Chapter 7.

Kubernetes Building Blocks

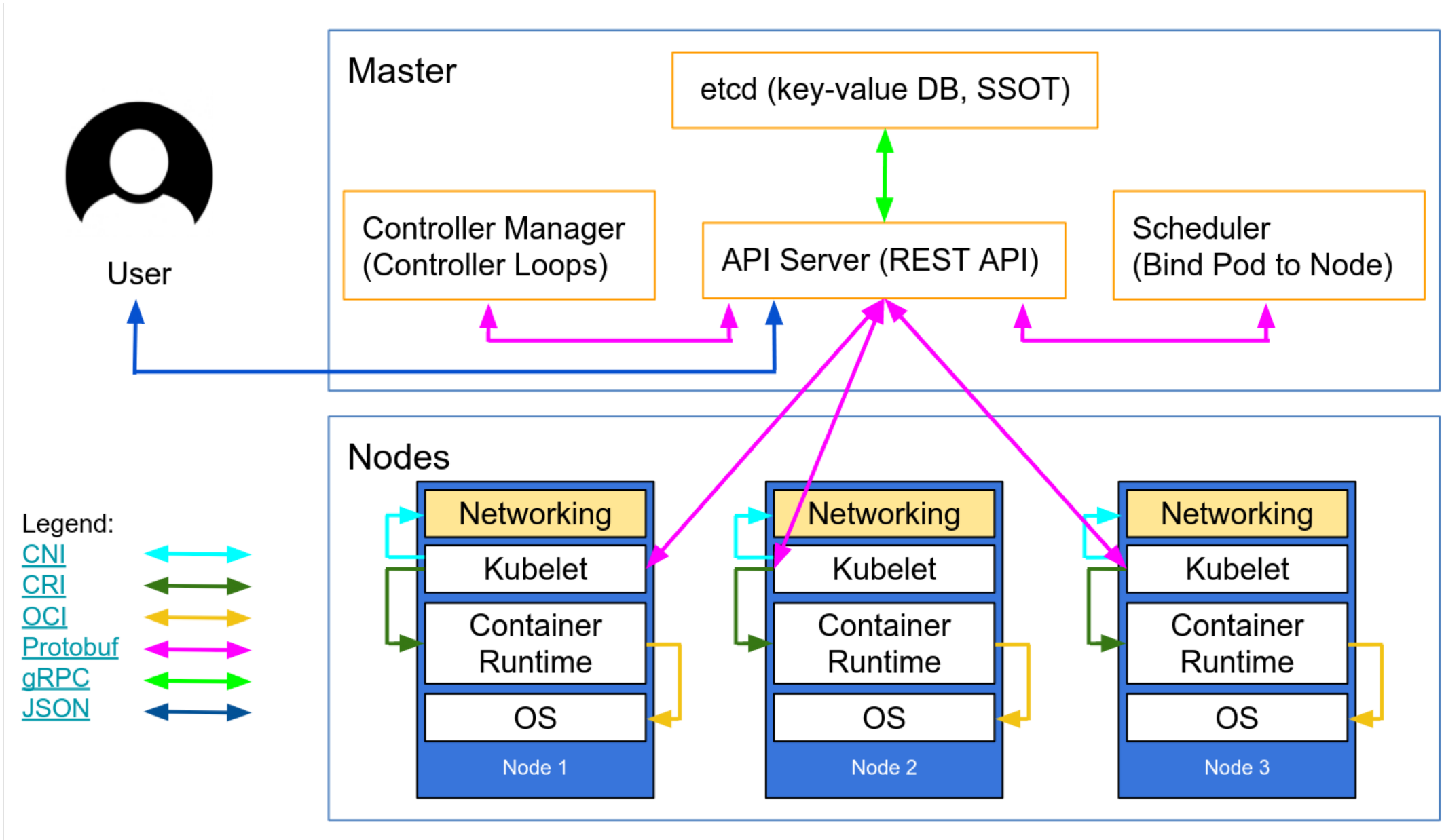
Introduction

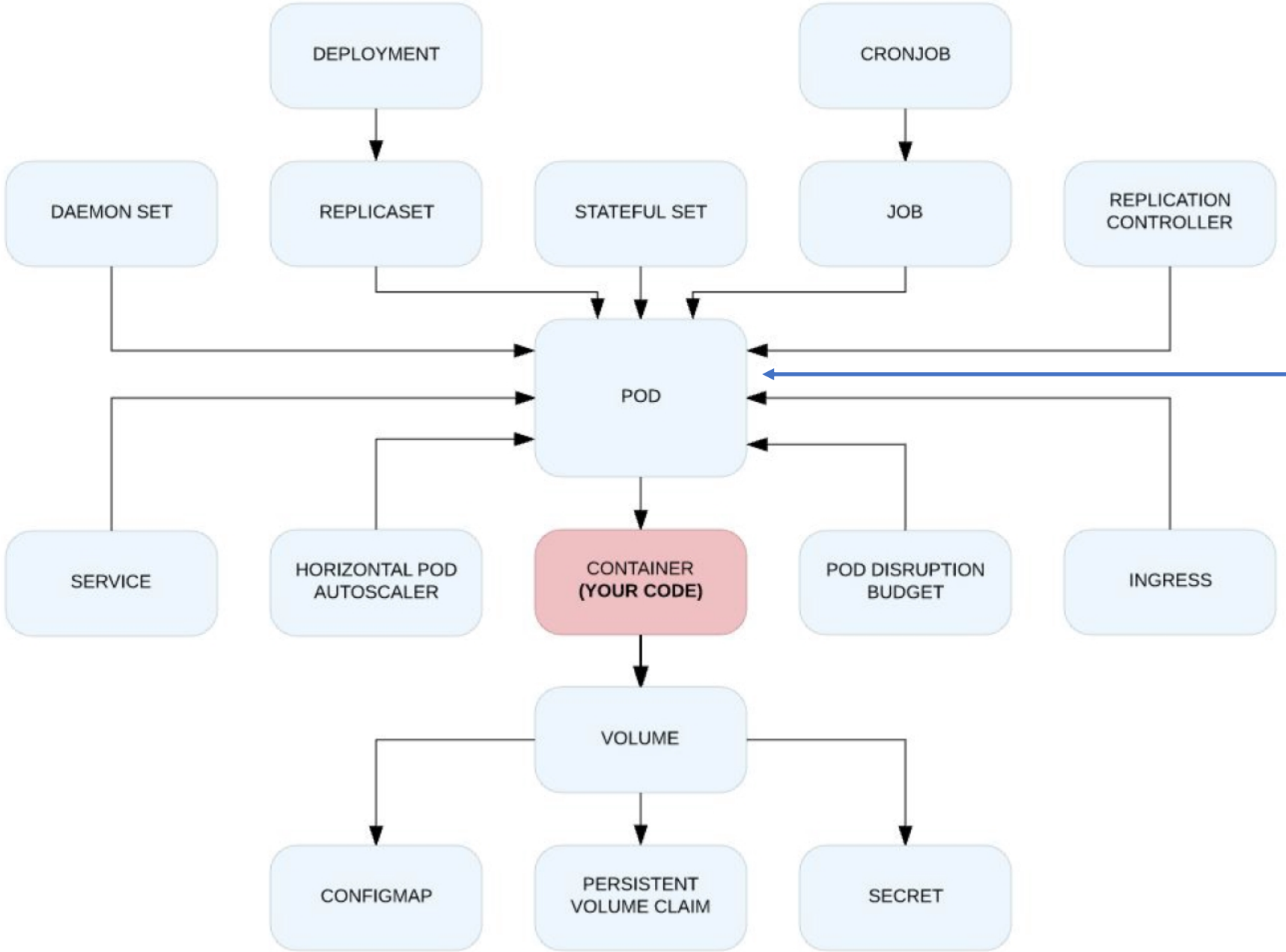
- In this chapter, we will explore the **Kubernetes object** model and discuss some of its building blocks, such as **Pods**, **ReplicaSets**, **Deployments**, **Namespaces**, etc. We will also discuss the role of **Labels and Selectors** when it comes to grouping objects together.

Learning Objectives

By the end of this chapter, you should be able to:

- Review the Kubernetes object model.
- Discuss Kubernetes building blocks, e.g. Pods, ReplicaSets, Deployments, Namespaces.
- Discuss Labels and Selectors.


Master
etcd (key-value DB, SSOT)
Controller Manager
(Controller Loops)
API Server (REST API)
Scheduler
(Bind Pod to Node)
User
Nodes
Networking
Kubelet
Container
Runtime
OS
Node 1
Networking
Kubelet
Container
Runtime
OS
Node 2
Networking
Kubelet
Container
Runtime
OS
Node 3
Legend:
CNI
CRI
OCI
Protobuf
gRPC
JSON



อะไร อะไร ก็ ตรู!

Kubernetes Object Model

