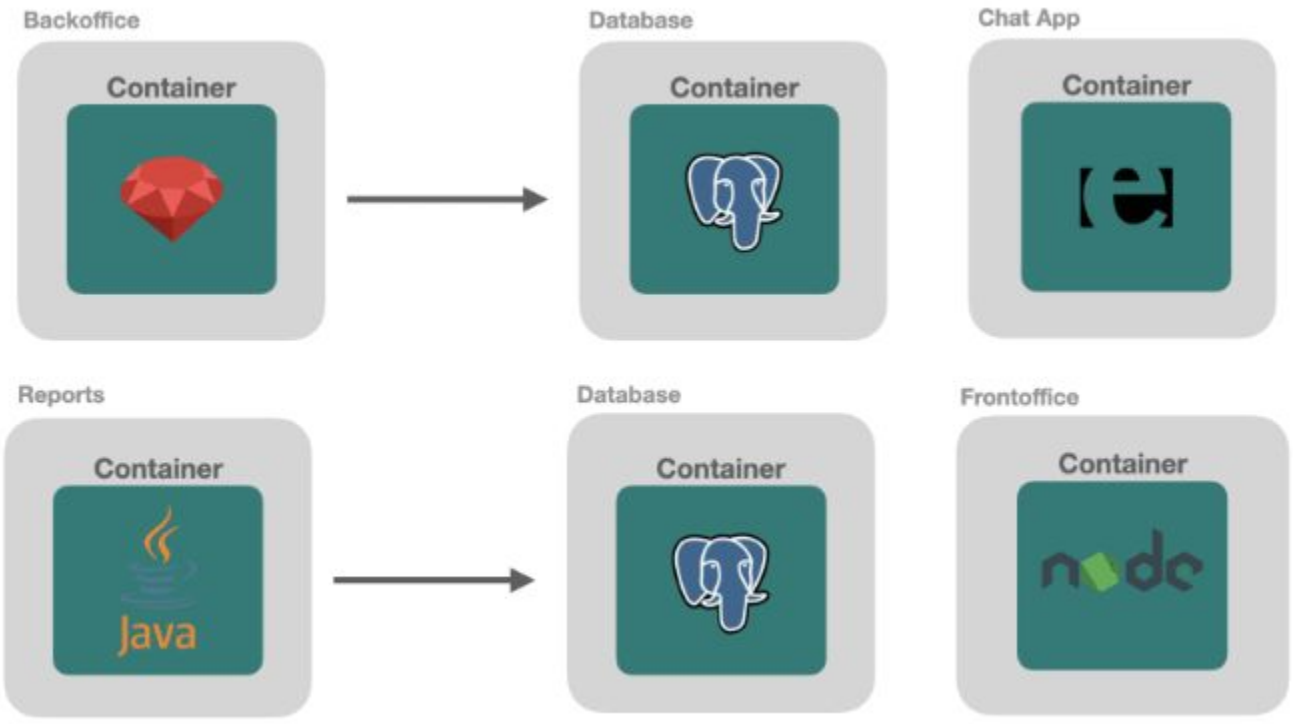
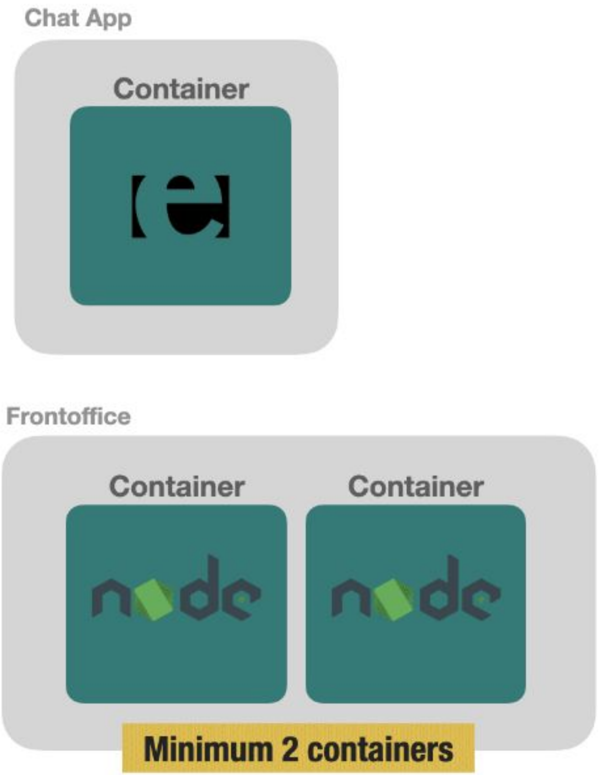
Kubernetes 101 Tutorial - 0

