# Kubernetes Pods, Replica sets and Deployment

# PODS

Kubernetes doesn’t run containers directly; instead, it wraps one or more containers into a higher-level structure called a pod. Pods are EPHIMERAL.

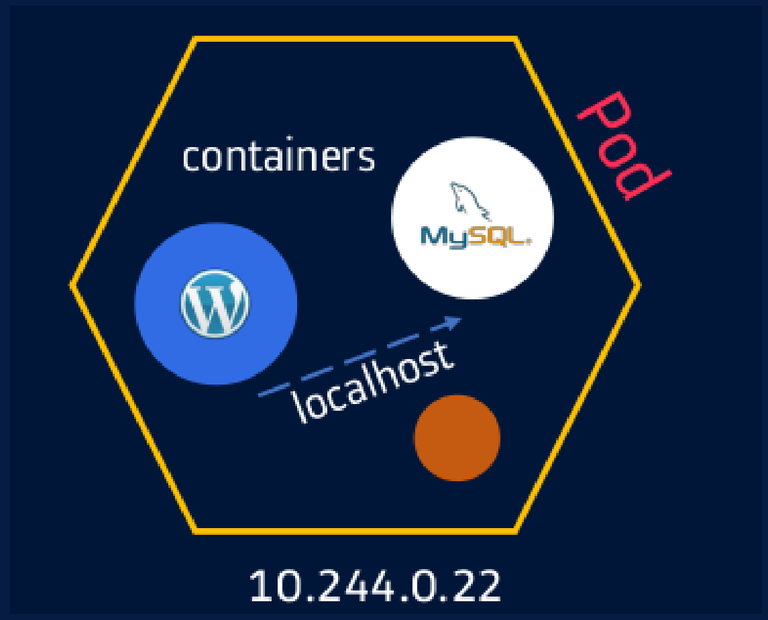
It is also the smallest deployable unit that can be created, scheduled, and managed on a Kubernetes cluster. Each pod is assigned a unique IP address within the cluster.

Pods can hold multiple containers as well, but you should limit yourself when possible. Because pods are scaled up and down as a unit, all containers in a pod must scale together, regardless of their ndividual needs. This leads to wasted resources

# PODS

**Any containers in the same pod will share the same storage volumes and network resources and communicate using localhost.**

**K8s uses YAML to describe the desired state of the containers in a pod. This is also called a Pod Spec. These objects are passed to the kubelet through the API server.**



# Imperative vs Declarative commands

Kubernetes API defines a lot of objects/resources, such as namespaces, pods, deployments,services, secrets, config maps etc.

IMPERATIVELY

- Involves using any of the verb-based commands like kubectl run, kubectl create, kubectl expose, kubectl delete, kubectl scale, and kubectl edit.
- Suitable for testing and interactive experimentation

DECLARATIVELY

- Objects are written in YAML files and deployed using kubectl create or kubectl apply
- Best suited for production environments

# Creating Pods

## COMMAND FORMAT

**kubectl run <pod-name> --image <image-name>**

## EXAMPLE COMMAND

**kubectl run nginx --image nginx --dry-run=client**

***“Dry-run doesn’t run the command but will show what the changes the command would do to the cluster”***

# Creating Pods: Imperative way

kubectl run test --image nginx --port 80

kubectl get pods –o wide

kubectl describe pod test

kubectl exec -it test -- /bin/bash

curl <IP of POD>

# Manifest / Spec file

K8s object configuration files - Written in YAML or JSON

EXAMPLE

apiVersion - version of the Kubernetes API
used to create the object
kind - kind of object being created
metadata - Data that helps uniquely identify
the object, including a name and
optional namespace
spec - a configuration that defines the desired for
the object

# Manifest files Man Pages

List all K8s API supported Objects and Versions

kubectl api-resources
kubectl api-versions

Man pages for objects

kubectl explain <object>.<option>
kubectl explain pod
kubectl explain pod.apiVersion
kubectl explain pod.spec

