Foundational Patterns

Kubernetes

Francesco Donzello

About me

Francesco Donzello

 @frabird
 @francescodonzello
 francesco.donzello@gmail.com

Introduction

In OOP we have objects, classes, methods, inheritance, encapsulation, package. We use Java to manage the lifecycle of the classes, objects, and the whole project. Along with Java, the JVM offers us the fundamentals for creating an application successfully.

Kubernetes adds an entirely new dimension by offering a new set of **distributed primitives** to implement the whole application behaviour.

Distributed Primitives

Users leverage Kubernetes APIs for the creation, scaling and termination of applications on the platform.

Kubernetes manages various types of **objects**, each targeted by a different operation.

Objects constitute the basic **building blocks** of Kubernetes, availed as primitives for managing containerized applications.

Distributed Primitives

In summary, below are the most important Kubernetes API objects:

- Clusters
- Nodes
- Namespaces
- Pods
- Labels and selectors
- Services
- Replication set
- Deployment

Distributed Primitives

We can compare Distributed Primitives with Java Local Primitives:

- class == Container Image
- object == Container
- .jar == Container Image
- Module, Package == Namespace, Pod
- ThreadPoolExecutor == Job
- Timer == CronJob
- System.getenv() == ConfigMap, Secret

Distributed Primitives: containers

Containers are the building blocks for Kubernetes-based cloud-native applications.

If we make a comparison with OOP and Java, **container images** are like **classes**, and **containers** are like **objects**.

The same way we can extend classes to reuse and alter behavior, we can have container images that **extend** other container images to reuse and alter behavior.

Distributed Primitives: containers

A container image is the unit of functionality that addresses a single concern.

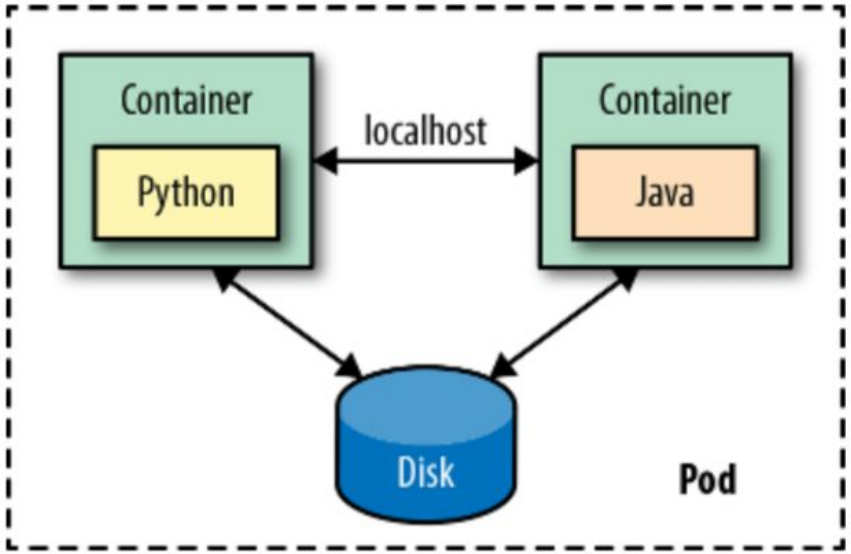
- A container image is owned by one team and has a **release cycle**.
- A container image is **self-contained** and defines and carries its runtime dependencies.
- A container image is **immutable**, and once it is built, it does not change; it is **configured**.
- A container image has defined **runtime dependencies** and **resource requirements**.
- A container image has well-defined APIs to **expose** its functionality.
- A container runs typically as a single Unix **process**.
- A container is disposable and safe to **scale** up or down at any moment.

Distributed Primitives: pods

A Pod is the **smallest** and **simplest** Kubernetes object.

It is the **unit of deployment** in Kubernetes, which represents a single instance of the application.

Distributed Primitives: pods



Distributed Primitives: pods

A Pod is a logical collection of one or more containers, which:

- Are **scheduled** together on the same host
- **Share** the same network namespace
- **Mount** the same external storage (volumes).

Distributed Primitives: pods

Pods are **ephemeral** in nature and they do not have the capability to self-heal by themselves.

That is why we use them with **controllers**, which can handle a Pod's replication, fault tolerance, self-heal, etc.

Examples of controllers are Deployments, ReplicaSets, ReplicationControllers, etc.