Kubernetes has a very rich object model, with which it **represents different persistent entities in the Kubernetes cluster**. Those entities describe:

- **What containerized applications** we are running and on which node
- **Application resource** consumption
- **Different policies** attached to applications, like restart/upgrade policies, fault tolerance, etc.

With each object, we declare our intent or **desired state** using the **`spec`** field. The Kubernetes system manages the **`status`** field for objects, in which it records the **actual state** of the object. At any given point in time, the Kubernetes Control Plane tries to match the object's actual state to the object's desired state.
Examples of Kubernetes objects are **Pods, ReplicaSets, Deployments, Namespaces**, etc.

Kubernetes objects

Primitives

- Namespace
- Node
- Pod
- Service
- Config Map
- Secret
- Volume
- Persistent Volume

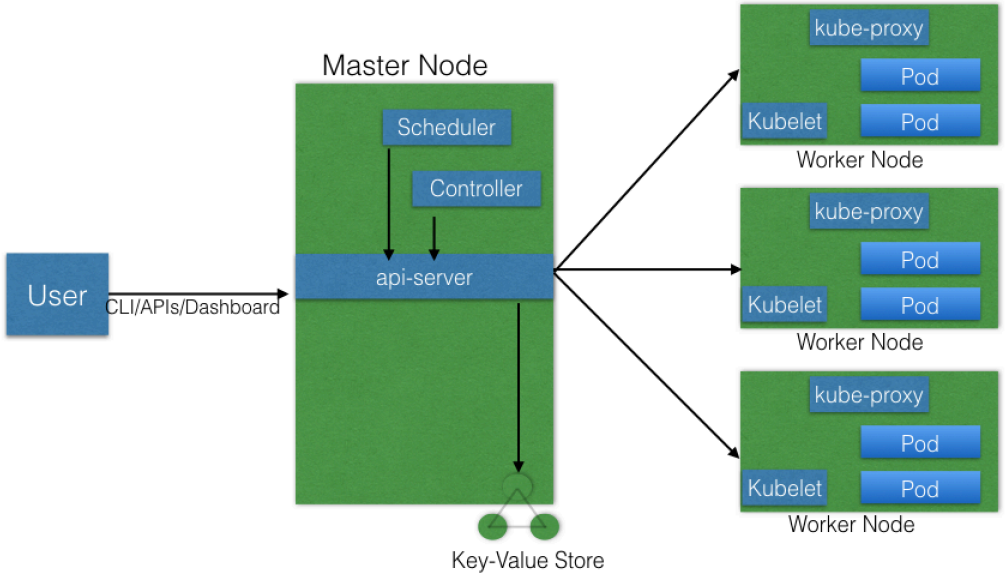
Controllers

- Replication Controller
- Deployment
- Job
- Daemon Set
- Ingress
- ...

