Introduction to Kubernetes

Instructor:

John J Rofrano

Senior Technical Staff Member, DevOps Champion
IBM T.J. Watson Research Center
rofrano@us.ibm.com (@JohnRofrano)

What will you lean?

- What is Kubernetes
- Why would you use a Kubernetes for deployment
- Kubernetes Architecture Overview
- How to deploy your own containers in Kubernetes

Create your Kubernetes Cluster

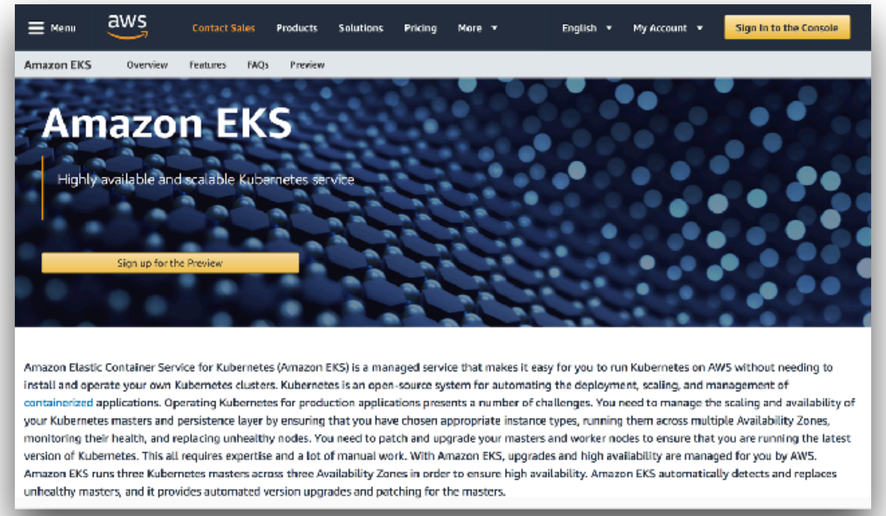
- We will use Minikube to create a Kubernetes Cluster in our VM

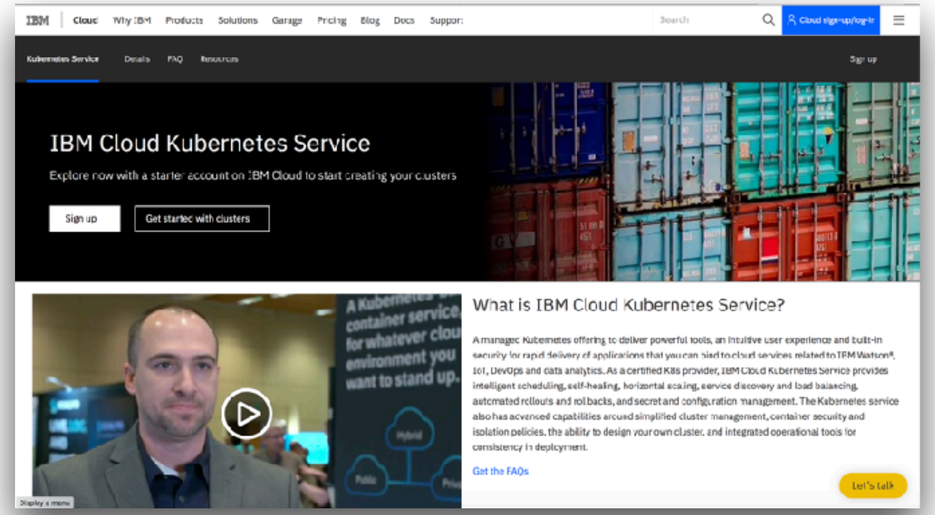
```
$ export CHANGE_MINIKUBE_NONE_USER=true

$ sudo minikube start --vm-driver=none
```