Distributed Primitives: pods

Each Pod is meant to run a **single instance** of a given application.

If you want to **scale** your application horizontally (e.g., run multiple instances), you should use multiple Pods, one for each instance.

In Kubernetes, this is generally referred to as **replication**.

Replicated Pods are usually created and managed as a group by an abstraction called a Controller.

Distributed Primitives: pods

A Pod IP address is known only after it is scheduled and started on a node.

A Pod can be rescheduled to a different node if the existing node it is running on is no longer healthy.

All that means is the Pod’s network address may change over the life of an application, and there is a need for another primitive for discovery and load balancing.

Distributed Primitives: pods

A Pod can be used in two main ways:

- **Pods that run a single container:** The “one-container-per-Pod” model is the most common Kubernetes use case; in this case, you can think of a Pod as a wrapper around a single container, and Kubernetes manages the Pods rather than the containers directly.

Distributed Primitives: pods

A Pod can be used in two main ways:

- **Pods that run multiple containers that need to work together:** These colocated containers might form a single cohesive unit of service.

  The Pod wraps these containers and storage resources together as a single manageable entity

Distributed Primitives: pods

kubectl get pods

or

curl -v localhost:8001/api/v1/pods

Distributed Primitives: pods

kubectl get pods

```
NAME                                READY   STATUS             RESTARTS   AGE
account-gw-59fd76d6b4-lv7wd         0/1     ImagePullBackOff   0          39d
admin-server-779bb85ff7-nd7zg       0/1     ImagePullBackOff   0          39d
authentication-ms-74485f8c84-f27lh  0/1     ImagePullBackOff   0          39d
bonus-gw-94bbd8c56-txg8m            0/1     ImagePullBackOff   0          39d
bonus-ms-6f6bc955fb-sfxz8           0/1     ImagePullBackOff   0          39d
config-service-5dc7575759-lfmjs     0/1     ImagePullBackOff   0          39d
discovery-service-77d6448c98-mzcm7  0/1     ImagePullBackOff   0          39d
gateway-service-85f757bbb4-z6jv5    0/1     ImagePullBackOff   0          39d
```

Distributed Primitives: pods

kubectl get pods **-o json**

Distributed Primitives: pods

kubectl get pods **--watch**

Distributed Primitives: pod definition

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
    - name: nginx
      image: nginx
```

Distributed Primitives: services

Service represents a **named entry point** for accessing an application.

Distributed Primitives: services

A Service binds its name to an IP address and port number **permanently**.

The Service is a generic primitive, and it may also point to functionality provided **outside** the Kubernetes cluster.

Distributed Primitives: services

kubectl get svc

```
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   39d
```

Distributed Primitives: labels

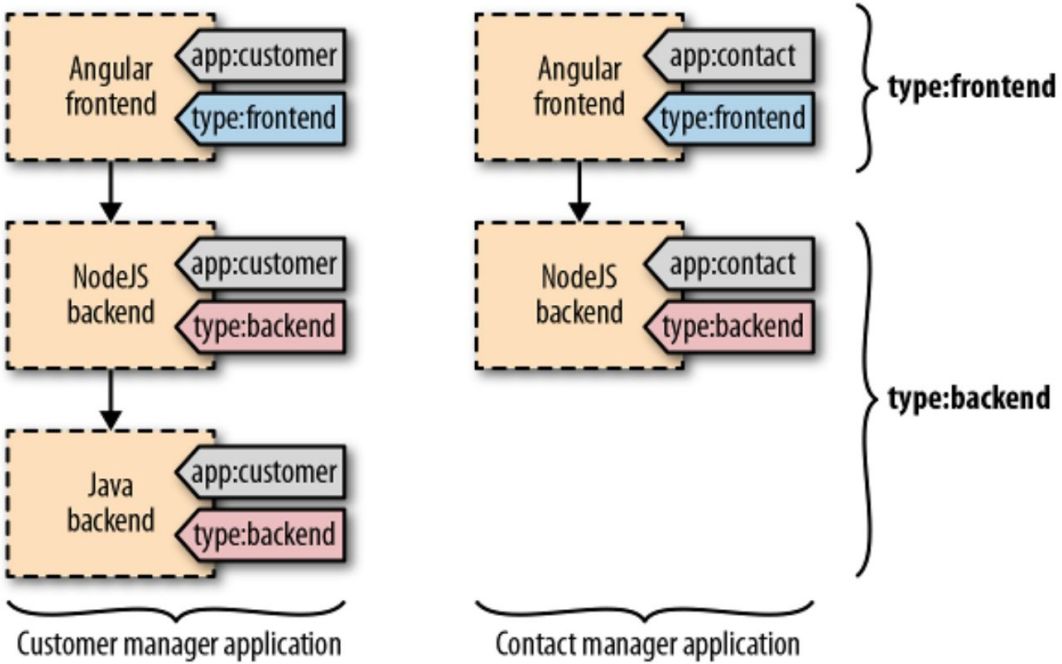
Labels are key/value pairs that are attached to objects, such as pods.

Labels are intended to be used to specify identifying attributes of objects that are meaningful and relevant to users, but do not directly imply semantics to the core system.

Labels can be used to organize and to select subsets of objects. Labels can be attached to objects at creation time and subsequently added and modified at any time.

Each Key must be unique for a given object.

Distributed Primitives: labels



Distributed Primitives: labels

kubectl get pods **-l type=backend**

Distributed Primitives: labels

kubectl get pods **-l type=backend,stage=test**

Distributed Primitives: labels

kubectl get pods **-L=tier**