# Pod.yml

kubectl run nginx --image nginx --dry-run=client –o yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: Nginx
  labels:
    app: my-app
spec:
  containers:
  - name: Nginx
    image: nginx
    ports:
    - containerPort: 80
```

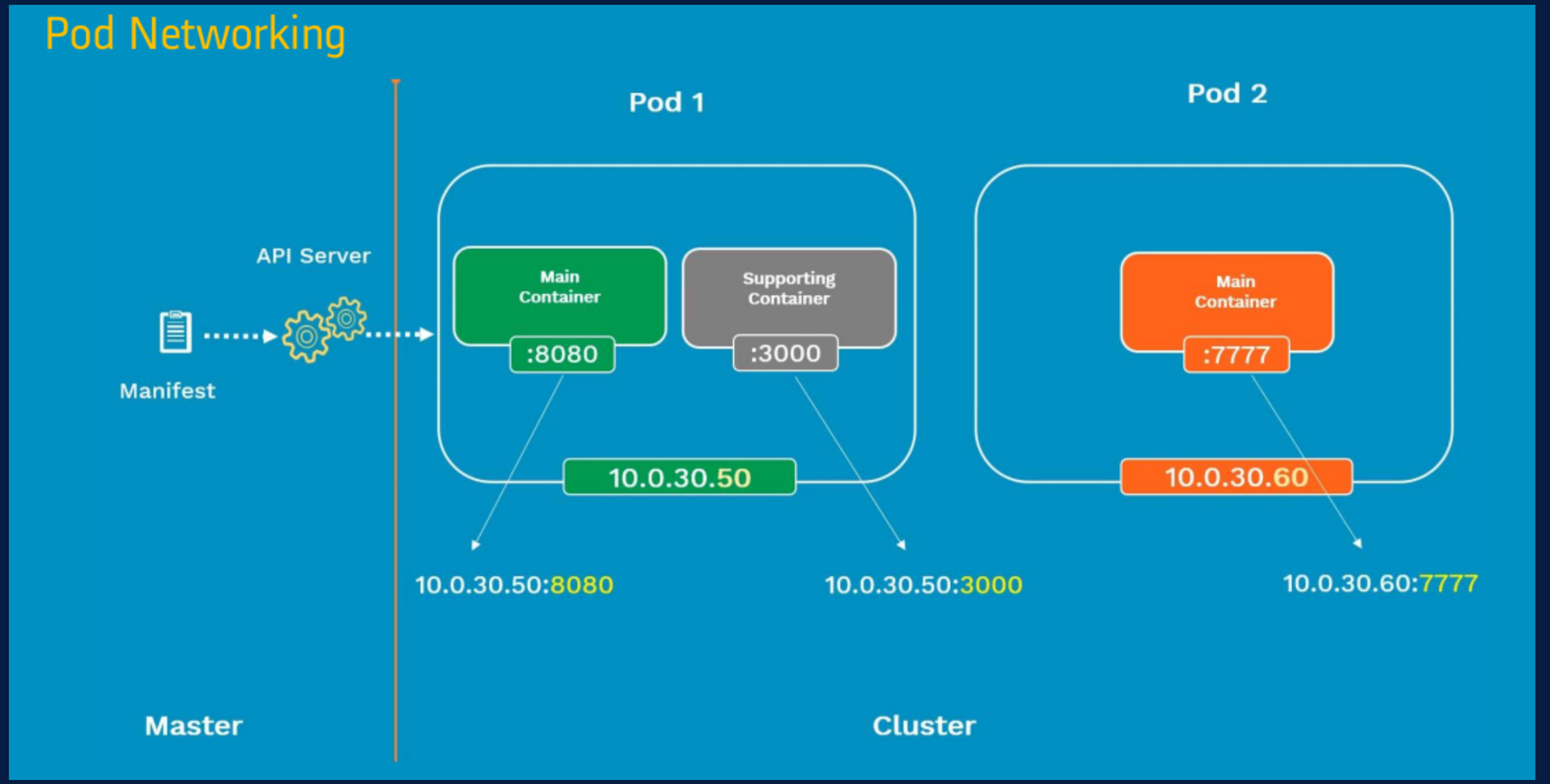
# Creating Pods: Declarative way

kubectl create –f pod-definition.yml

kubectl apply –f pod-definition.yml – if manifest file is changed/updated after deployment and need to re-deploy the pod again

kubectl delete pod <pod-name>d a subheading

