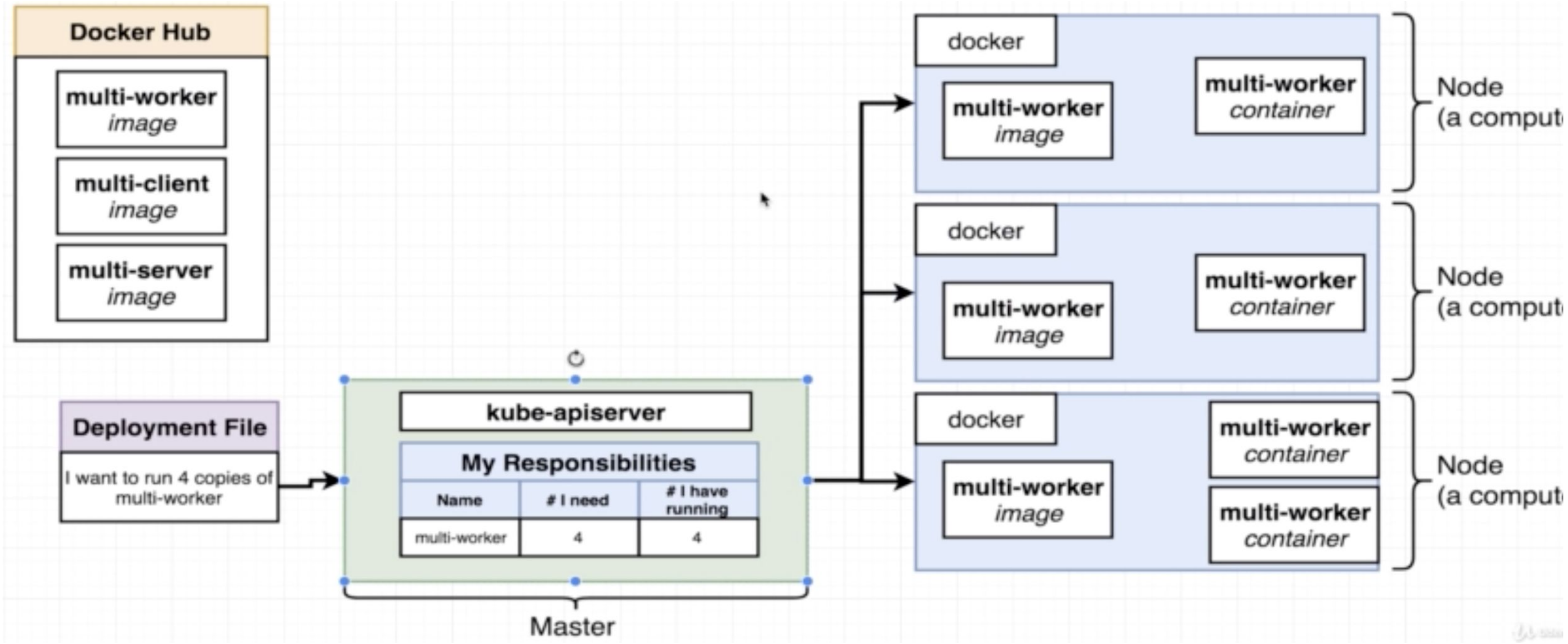
Working with Kubernetes


Development
minikube
Production
Amazon Elastic Container Service for Kubernetes (EKS)
Google Cloud Kubernetes Engine (GKE)
Do it yourself

Kubernetes - Object types


Example Object Types
Config File
Used to create 'Objects'
StatefulSet
ReplicaController
Pod
Service
Objects serve different purposes - running a container, monitoring a container, setting up networking, etc

Deployment Flow



Name	# I need	# I have running
multi-worker	4	4

Deploy in cluster

Push the image to registry

Create a container cluster

```
gcloud container clusters create my-cluster --num-nodes=3 --machine-type=f1-micro
gcloud compute instances list
```

Create Deployment

```
kubectl run hello-web --image=gcr.io/${PROJECT_ID}/hello-app:v1 --port 8080
kubectl get pods -o wide
```

Create Service

```
kubectl expose deployment hello-web --type=LoadBalancer --port 80 --target-port 8080
kubectl get service
```

Scale the deployment

Scale up the application

```
kubectl scale deployment hello-web --replicas=4
kubectl get deployment hello-web
```

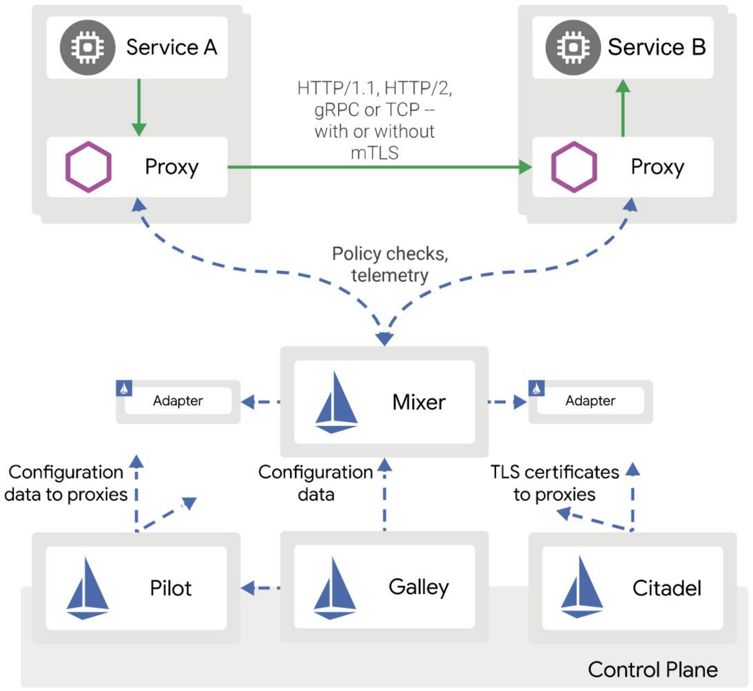
Service Mesh - Istio

Service Mesh is used to describe a network of microservices and communication between them

Istio deploys a special sidecar proxy that intercepts all network communication between microservices, These are configured and managed using its control plane functionality, which includes:

- Automatic load balancing for HTTP, gRPC, WebSocket, and TCP traffic.
- Fine-grained control of traffic behavior with rich routing rules, retries, failovers, and fault injection.
- A pluggable policy layer and configuration API supporting access controls, rate limits and quotas.
- Automatic metrics, logs, and traces for all traffic within a cluster, including cluster ingress and egress.
- Secure service-to-service communication in a cluster with strong identity-based authentication and authorization.

Istio Architecture



Istio Architecture

Docker and Kubernetes Recap

Containers are packaged in a pod

Pod is unit of deployment in Kubernetes

Pods are exposed through a service

Deployment Object is used to manage release of applications