EASTBANC TECHNOLOGIES

Kubernetes Object Model

To create an object, we need to provide the **`spec`** field to the Kubernetes API server. The **`spec`** field describes the desired state, along with some basic information, like the name. The API request to create the object must have the **`spec`** field, as well as other details, in a JSON format. Most often, we provide an object's definition in a **`.yaml`** file, which is converted by **`kubectl`** in a JSON payload and sent to the API server.



Below is an example of a Deployment object:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

With the **apiVersion** field in the example above, we mention the API endpoint on the API server which we want to connect to. With the **kind** field, we mention the object type - in our case, we have **Deployment**.

With the **metadata** field, we attach the basic information to objects, like the name. You may have noticed that in our example we have two **spec** fields (**spec** and **spec.template.spec**).

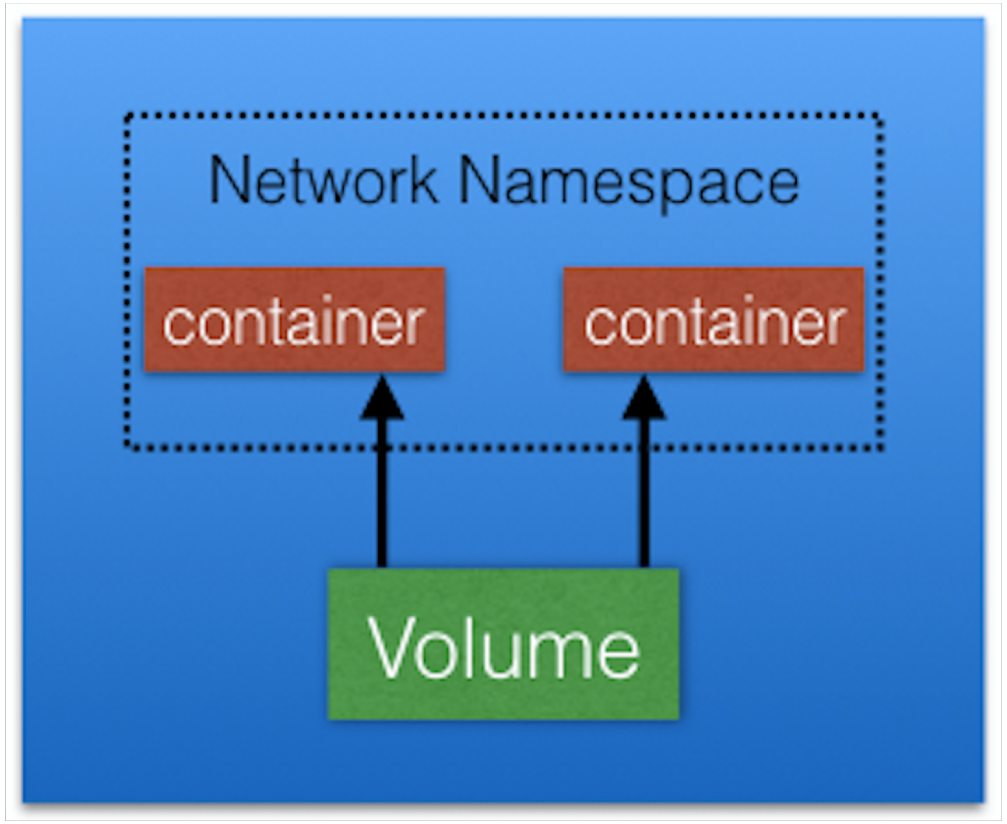
With **spec**, we define the desired state of the deployment. In our example, we want to make sure that, at any point in time, at least 3 Pods are running, which are created using the Pods Template defined in **spec.template**. In **spec.template.spec**, we define the desired state of the Pod. Here, our Pod would be created using **nginx:1.7.9**.

Once the object is created, the Kubernetes system attaches the **status** field to the object.