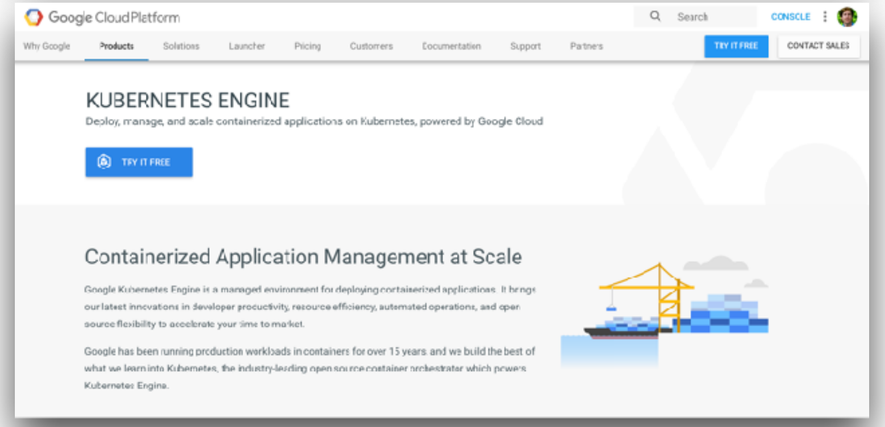
What is Kubernetes?

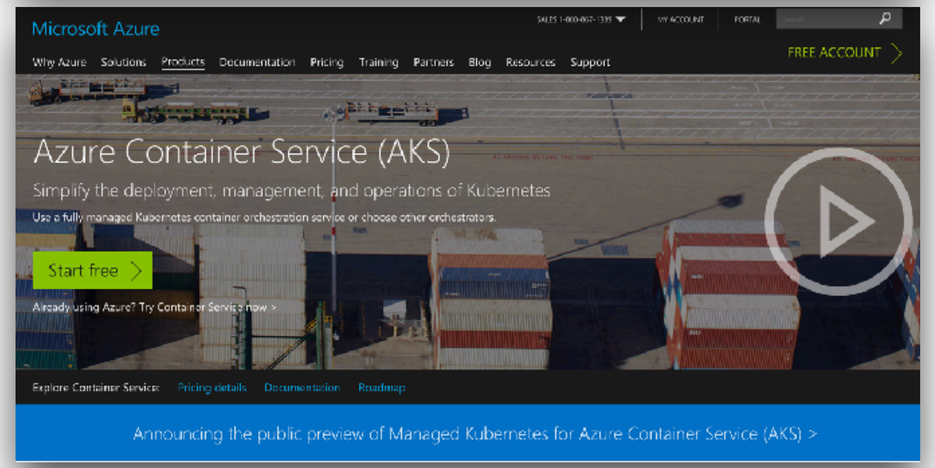
Kubernetes is the de facto Orchestrator for Containers

Every major cloud provider support Kubernetes









Why choose Kubernetes?

- An open-source system for automating deployment, scaling, and management of containerized applications.
- No Vendor Lock-In
- Large Community of support
- Robust platform for container orchestration
- Based on 15 years of experience of running production workloads at Google, combined with best-of-breed ideas and practices from the community

Kubernetes is an Orchestration Platform

- **Scheduling**
  decide where containers run
- **Lifecycle and Health**
  keep containers running and restart them if they fail
- **Scaling**
  grow and shrink deployments as needed
- **Naming and Discovery**
  help containers find each other
- **Load Balancing**
  distribute traffic across containers

  ...and a whole lot more

Why Do You Need Container Orchestration?

- Deploy applications to servers without worrying about specific servers
- Scale the application horizontally up and down
- Restore the application if the server on which it worked fails
  - This is called container auto-healing or rescheduling