03/19/2023

A Sample Enterprise System

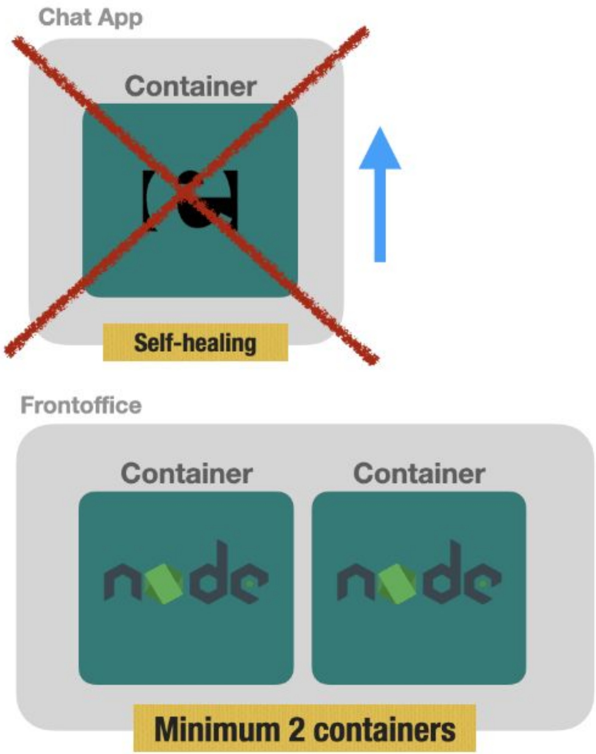


Ensure maximum availability and scalability

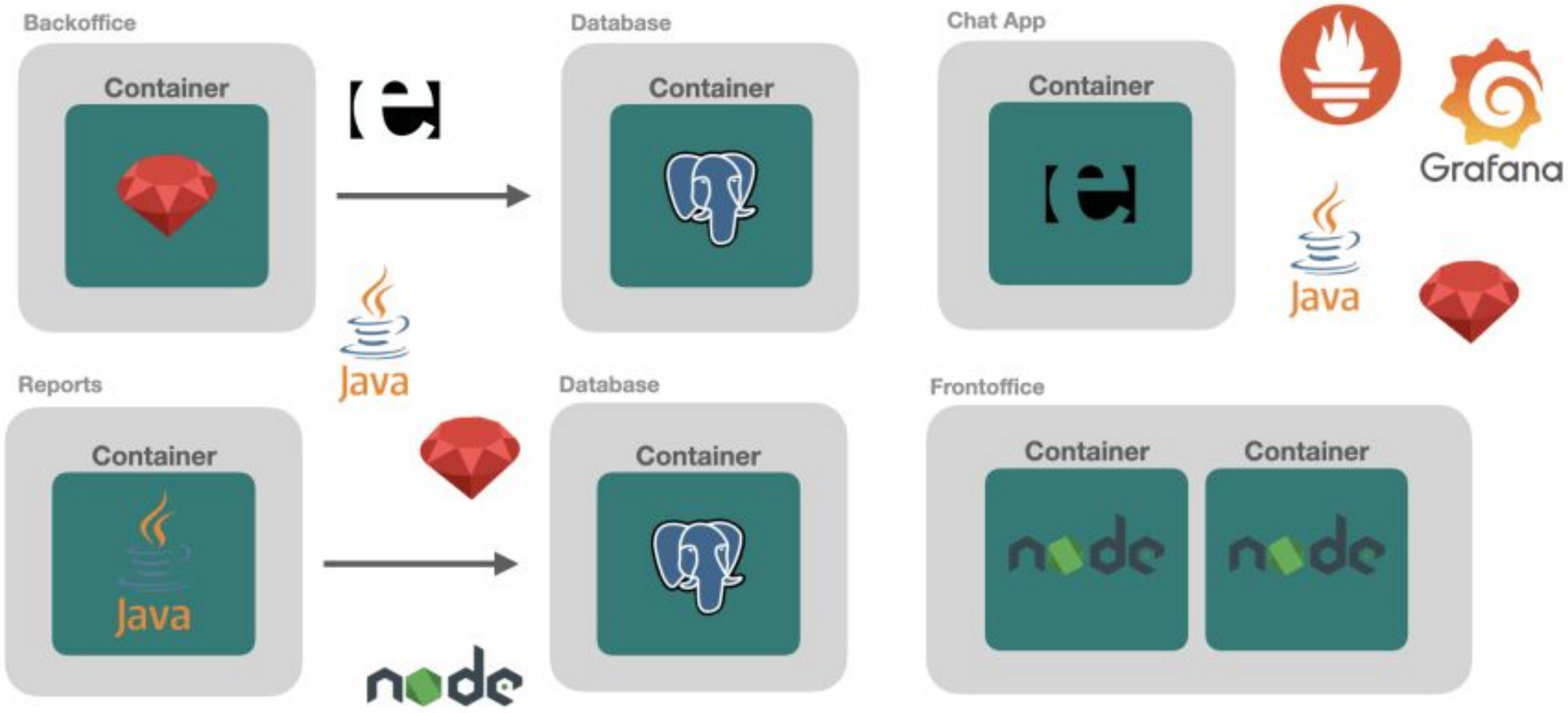


A furthermore functional requirement

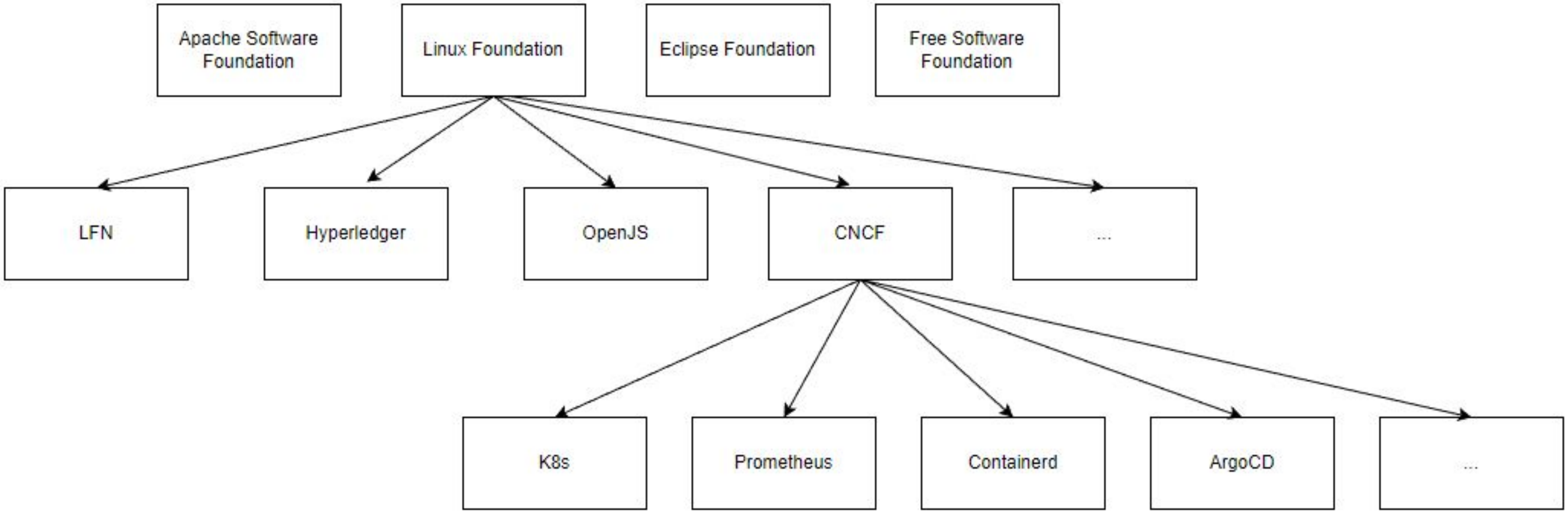
Chat app cannot be down for much time, and in case it goes down, we should **make sure it is started again**, having the capability of **self-healing**