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

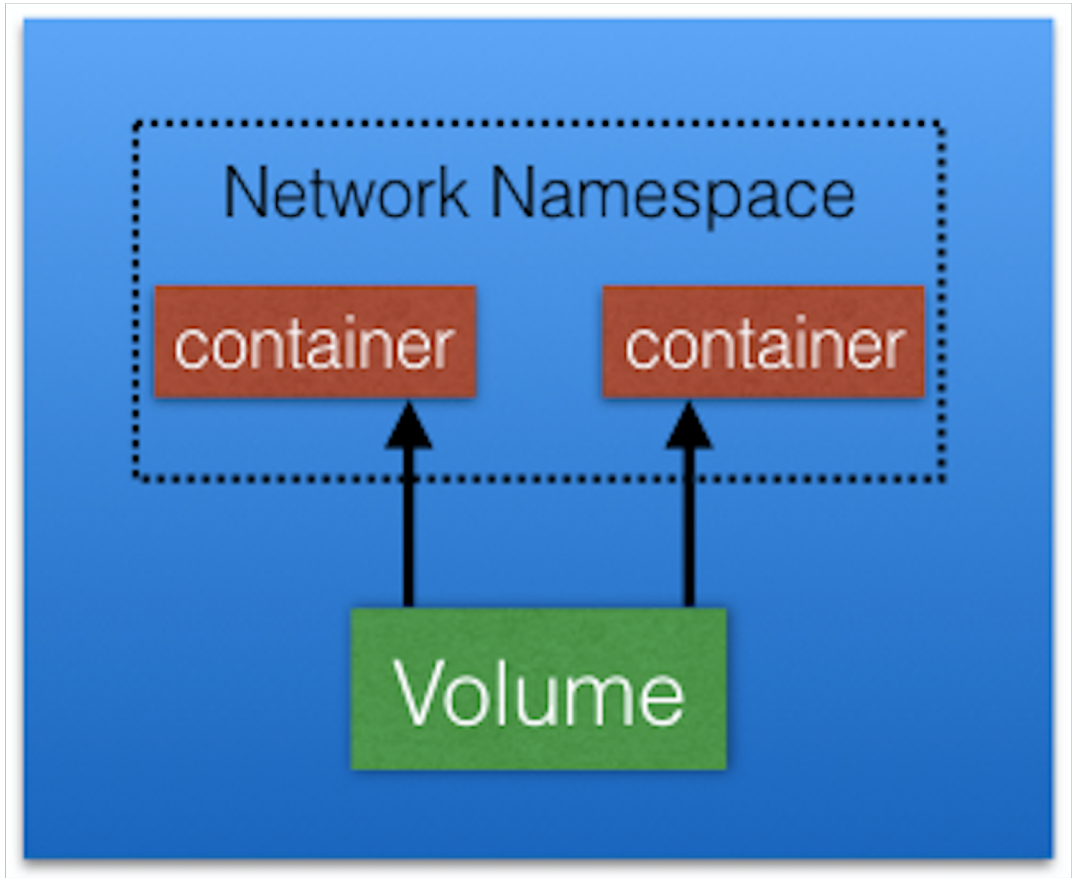
- Are scheduled together on the **same host**
- Share the **same network namespace**
- Mount the **same external storage (volumes).**



Pods

ชั่วคราว

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. We attach the Pod's specification to other objects using Pods Templates, as we have seen in the previous section.



Labels

Labels are key-value pairs that can be attached to **any Kubernetes objects** (e.g. Pods). **Labels are used to organize** and select a subset of objects, based on the requirements in place. Many objects can have the same Label(s). Labels do not provide uniqueness to objects.

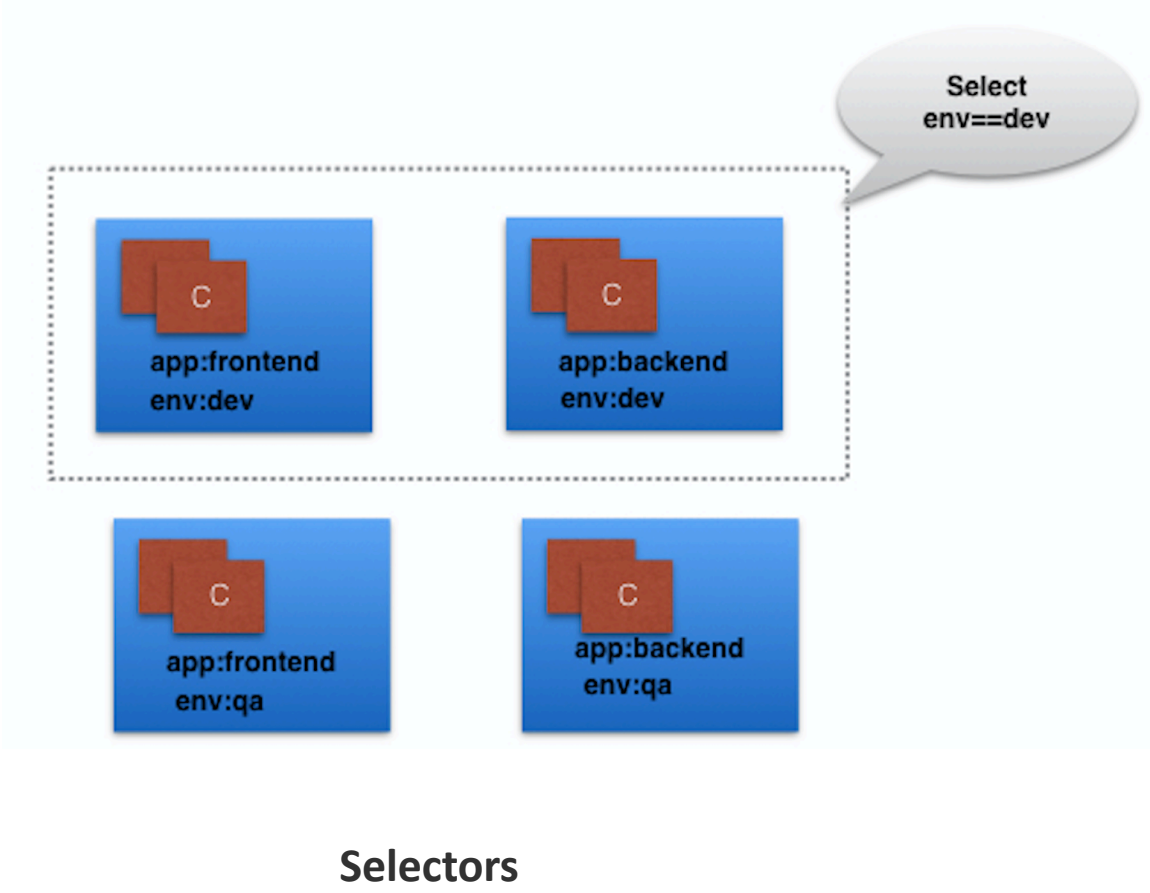
Labels

In the image above, we have used two Labels: **`app`** and **`env`**. Based on our requirements, we have given different values to our four Pods.