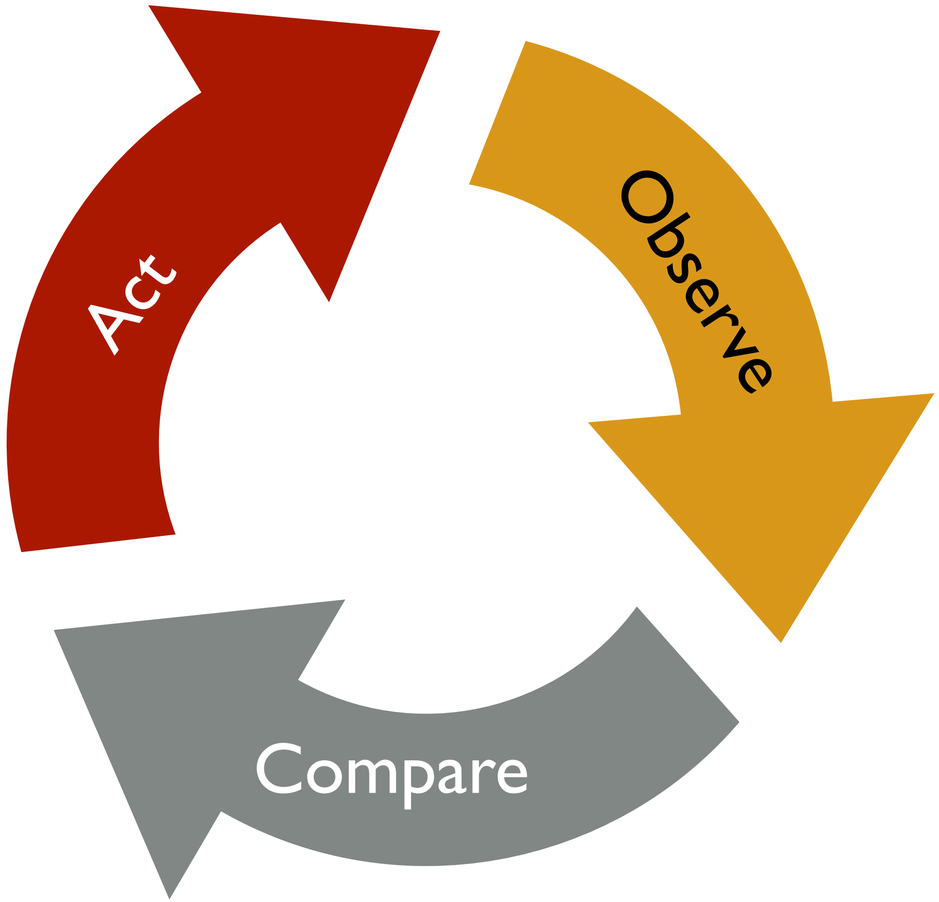
Kubernetes has a Declarative API

Kubernetes is a Declarative Model