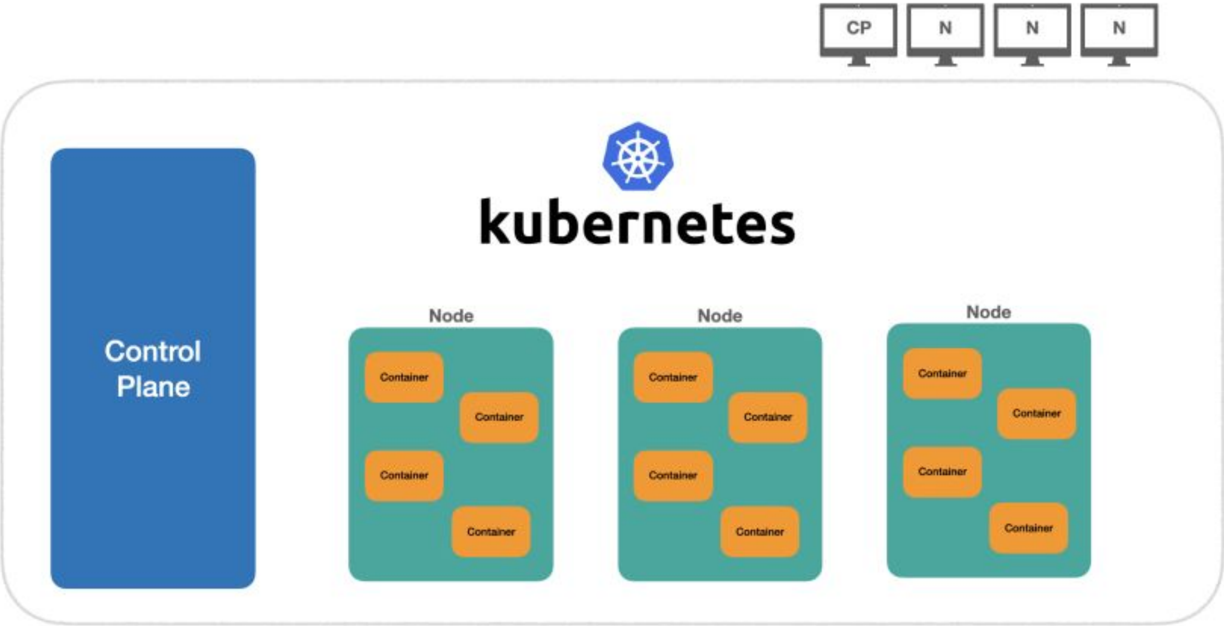
Container Management



A bit history - Linux Foundation Architecture



K8s Architecture



- 1 machine called **Control Plane**, in which the cluster is created and is responsible to accept new machines (or nodes) on the cluster
- 3 other machines called **Nodes**, which will contain all the managed containers by the cluster.