Label Selectors

With Label Selectors, we can select a subset of objects.
Kubernetes supports two types of Selectors:

- **Equality-Based Selectors**
  Equality-Based Selectors allow filtering of objects based on Label keys and values. With this type of selectors, we can use the **=**, **==**, or **!=**operators. For example, with **`env==dev`** we are selecting the objects where the **`env`** Label is set to **`dev`**.
- **Set-Based Selectors**
  Set-Based Selectors allow filtering of objects based on a set of values. With this type of Selectors, we can use the **`in`**, **`notin`**, and **`exist`**operators. For example, with **`env in (dev,qa)`**, we are selecting objects where the **`env`** Label is set to **`dev`** or **`qa`**.



Selectors

กิจกรรม Labels / Selectors

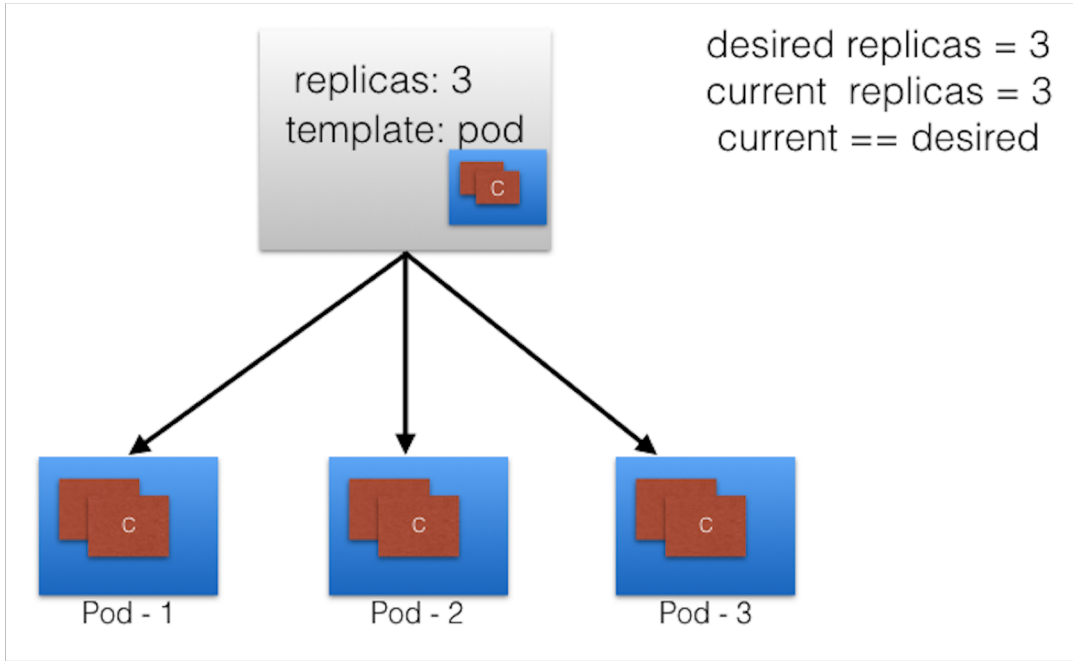
ReplicationControllers

A ReplicationController (rc) is a controller that is **part of the master node's controller manager**. It makes sure the specified number of replicas for a Pod is running at any given point in time. If there are more Pods than the desired count, the ReplicationController would kill the extra Pods, and, if there are less Pods, then the ReplicationController would create more Pods to match the desired count.

Generally, we don't deploy a Pod independently, **as it would not be able to re-start itself, if something goes wrong**. We always use controllers like ReplicationController to create and manage Pods.