You express the desired state

Kubernetes maintains it

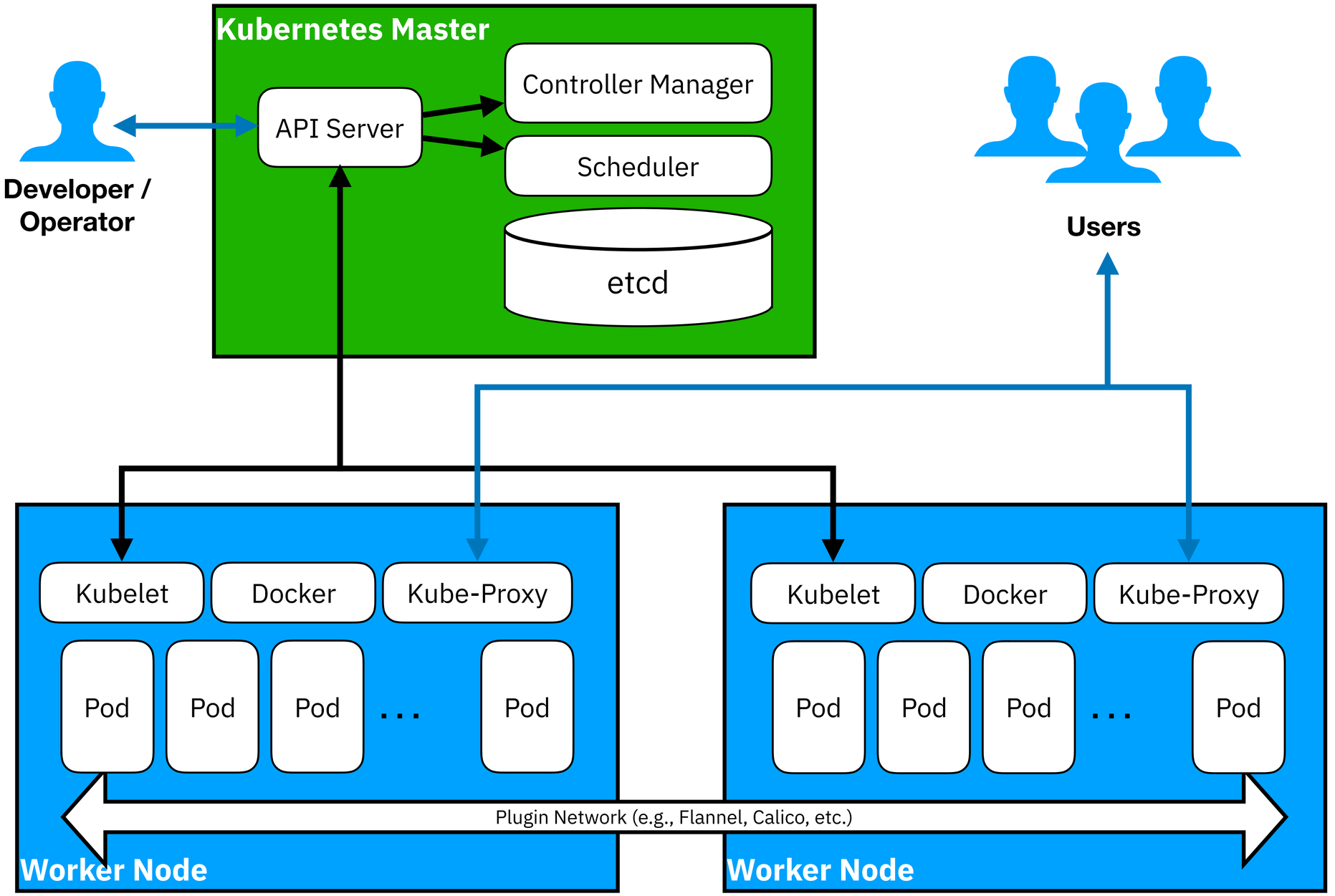
What could be simpler?