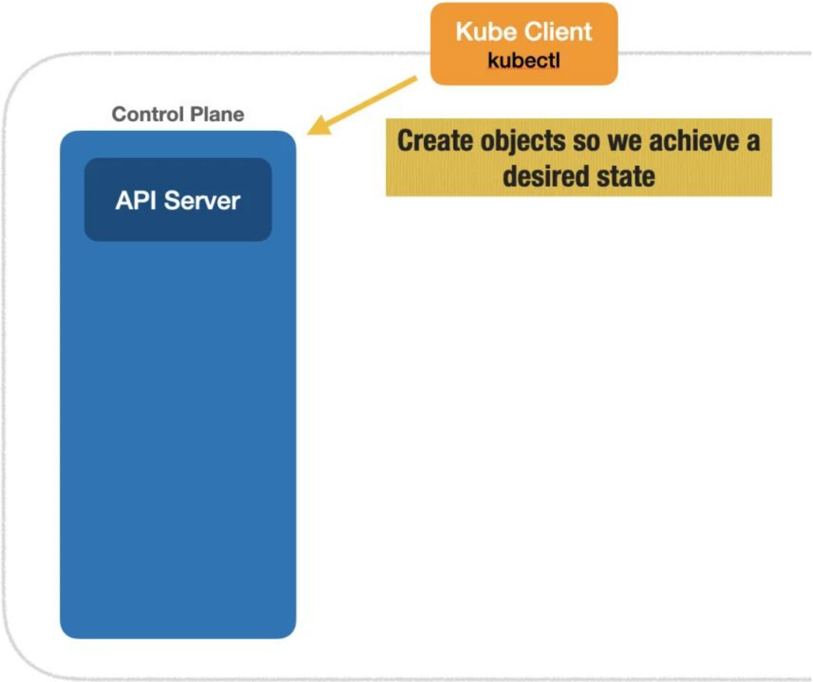
A rule of thumb

All the running containers establish what we call the **cluster state.**

In Kubernetes, we declare the desired state of the cluster by making HTTP requests to the Kubernetes API, and Kubernetes will "work hard" to achieve the **desired state**.

However, making plain HTTP requests in order to declare the state can be somewhat cumbersome, error prone and a tedious job. How about having some CLI in the command-line which would do the hard work of authenticating and making HTTP requests? **Kubectl**