# Pod Networking



# Replication Controller

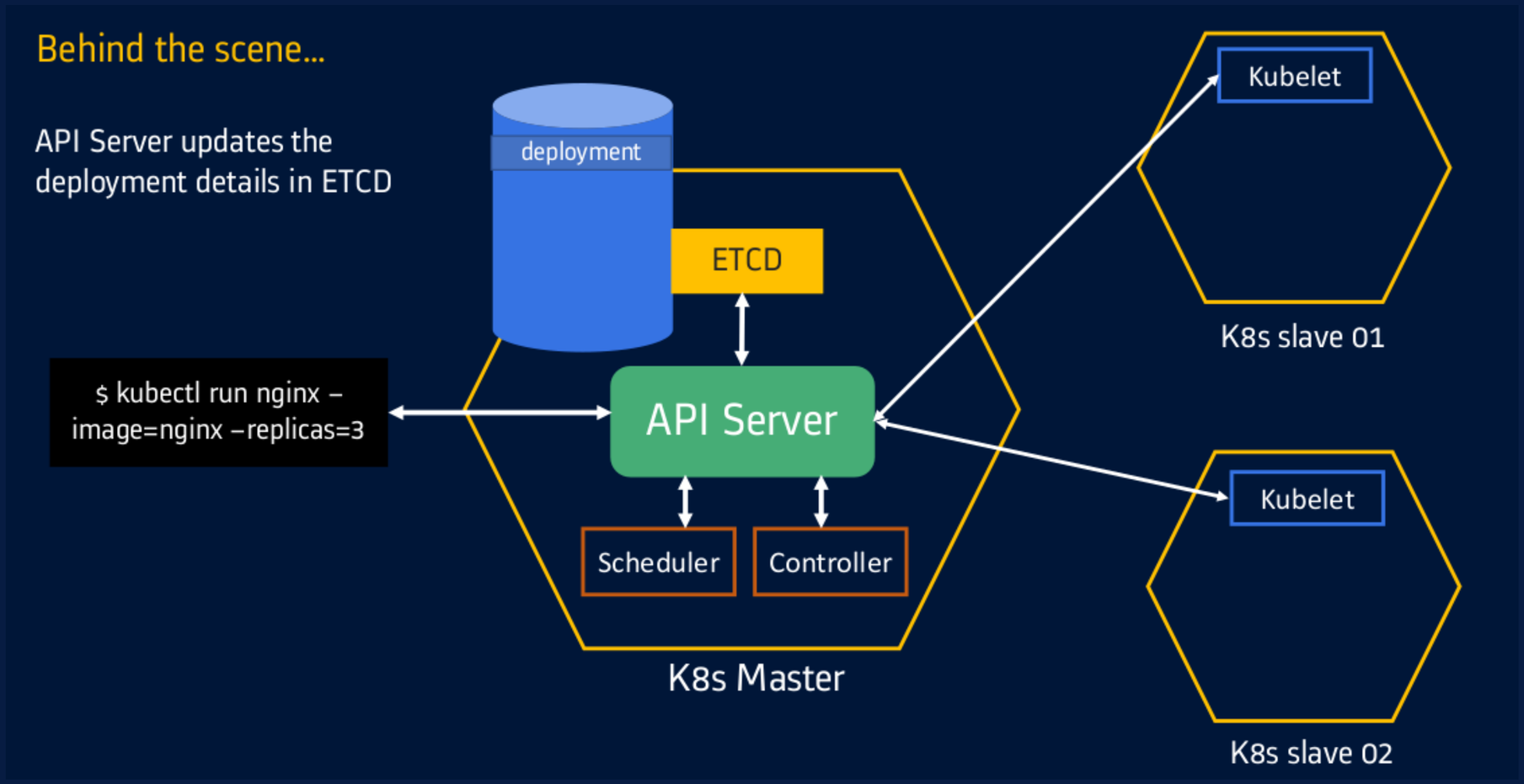
A single pod may not be sufficient to handle the user traffic. Also if this only pod goes down because of a failure, K8s will not bring this pod up again automatically.

Kubernetes supports different controllers(Replicacontroller & ReplicaSet) to handle multiple instances of a pod. Ex: 3 replicas of the nginx webserver

Replication Controller ensures high availability by replacing the unhealthy/dead pods with a new one to ensure required replicas are always running inside a cluster

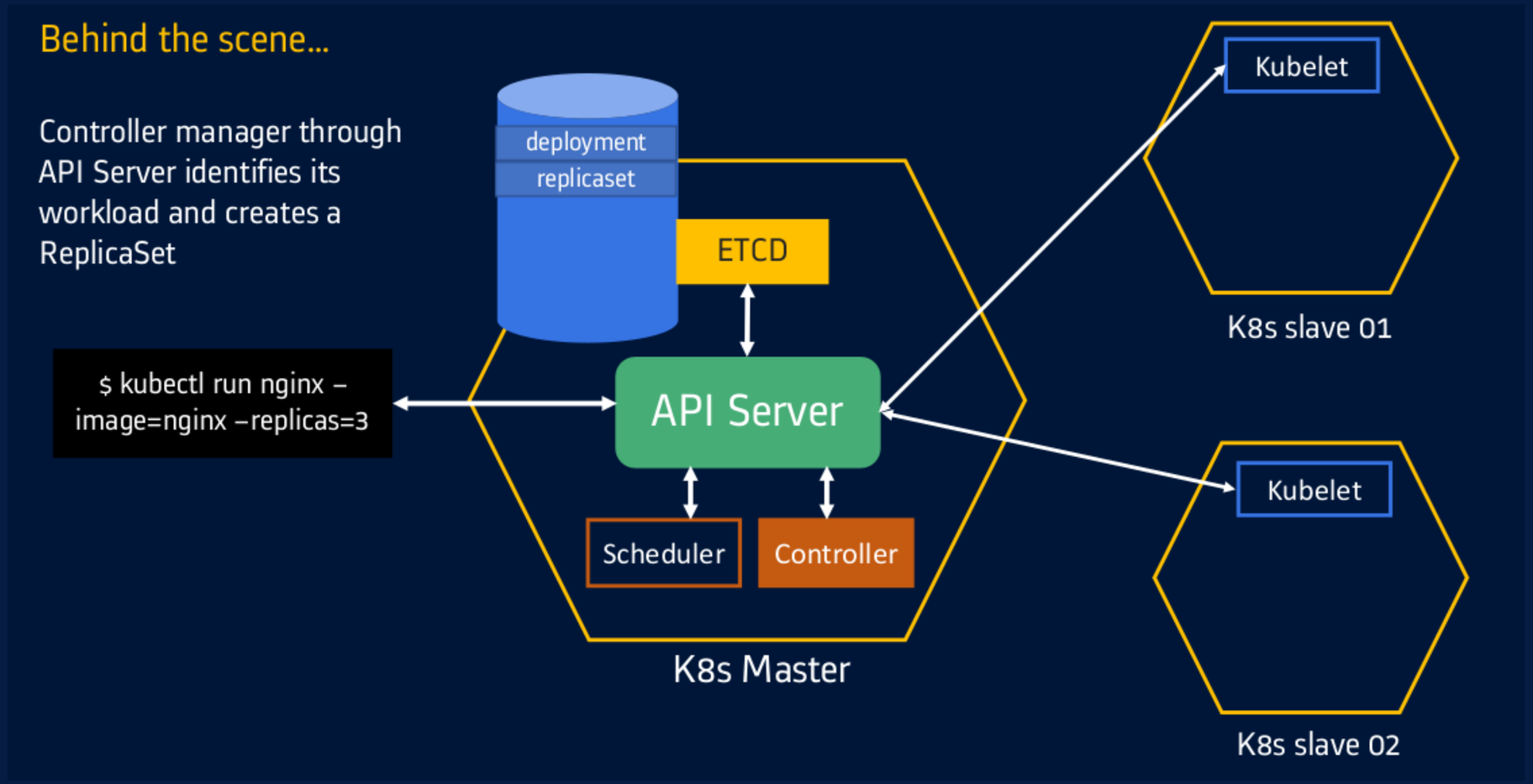