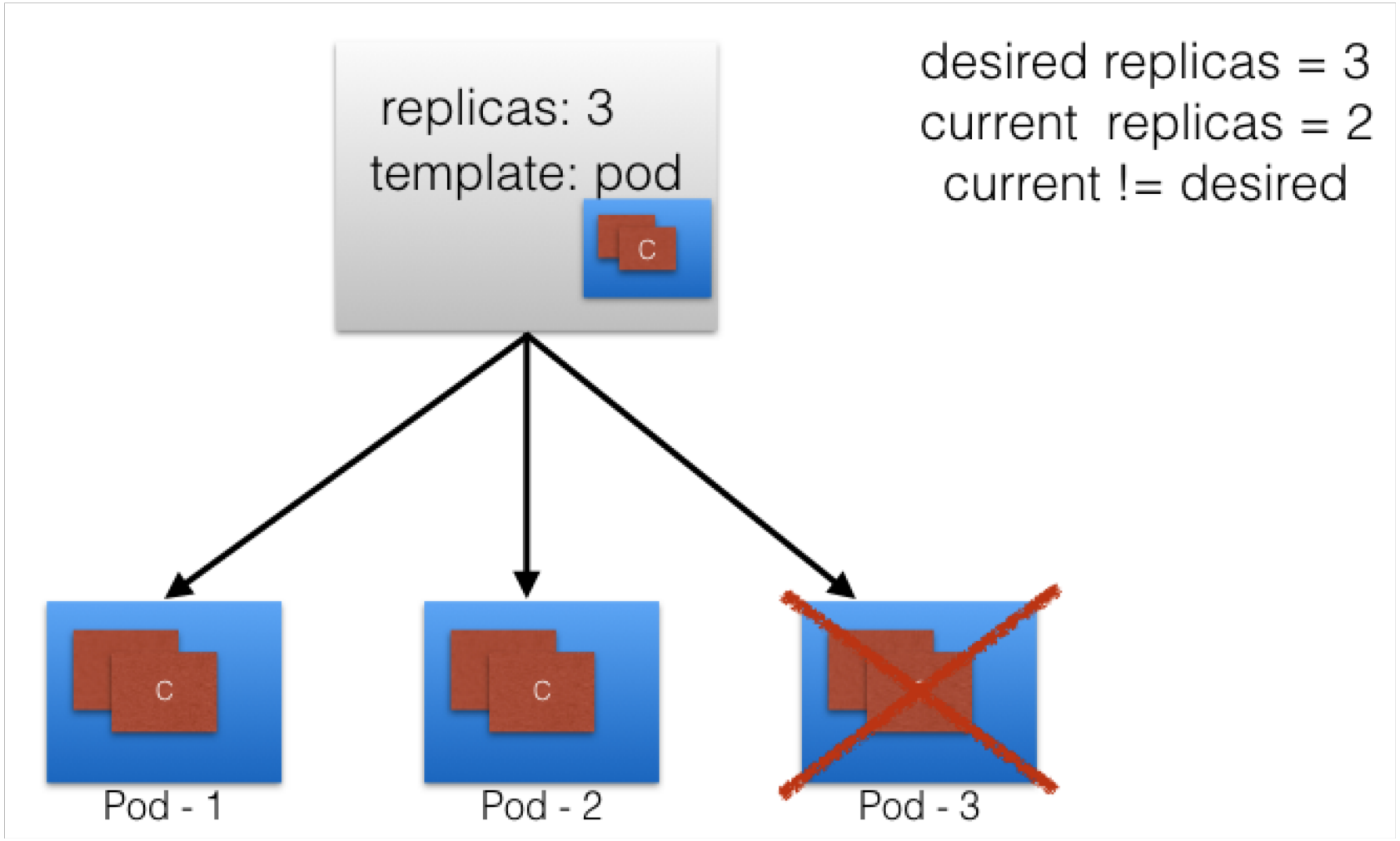
ReplicaSets I

A ReplicaSet (rs) is the **next-generation ReplicationController**. ReplicaSets support **both equality- and set-based selectors**, whereas ReplicationControllers only support equality-based Selectors. Currently, this is the only difference.
Next, you can see a graphical representation of a ReplicaSet, where we have set the replica count to 3 for a Pod.



ReplicaSet (Current State and Desired State Are the Same)

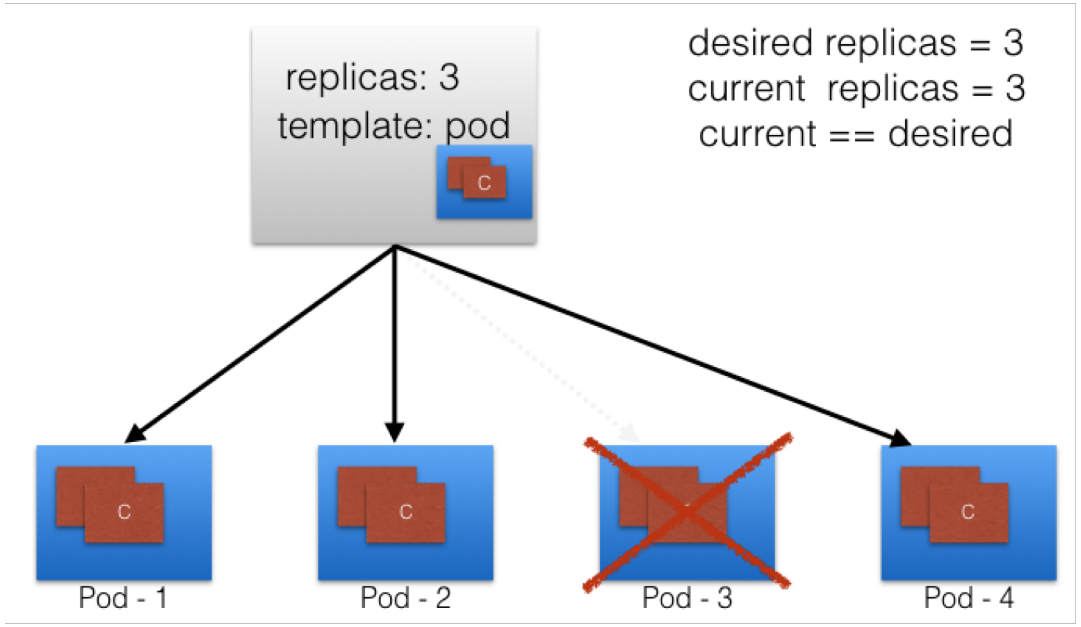
ReplicaSets II



ReplicaSet (Current State and Desired State Are Different)

ReplicaSets III

The ReplicaSet will detect that the current state is no longer matching the desired state. So, in our given scenario, the **ReplicaSet will create one more Pod,** thus ensuring that the current state matches the desired state.