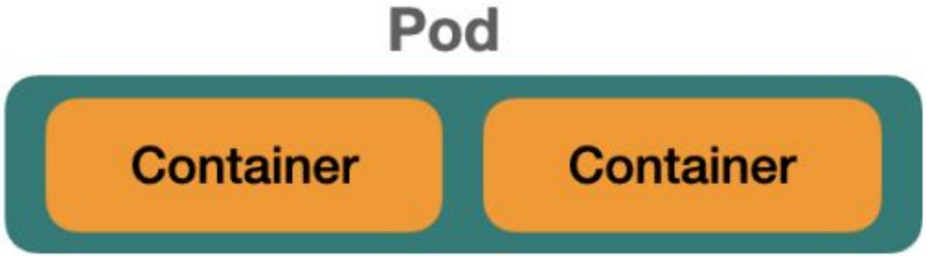
Creating Objects in K8s Cluster



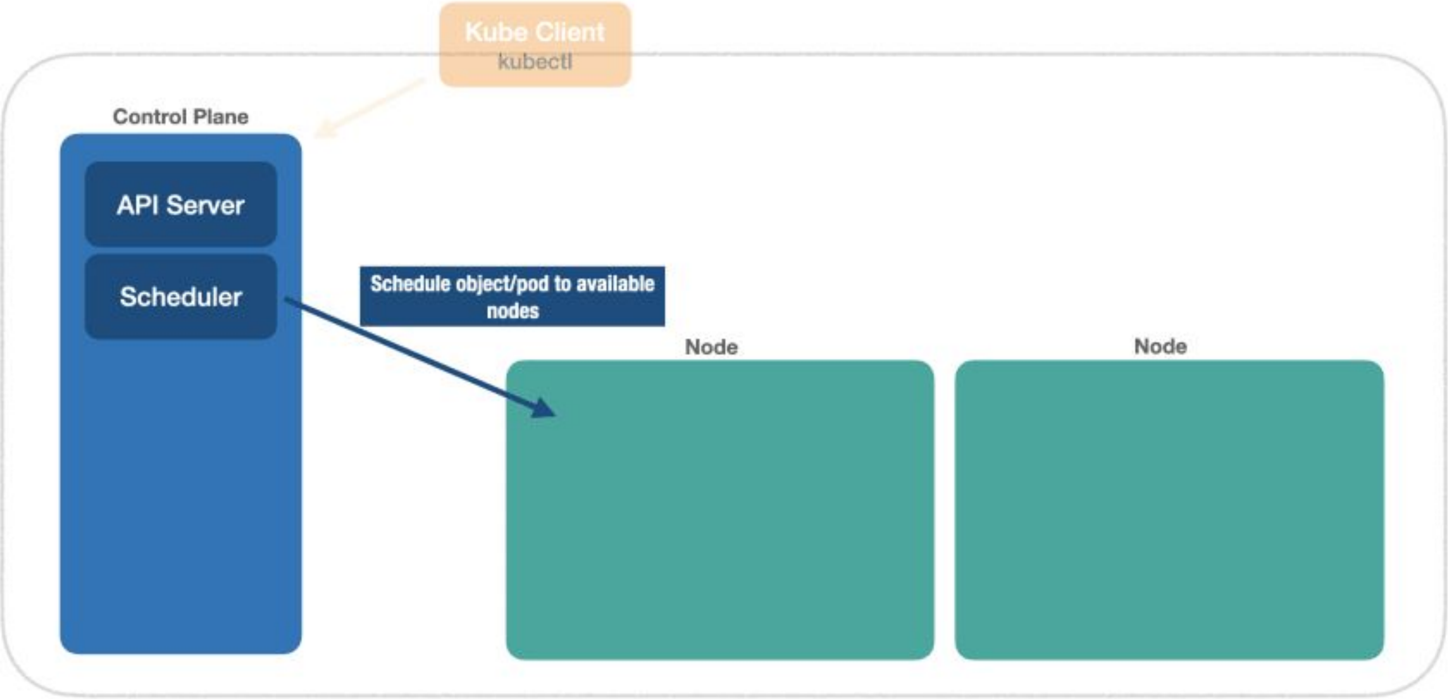
Demo: create an nginx object

What is pod

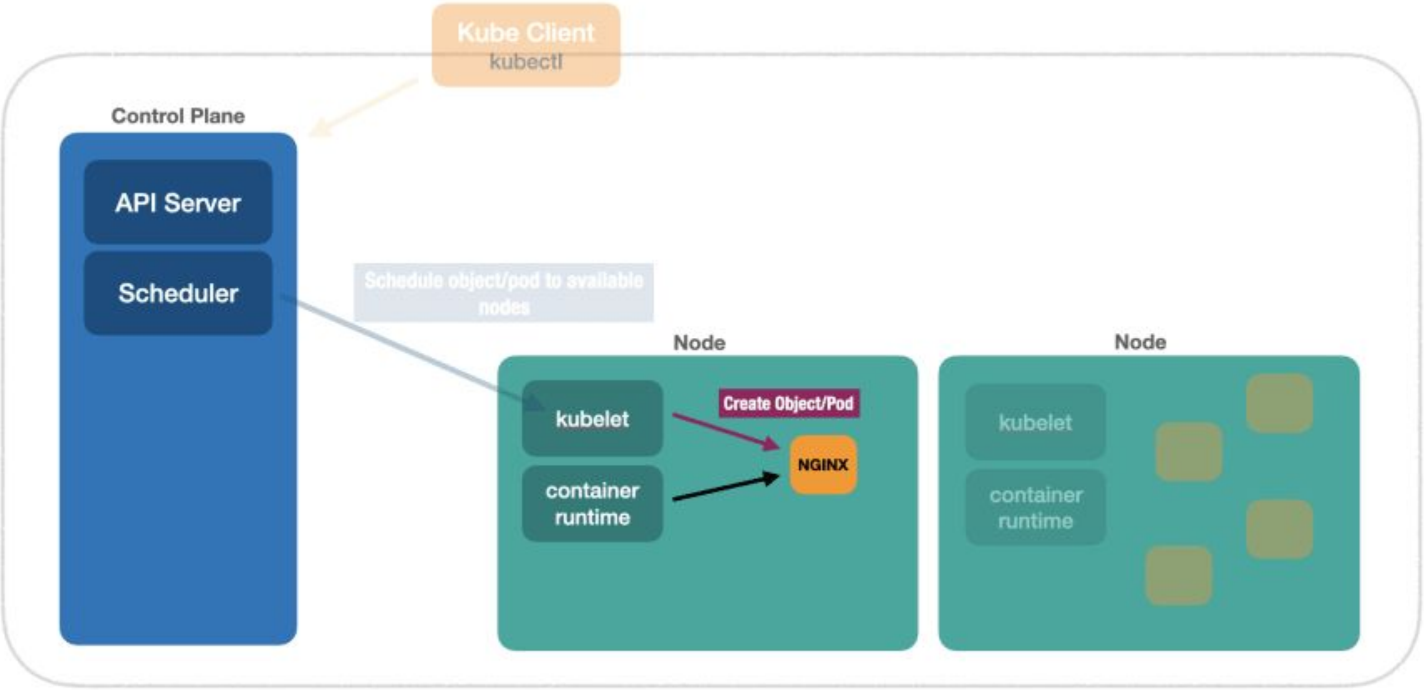
```
$ kubectl run nginx --image=nginx
pod/nginx created
```