Kubernetes Architecture

Kubernetes Architecture

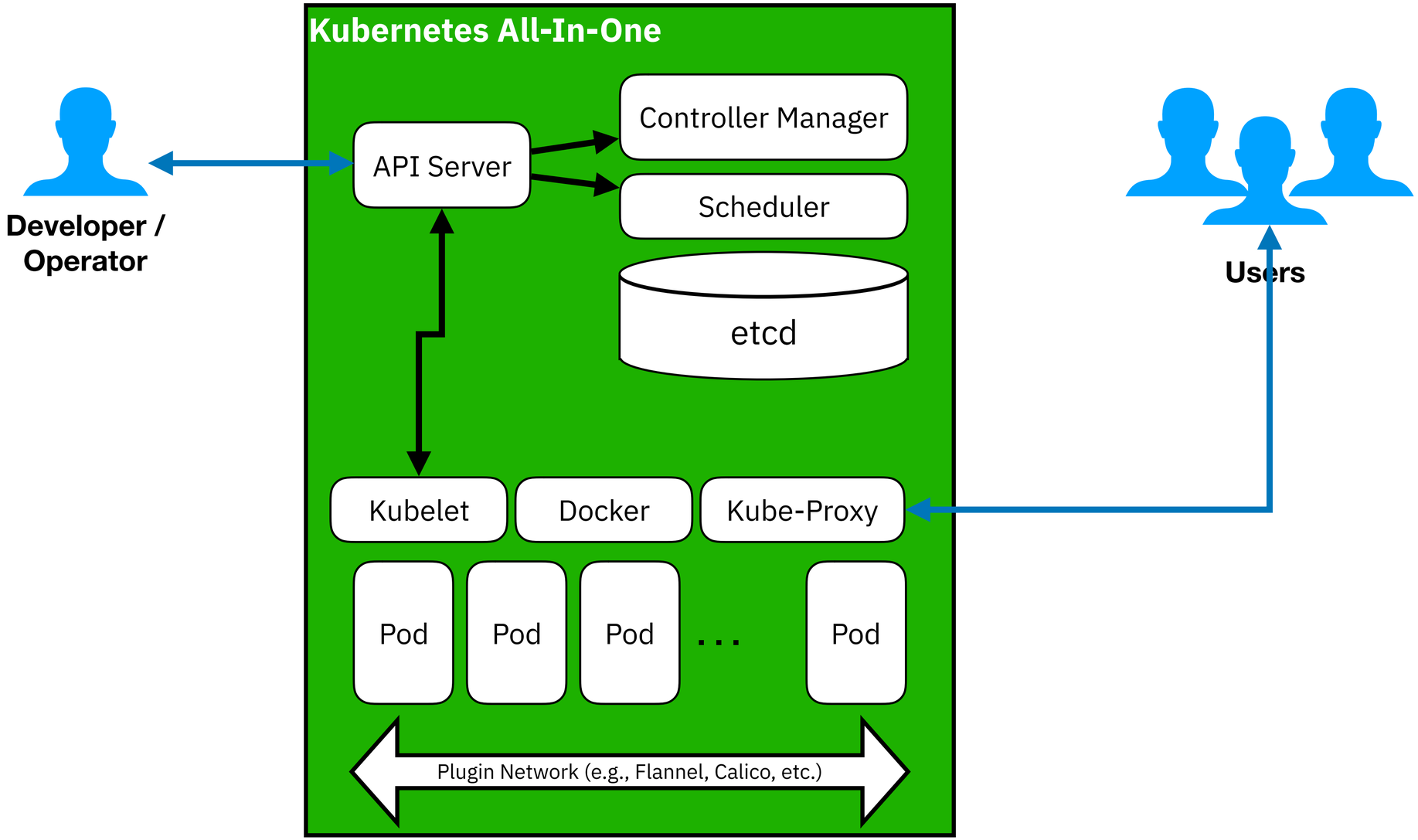
- There can be one or more Master Nodes (HA)
- There can be zero or more Worker Nodes
- Everyone communicates via the API Server
- Kubelet on each Worker Node acting as an agent
- Everything can be on one node for development use



Source: wikipedia

Kubernetes All-In-One

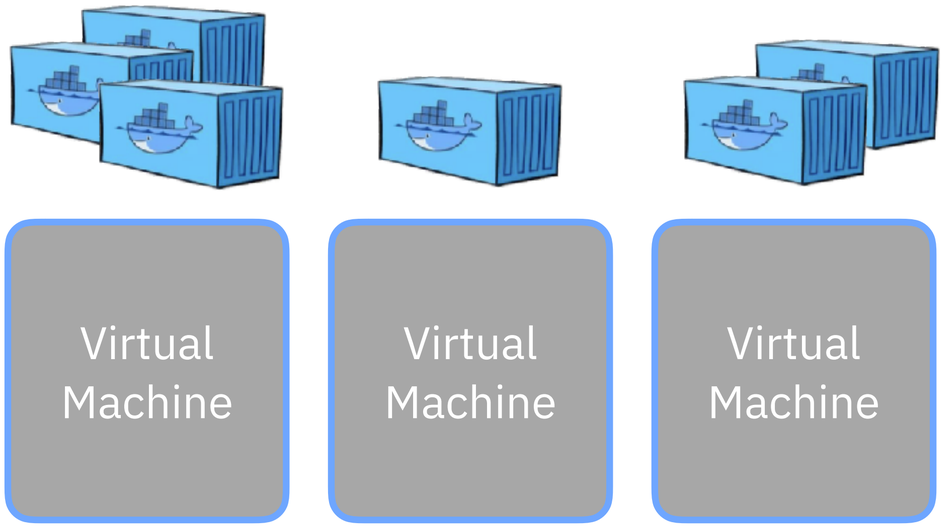
- You can run Kubernetes all in one VM for development work (not for production!)
- **MiniKube** is great for setting this up
- **MiniShift** will deploy a development version of RedHat OpenShift (Origin) which uses Kubernetes



Kubernetes is the new "Cloud OS"