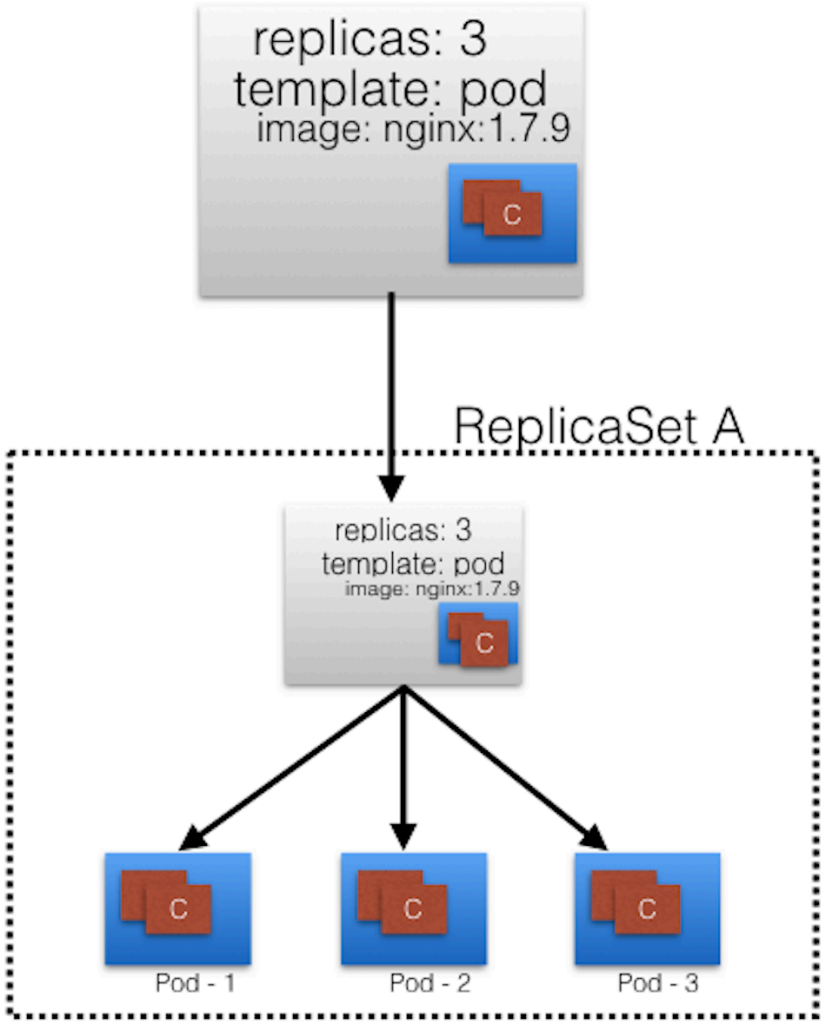
ReplicaSet (Creating a Pod to Match Current and Desired State)

ReplicaSets can be used independently, but they are mostly used by Deployments to orchestrate the Pod creation, deletion, and updates. **A Deployment automatically creates the ReplicaSets**, and we do not have to worry about managing them.

Deployments I

Deployment objects provide **declarative updates to Pods and ReplicaSets**. The **DeploymentController is part of the master node's controller manager**, and it makes sure that the current state always matches the desired state.

In the following example, we have a **Deployment** which creates a **`ReplicaSet A`**. **`ReplicaSet A`** then creates **3 Pods**. In each Pod, one of the containers uses the **`nginx:1.7.9`** image.

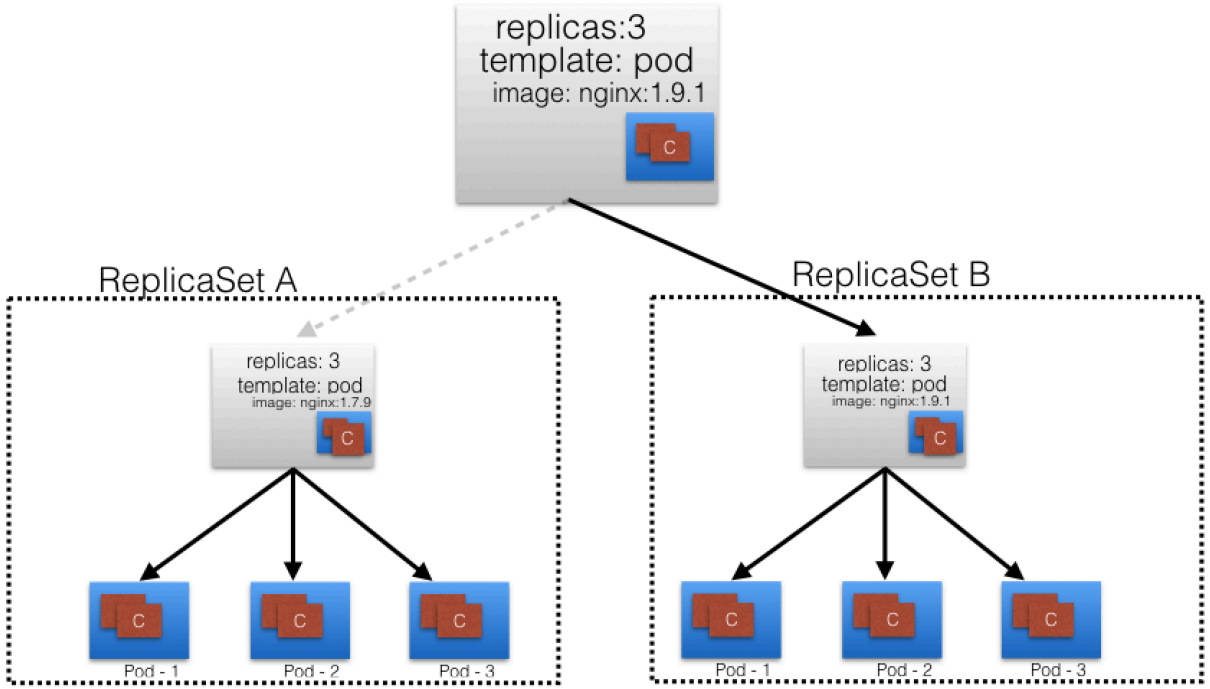


Deployments II

Now, in the Deployment, we change the Pods Template and we update the image for the **`nginx`** container from **`nginx:1.7.9`** to **`nginx:1.9.1`**.