Architecture Flow



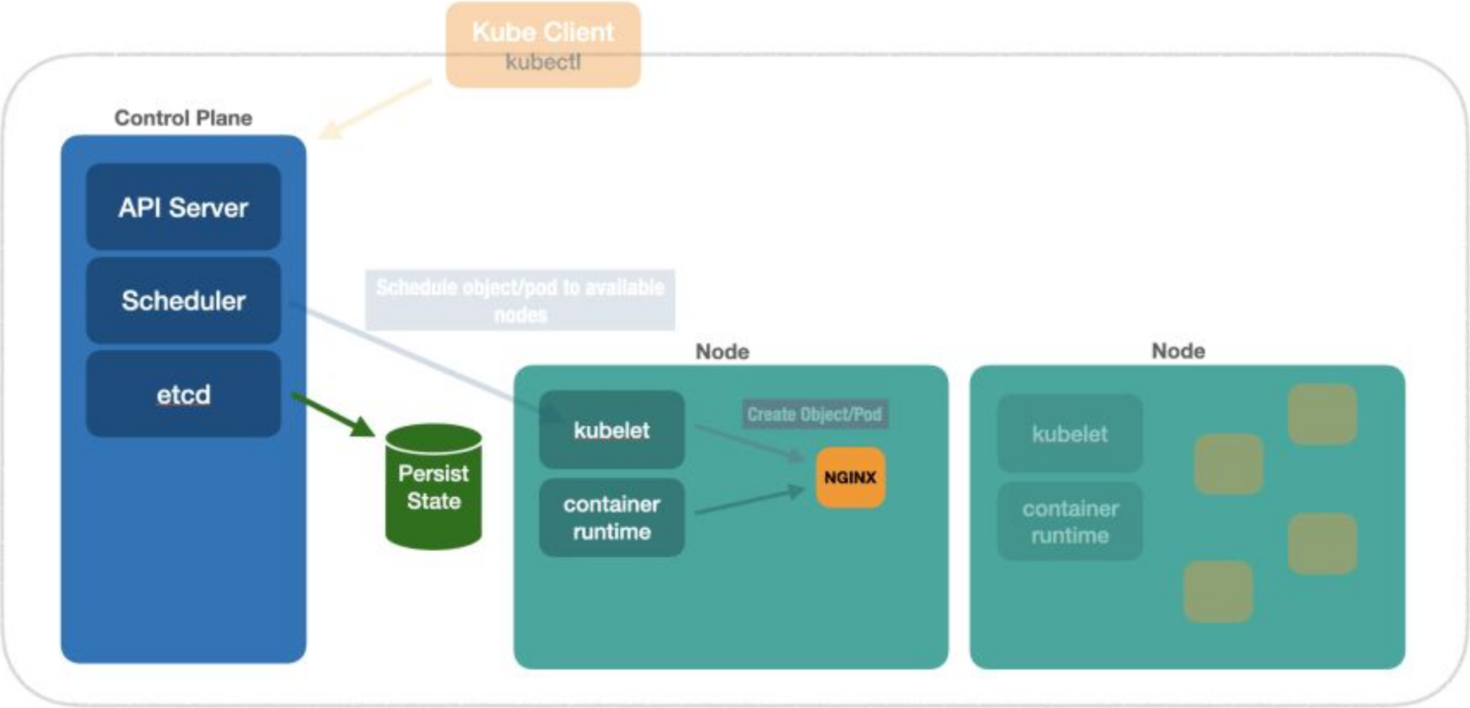
Node kubelet



Each node contains a component called **Kubelet**, which admits objects coming from the Scheduler.

And, using the container runtime installed in the node (could be Docker, containerd, CRI-O, etc), creates the object in the node

etcd - A distributed k-v store



A bit K8s networking

Create 2 deployments:

```
kubectl run server --image=nginx
kubectl run client --image=nginx
```

In containerized applications, by default, containers are isolated and do not share the host network. *Neither do Pods*.

Execute commands in a running pod (server):

```
kubectl exec server -- curl localhost
```

Execute commands in another running pod (client):

```
kubectl describe pod server | grep IP
kubectl exec client -- curl 172.17.0.6
```

Service

in case we perform a deploy, i.e change the old `server` Pod to a *newer Pod*, **there's no guarantee that the new Pod will get the same previous IP**.

We need some mechanism of **pod discovery**, where we can declare a *special object* in Kubernetes that will **give a name** to a given pod. Therefore, within the cluster, we could **reach Pods by their names** instead of internal IP's.