## Behind the scene...

API Server updates the deployment details in ETCD



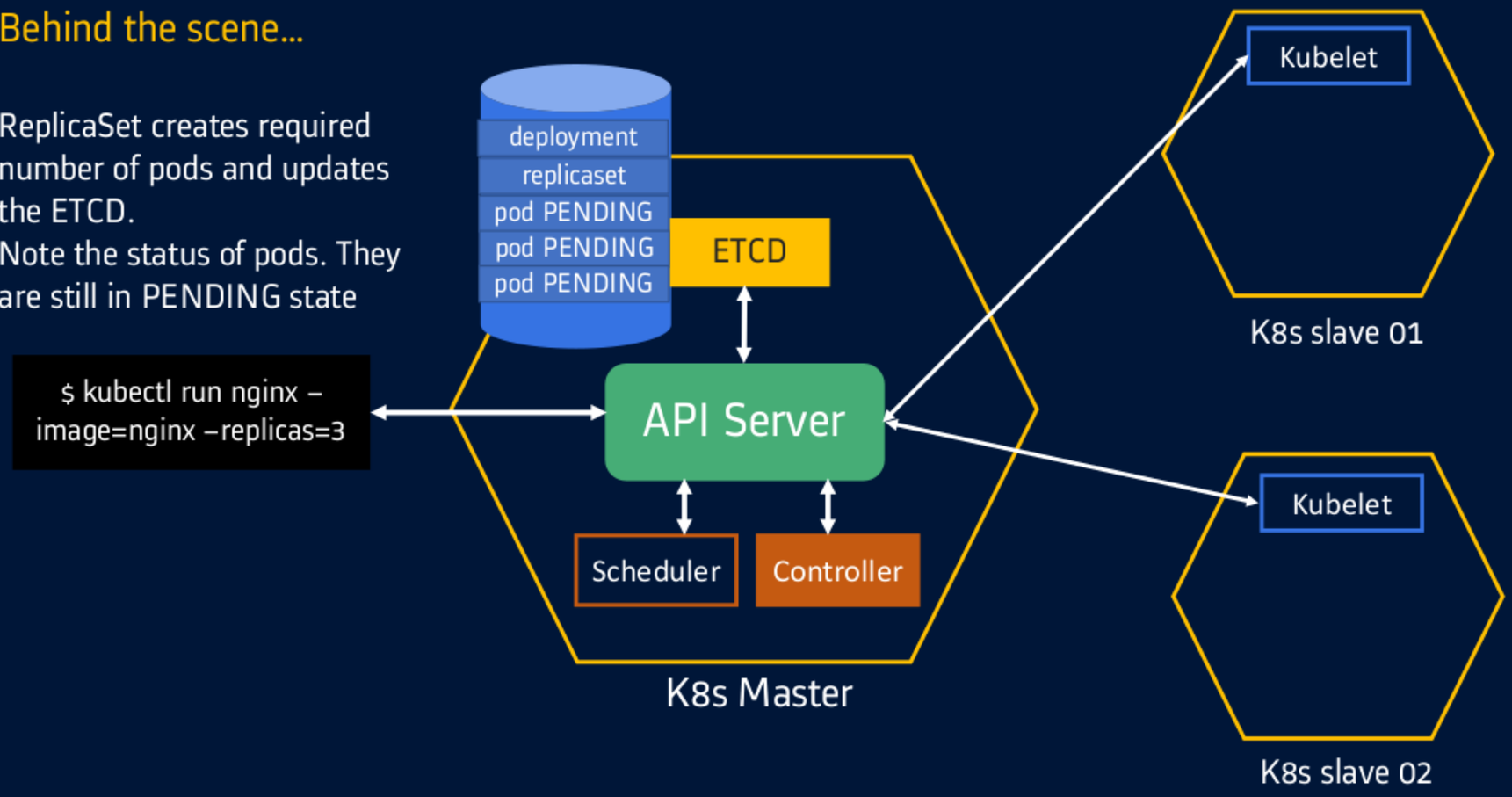
## Behind the scene...

Controller manager through API Server identifies its workload and creates a ReplicaSet



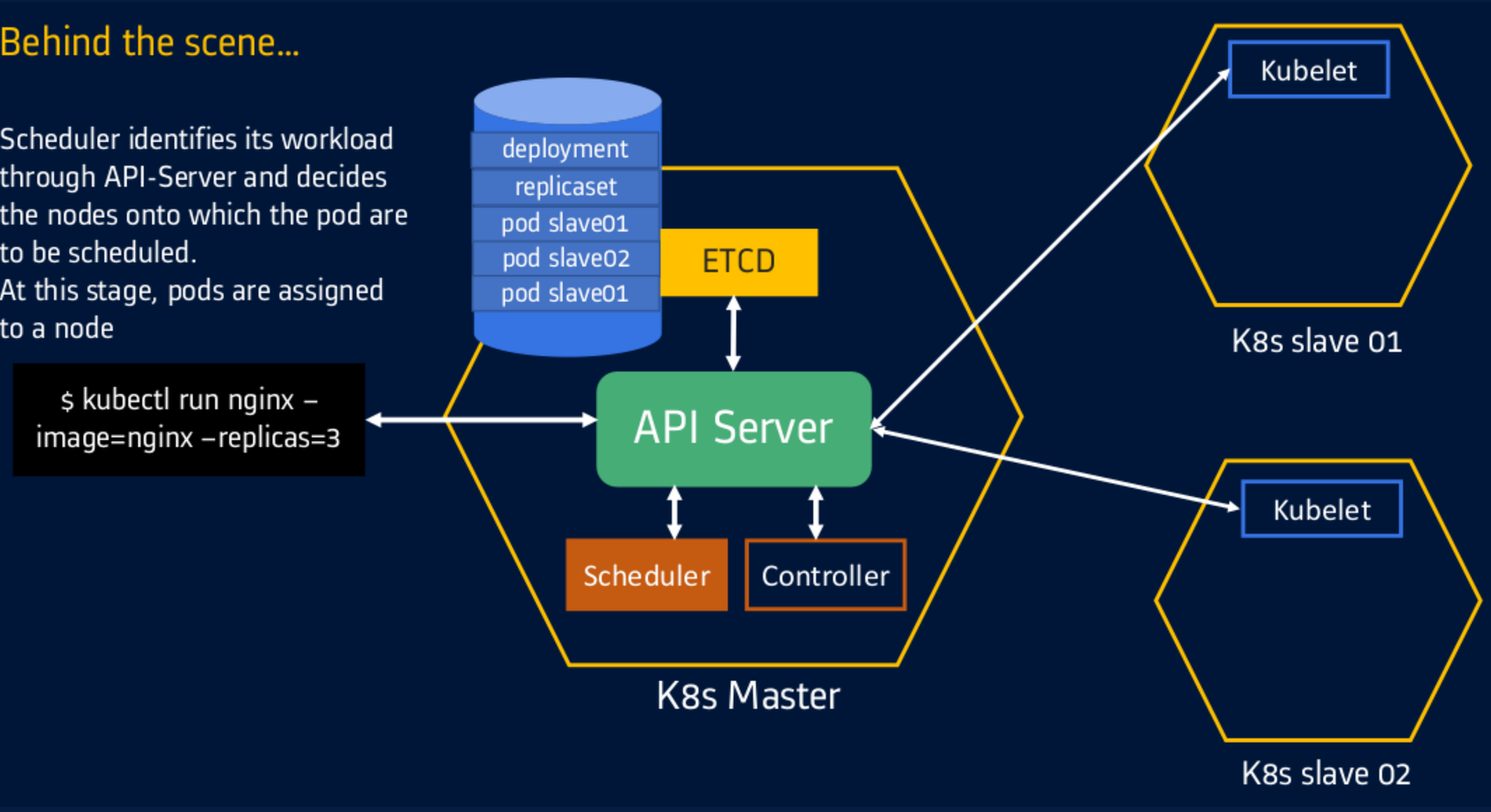
## Behind the scene...

ReplicaSet creates required number of pods and updates the ETCD.
Note the status of pods. They are still in PENDING state



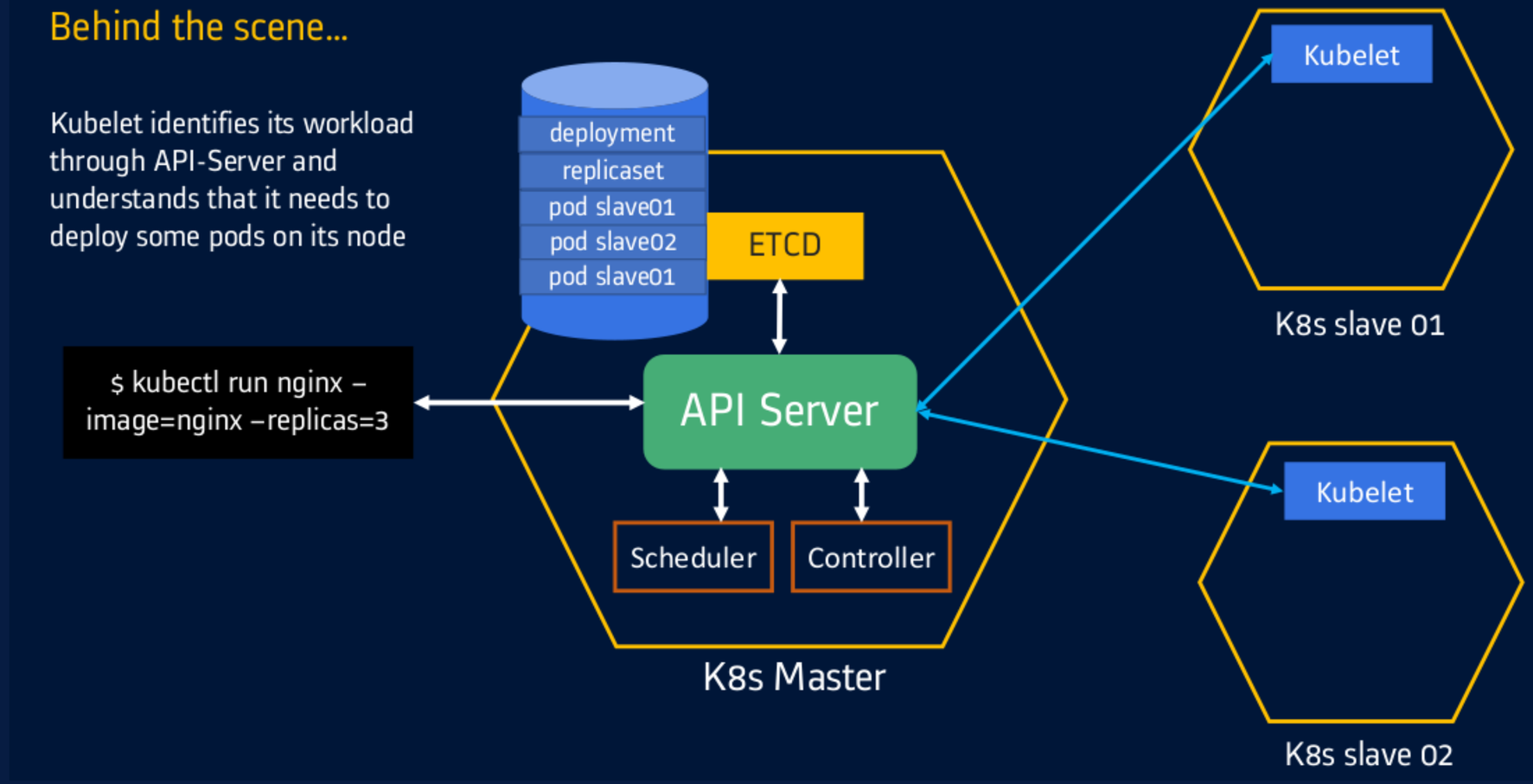
## Behind the scene...

Scheduler identifies its workload through API-Server and decides the nodes onto which the pod are to be scheduled.