```
NAME                                READY   STATUS             RESTARTS   AGE   TIER
account-gw-59fd76d6b4-lv7wd         0/1     ImagePullBackOff   0          39d   ms
admin-server-779bb85ff7-nd7zg       0/1     ImagePullBackOff   0          39d   ms
authentication-ms-74485f8c84-f27lh  0/1     ImagePullBackOff   0          39d   ms
bonus-gw-94bbd8c56-txg8m            0/1     ImagePullBackOff   0          39d   ms
bonus-ms-6f6bc955fb-sfxz8           0/1     ImagePullBackOff   0          39d   ms
config-service-5dc7575759-lfmjs     0/1     ImagePullBackOff   0          39d   cloud
discovery-service-77d6448c98-mzcm7  0/1     ImagePullBackOff   0          39d   cloud
gateway-service-85f757bbb4-z6jv5    0/1     ImagePullBackOff   0          39d   cloud
```

Distributed Primitives: labels

- Labels are used by the ReplicaSets to keep some instances of a specific Pod running. That means every Pod definition needs to have a unique combination of labels used for scheduling.
- A Label can indicate a logical grouping of set of Pods and give an application identity to them.
- Labels can be used to store meta-data.

Distributed Primitives: labels definition

```
kubectl label pods bar color=red
```

Distributed Primitives: labels remove

```
kubectl label pods bar color-
```

Distributed Primitives: labels definition

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
    - name: nginx
      image: nginx
```

Distributed Primitives: annotations

Like labels, annotations are organized as a map.

The information on the annotations is not intended for querying and matching objects.

Some examples of using annotations include build IDs, release IDs, image information, timestamps, Git branch names, pull request numbers

Distributed Primitives: annotations definition

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  annotations:
    project: TestX
  labels:
    env: test
spec:
  containers:
    - name: nginx
      image: nginx
```

Distributed Primitives: annotations cmd

kubectl **annotate** pods bar color=yellow

kubectl **annotate** pods bar color-

Distributed Primitives: namespaces

Kubernetes namespaces allow dividing a Kubernetes cluster into a logical pool of resources.

The most common use case of namespaces is representing different software environments such as development, testing, integration testing, or production.

Namespaces can also be used to achieve multitenancy, and provide isolation for team workspaces, projects, and even specific applications.

Distributed Primitives: namespaces

- A namespace is managed as a Kubernetes resource.
- A namespace provides scope for resources such as containers, Pods, Services, or ReplicaSets. The names of resources need to be unique within a namespace, but not across them.
- Each Kubernetes Service belongs to a namespace and gets a corresponding DNS address that has the namespace in the form of <service-name>.<namespace-name>.svc.cluster.local.
- ResourceQuotas provide constraints that limit the aggregated resource consumption per namespace.

Distributed Primitives: namespaces

kubectl get ns

or

curl -v localhost:8001/api/v1/namespaces

```
NAME              STATUS   AGE
default           Active   40d
docker            Active   40d
kube-node-lease   Active   40d
kube-public       Active   40d
kube-system       Active   40d
```

Distributed Primitives: namespaces

kubectl get pods **-n prod**

Distributed Primitives: namespaces

kubectl get pods -n prod **--watch**