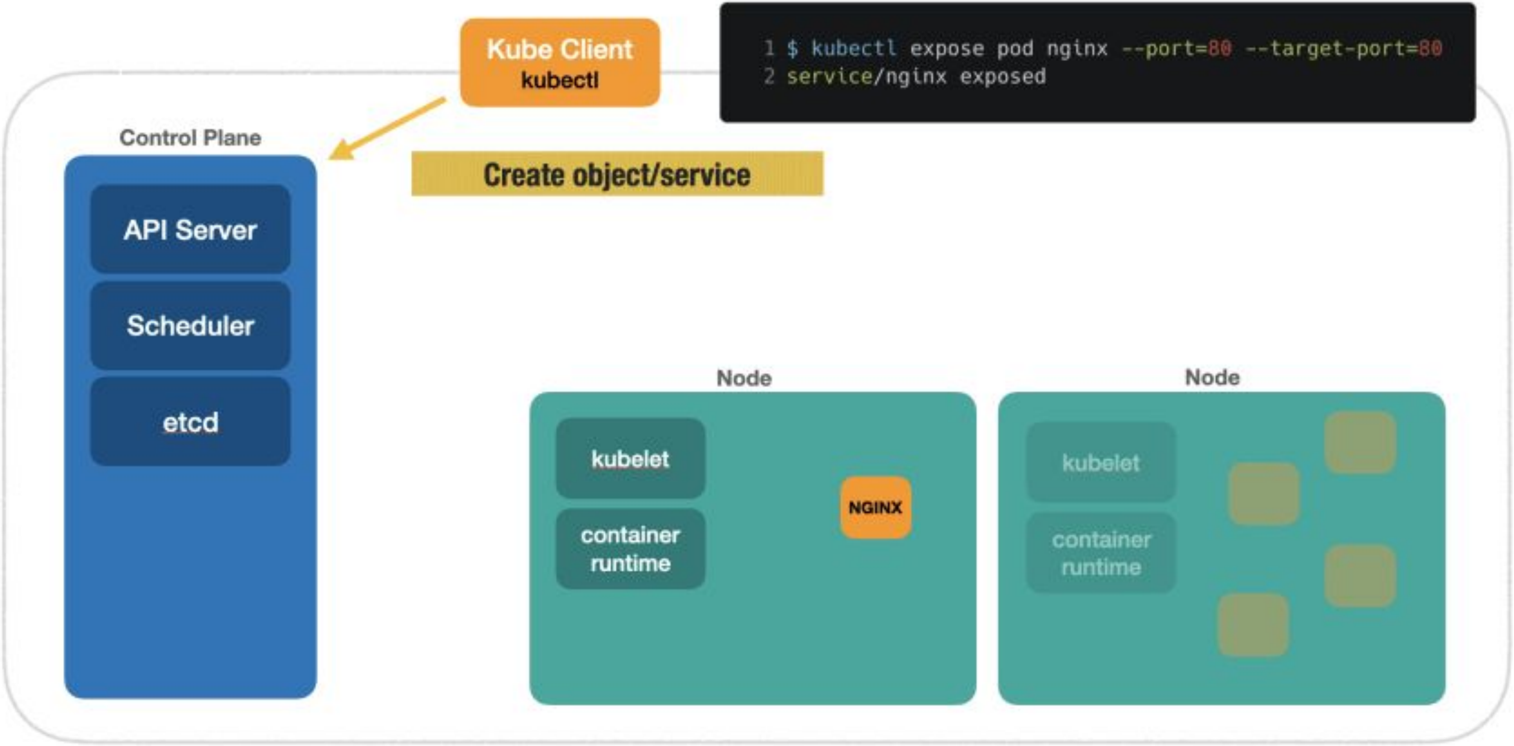
Controller Manager

responsible to receive a request for special objects like **Services** and expose them via service discovery.

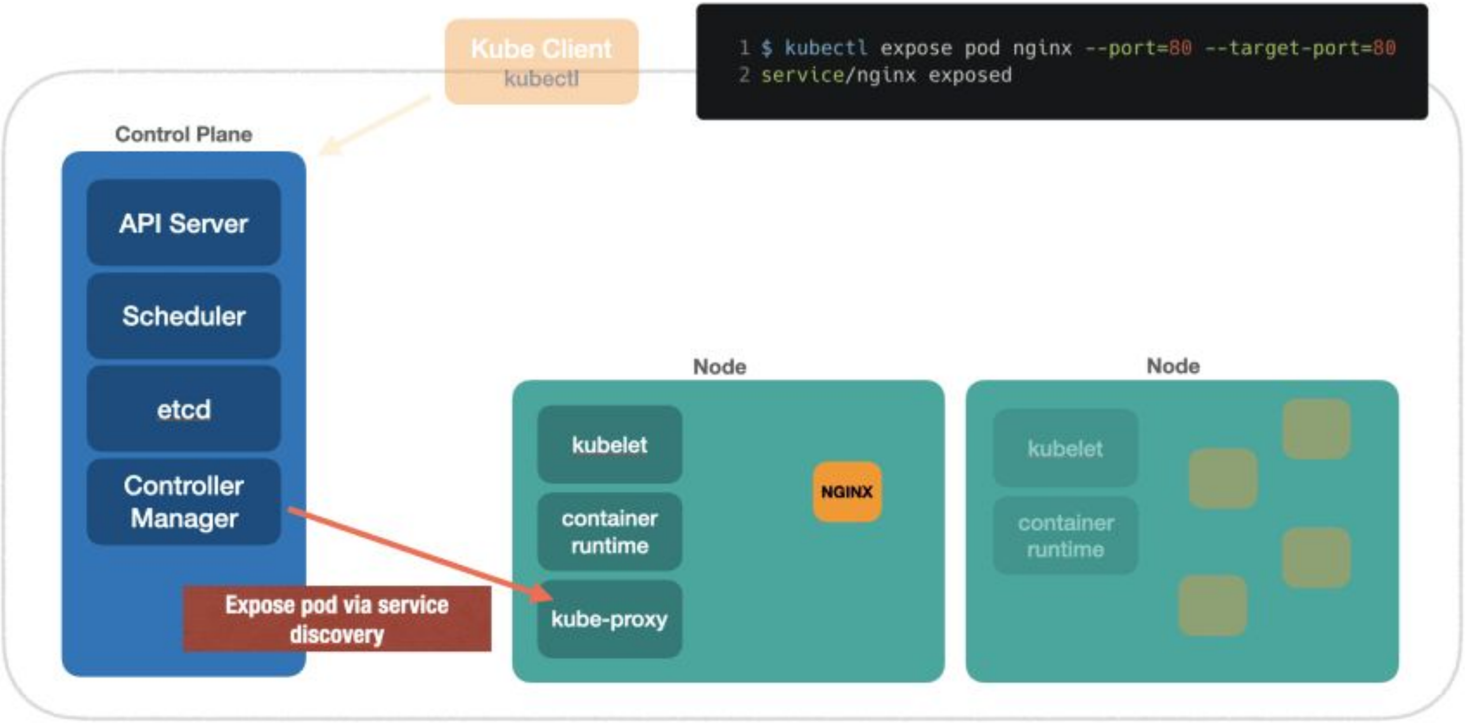
Expose server name (not IP):

```
kubectl expose pod server --port=80 --target-port=80
kubectl exec client -- curl server
```