At this stage, pods are assigned to a node



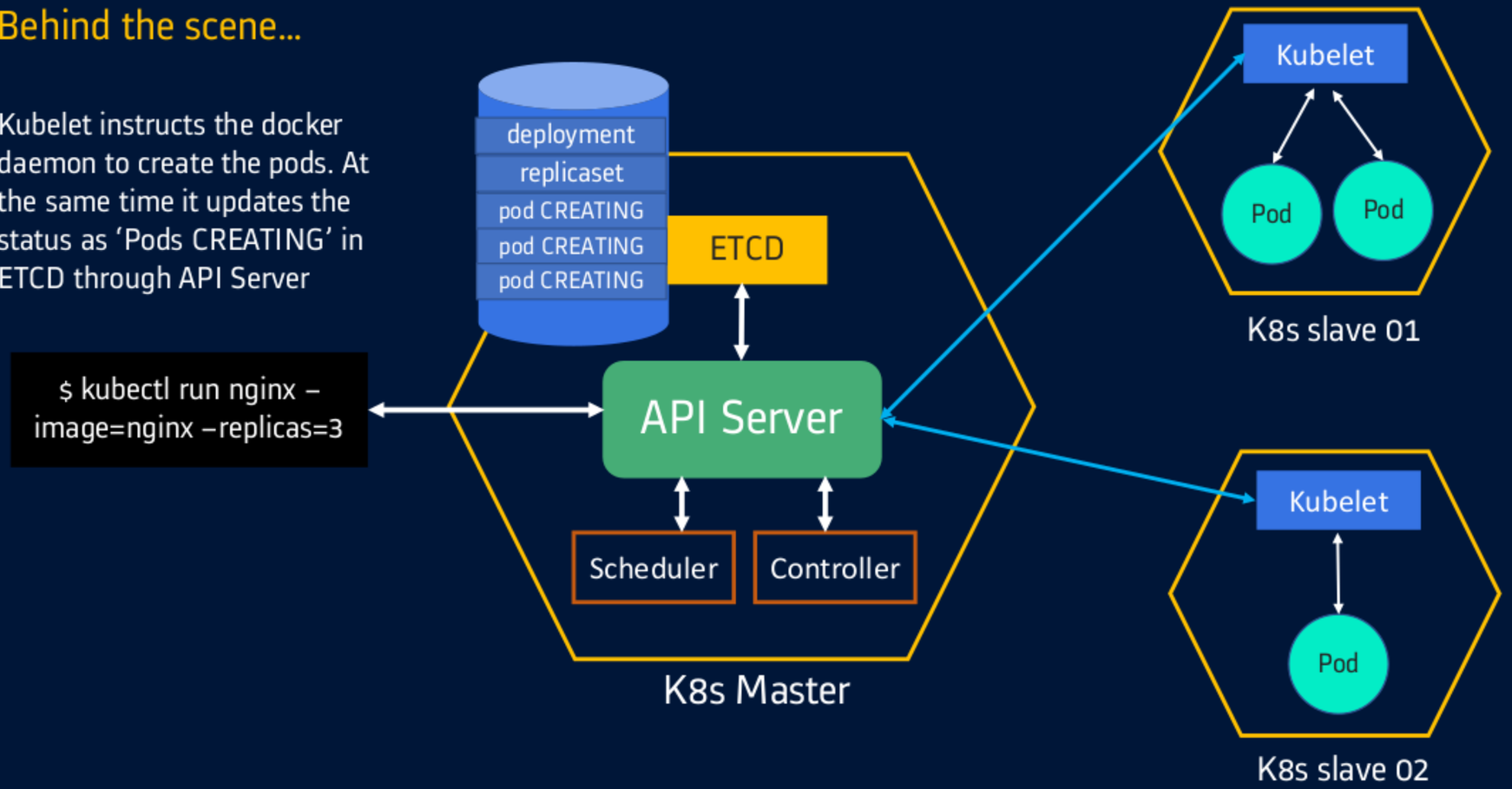
## Behind the scene...

Kubelet identifies its workload through API-Server and understands that it needs to deploy some pods on its node



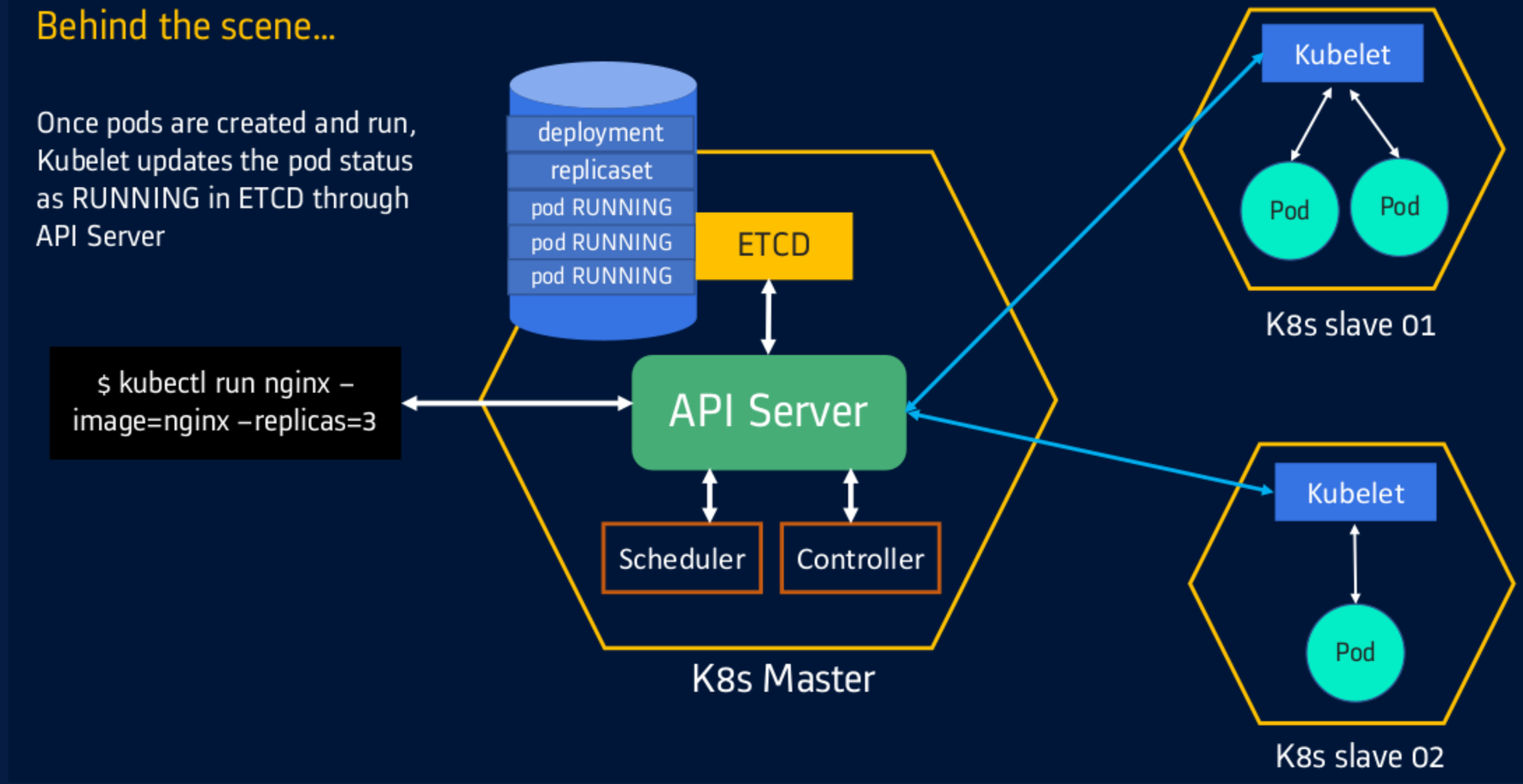
## Behind the scene...

Kubelet instructs the docker daemon to create the pods. At the same time it updates the status as 'Pods CREATING' in ETCD through API Server



## Behind the scene...

Once pods are created and run, Kubelet updates the pod status as RUNNING in ETCD through API Server



# Labels

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

Labels allow us to logically group certain objects by giving various names to them. You can label pods, services, deployments and even nodes.

kubectl get pods -l environment=production

kubectl get pods -l environment=production,
tier=frontend

# Labels

If labels are not mentioned while deploying k8s objects using imperative commands, the label is auto-set as app: <object-name>

kubectl get pods --show-labels

kubectl label pod nginx environment=dev