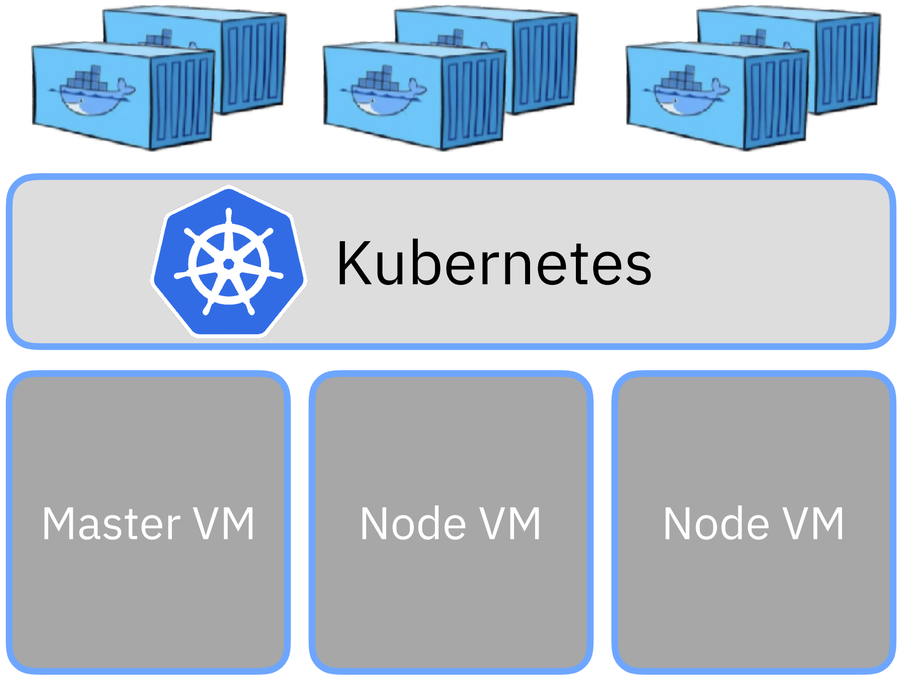
Managing Containers with VMs

- SysAdmins must decide where to place containers
- Workload balancing is manual

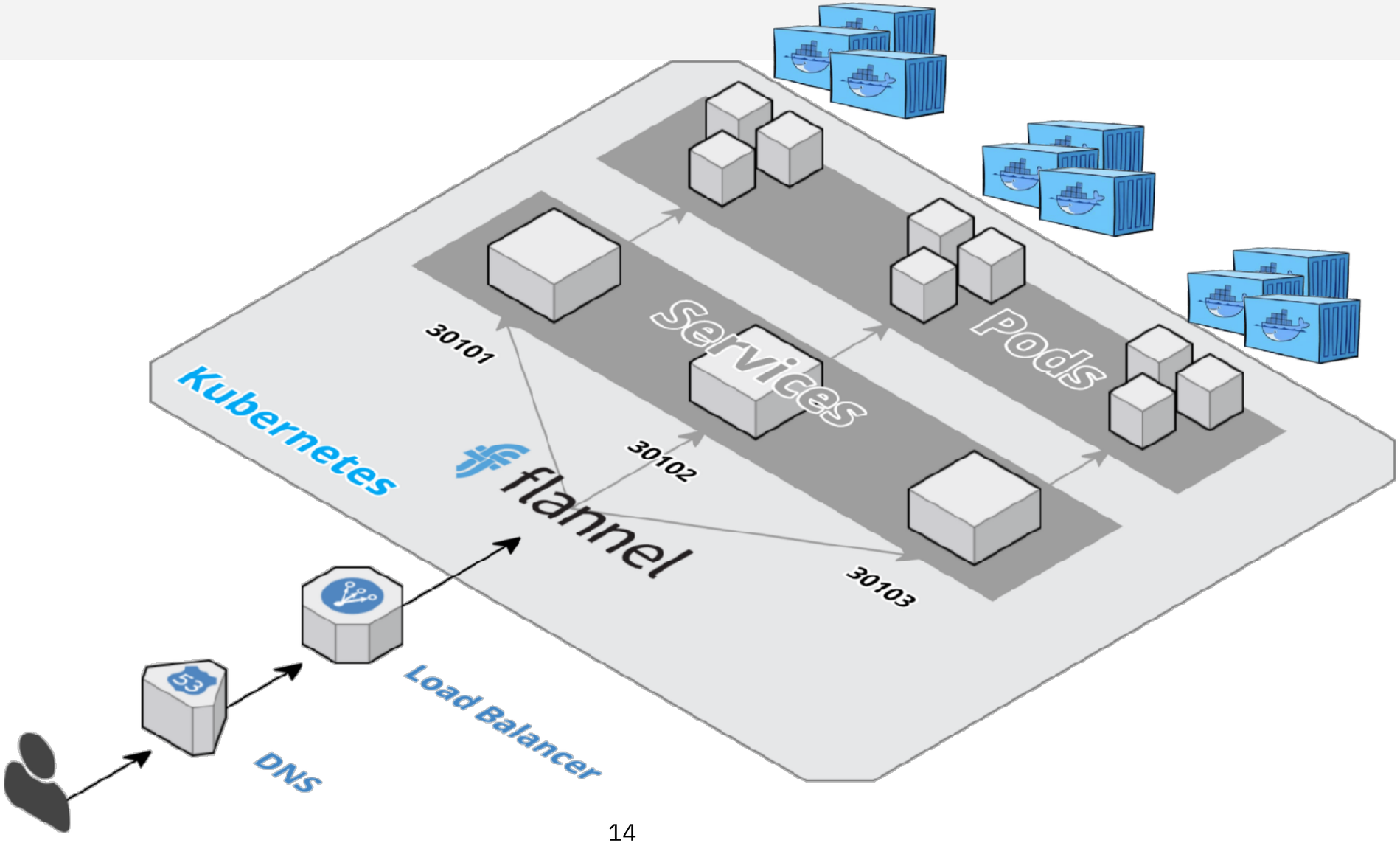


Managing Containers with Kubernetes

- Kubernetes schedules and optimizes workloads automatically