Architecture Flow - Issued the creation of the Service Object

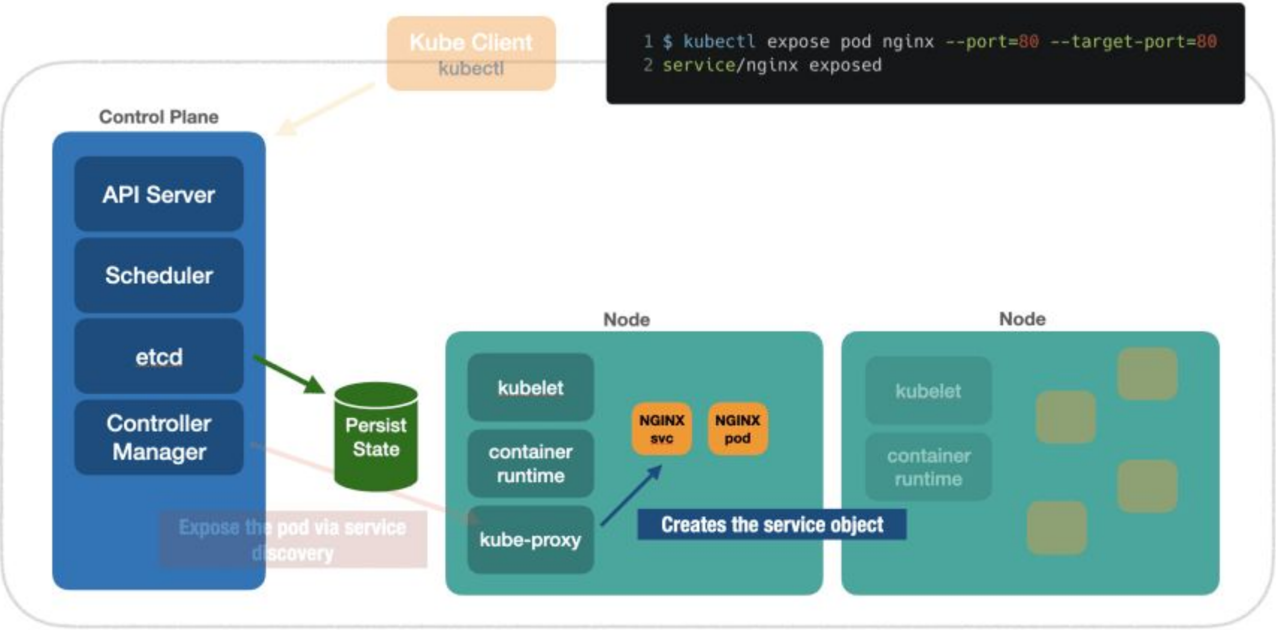


Architecture Flow - Expose the Pod via service discovery



Afterwards

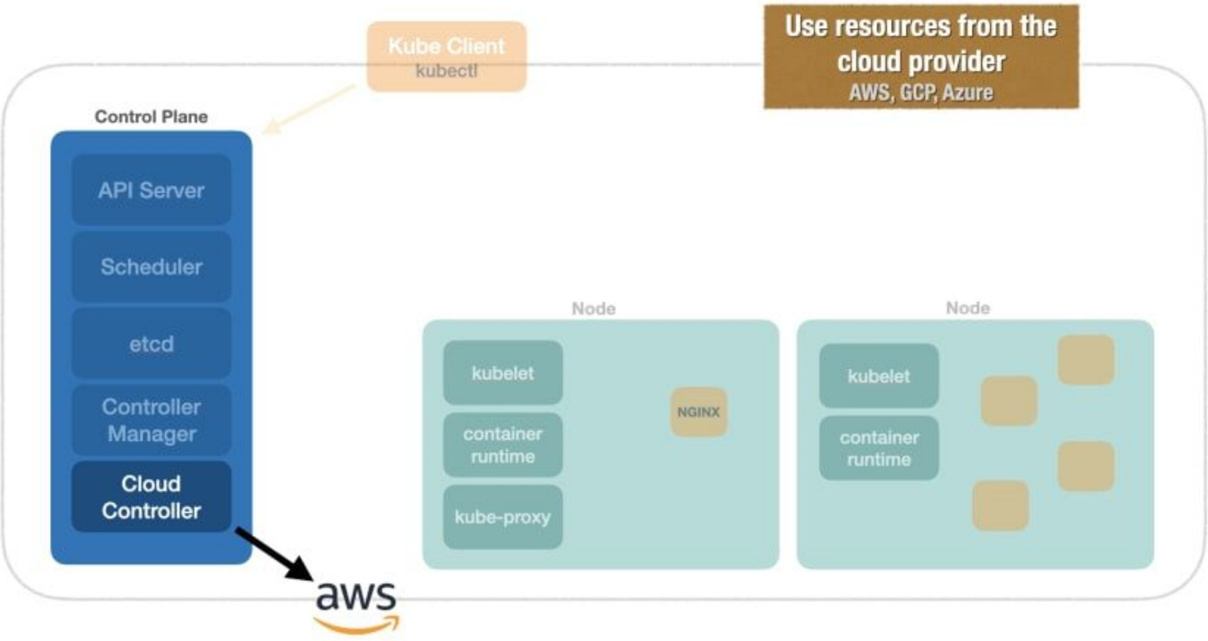
the controller manager *routes* to the **`kube-proxy`** component that is running in the node, which will **create the Service object** for the respective Pod. At the end of the process, the state is persisted in `etcd`.



Cloud Controller

Responsible for receiving requests to create objects and interacting with the underlying cloud provider if needed.

For example, when we create a Service object of type `LoadBalancer`, the Cloud Controller *will create a LB* in the underlying provider, be it AWS, GCP, Azure etc



Main Architecture Flow