```
metadata:
  name: label-demo
  labels:
    environment: production
    app: nginx
```

# Selectors

Selectors allow filtering the objects based on labels
A label selector can be made of multiple requirements which are comma-separated

Equality-based Selector

- Equality- or inequality-based requirements allow filtering by label keys and values.
- Three kinds of operators are admitted =,==,!=

Used by Replication Controllers and Services

# Selectors

## Set-based Selector

- Set-based label requirements allow filtering keys according to a set of values.
- Three kinds of operators are supported: in,notin and exists (only the key identifier).

kubectl get pods -l 'environment in (production, qa)'

```
environment in (production, qa)
tier notin (frontend, backend)
partition
!partition
```

# ReplicaSet

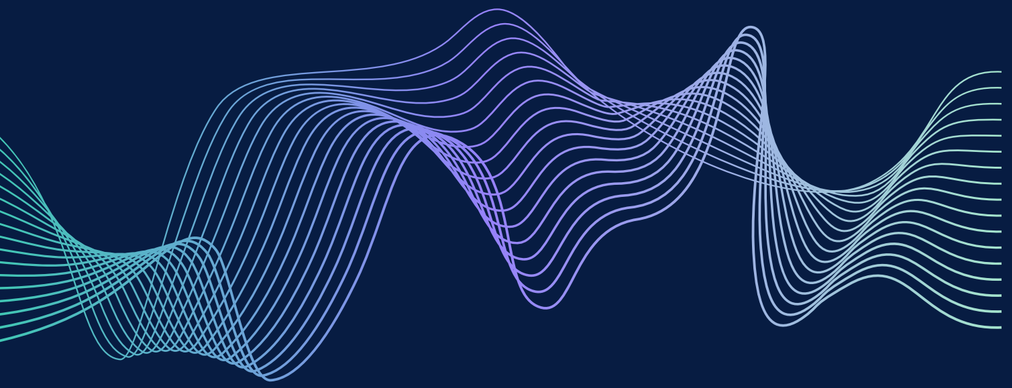
ReplicaSets are a higher-level API that gives the ability to easily run multiple instances of a given pod.

Rensures that the exact number of pods(replicas) are always running in the cluster by replacing any failed pods with new ones.

The replica count is controlled by the replicas field in the resource definition file.

Replication uses set-based selectors whereas replicacontroller uses equality based selectors

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replicaset
spec:
  replicas: 3  # Number of NGINX replicas you want to maintain
  selector:
    matchLabels:
      app: nginx
```



```
template:
  metadata:
 name: nginx
 labels:
 app: nginx
 spec:
 containers:
 - name: nginx
 image: nginx
 ports:
 - containerPort: 80
```