As have modified the Pods Template, a new **`ReplicaSet B`** gets created. This process is referred to as a **Deployment rollout**.

A rollout is only triggered when we update the **Pods Template** for a deployment. Operations like scaling the deployment do not trigger the deployment.

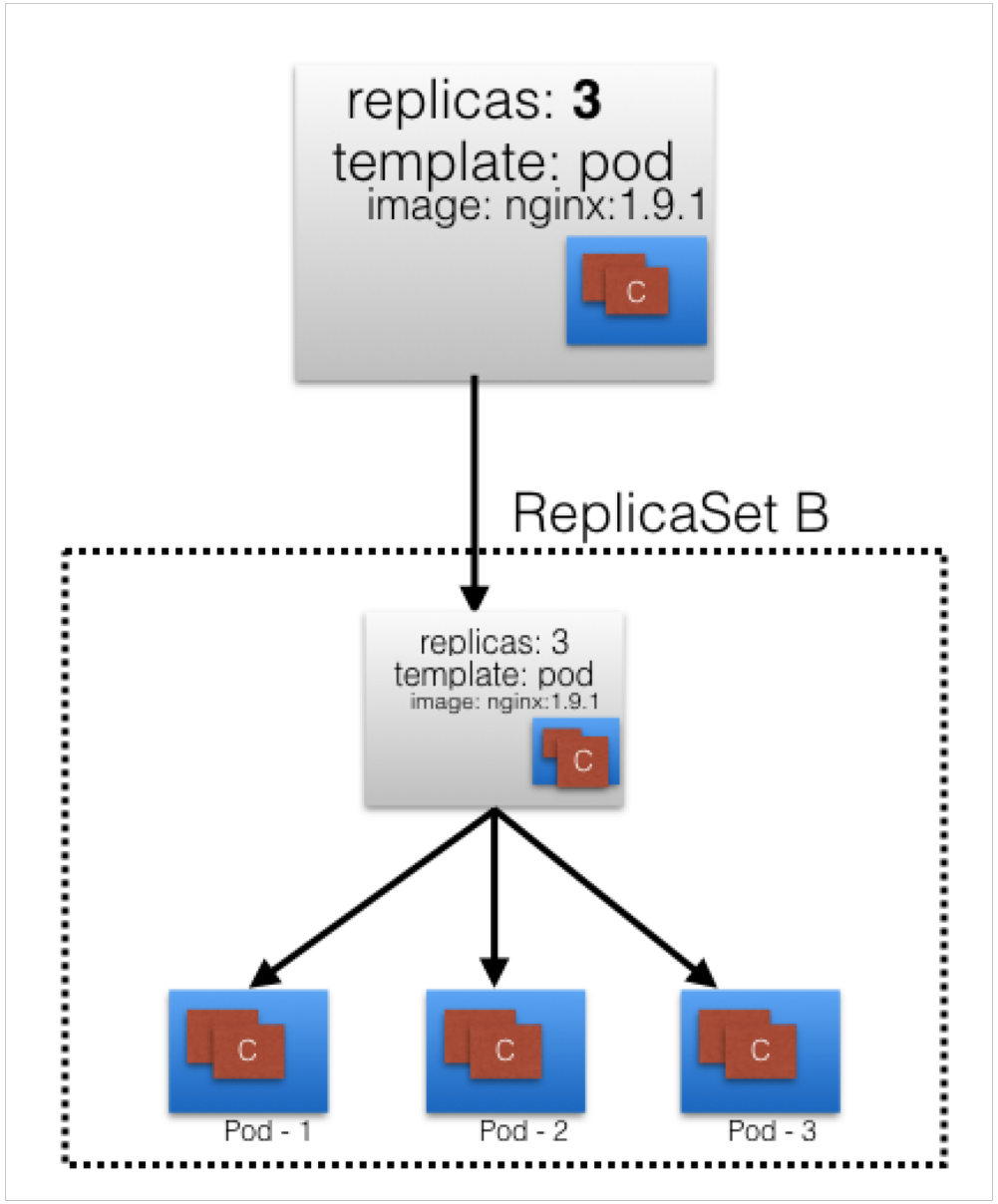


Deployment (ReplicaSet B Created)

Deployments III

Once **`ReplicaSet B`** is ready, the Deployment starts pointing to it.

On top of ReplicaSets, Deployments provide features like Deployment recording, with which, if something goes wrong, we can rollback to a previously known state.



Namespaces

If we have numerous users whom we would like to organize into teams/projects, we can **partition the Kubernetes cluster into sub-clusters** using Namespaces. The names of the resources/objects created inside a Namespace are unique, but not across Namespaces.

To list all the Namespaces, we can run the following command:

```
$ kubectl get namespaces
NAME          STATUS       AGE
default       Active       11h
kube-public   Active       11h
kube-system   Active       11h
```

Generally, Kubernetes creates two default Namespaces: **`kube-system`** and **`default`**. The **`kube-system`** Namespace contains the objects created by the Kubernetes system. The **`default`** Namespace contains the objects which belong to any other Namespace. By default, we connect to the **`default`** Namespace. **`kube-public`** is a special Namespace, which is readable by all users and used for special purposes, like bootstrapping a cluster.

Using Resource Quotas, we can divide the cluster resources within Namespaces. We will briefly cover **resource quotas** in one of the future chapters.

Learning Objectives (Review)

You should now be able to:

- Review the Kubernetes object model.
- Discuss Kubernetes building blocks, e.g. Pods, ReplicaSets, Deployments, Namespaces.
- Discuss Labels and Selectors.

Lab

- Lab 2 - Deploy a test application
- Lab 3 - Deploy a test application 2