# Creation of Replicaset

kubectl create –f replica-set.yml

kubectl get rs –o wide

kubectl delete replicaset <replicaset-name> - delete a replicaset;

kubectl delete -f replica-set.yml

kubectl get all - get all in one shot

kubectl scale -–replicas=6 –f replicaset-definition.yml – scale using definition file

kubectl scale -–replicas=6 replicaset <replicaset-name> - using name of replicaset

# Deployment

A Deployment provides declarative updates for Pods and ReplicaSets.

You describe a desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate.

It seems similar to ReplicaSets but with advanced functions
Deployment is the recommended way to deploy a pod or RS

# Features

By default Kubernetes performs deployments in rolling update strategy.

Below are some of the key features of deployment:

✓ Easily deploy a RS

✓ Rolling updates pods

✓ Rollback to previous deployment versions

✓ Scale deployment

✓ Pause and resume deployment

# Features

- In Kubernetes there are a few different ways to release updates to an application

- Recreate: terminate the old version and release the new one. Application experiences downtime.

- RollingUpdate: release a new version on a rolling update fashion, one after the other. It’s thedefault strategy in K8s. No application downtime is required.

- Blue/green: release a new version alongside the old version then switch traffic

# Deployment Strategy

Rollout is the process of gradually deploying or upgrading your application containers.

• For every rollout/upgrade, a version history will be created, which helps in rolling back to working version in case of an update failure

```
spec:
  replicas: 10
strategy:
    type: Recreate
```

```
spec:
  replicas: 10
strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 2
      maxUnavailable: 0
```

# Rolling Update Strategy

By default, deployment ensures that only 25% of your pods are unavailable during an update and does not
update more that 25% of the pods at a given time
It does not kill old pods until/unless enough new pods come up
It does not create new pods until a sufficient number of old pods are killed

kubectl create deployment nginx --image nginx --dry-run -o yaml

kubectl create -f deployment.yml --record (--record is optional, it just records the events in the deployment)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3  # Number of NGINX replicas you want to run
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx-container
        image: nginx:latest  # NGINX Docker image
        ports:
        - containerPort: 80  # Port on which NGINX listens
```

# Only Kind Changes

# Deployment

kubectl get deployments

kubectl describe deployment <deployment-name>

kubectl edit deployment <deployment -name>

kubectl scale deployment <deployment -name> --replicas2

kubectl rollout status deployment <deployment -name>

kubectl rollout history deployment <deployment -name>

kubectl set image deplyment  deployment -name> image=newimage --record

kubectl rollout undo <deployment-name>