Docker with Kubernetes

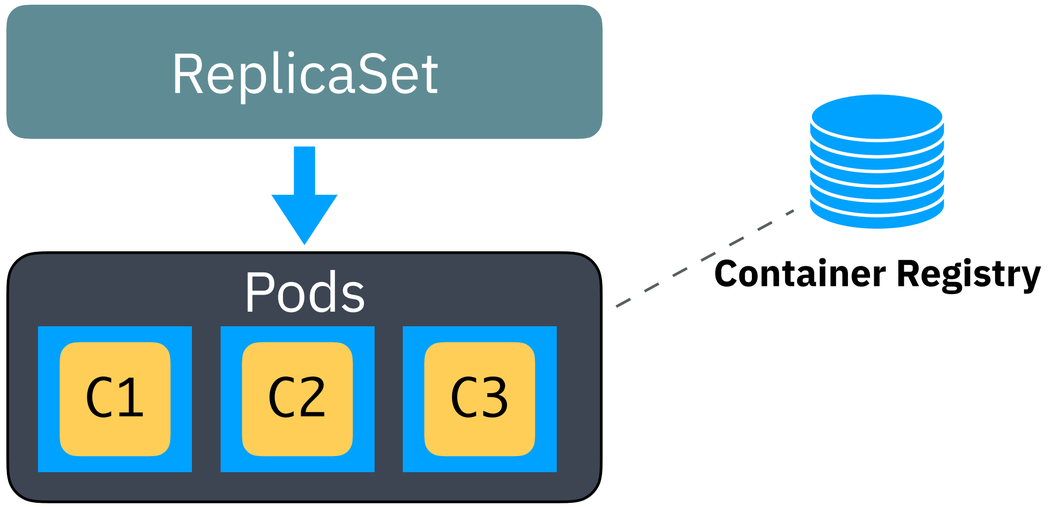


Kubernetes Pods

- Pods
  - Containers run in Pods
  - The Pod is the smallest deployment possible
  - Containers are deployed from a container registry (public or private)

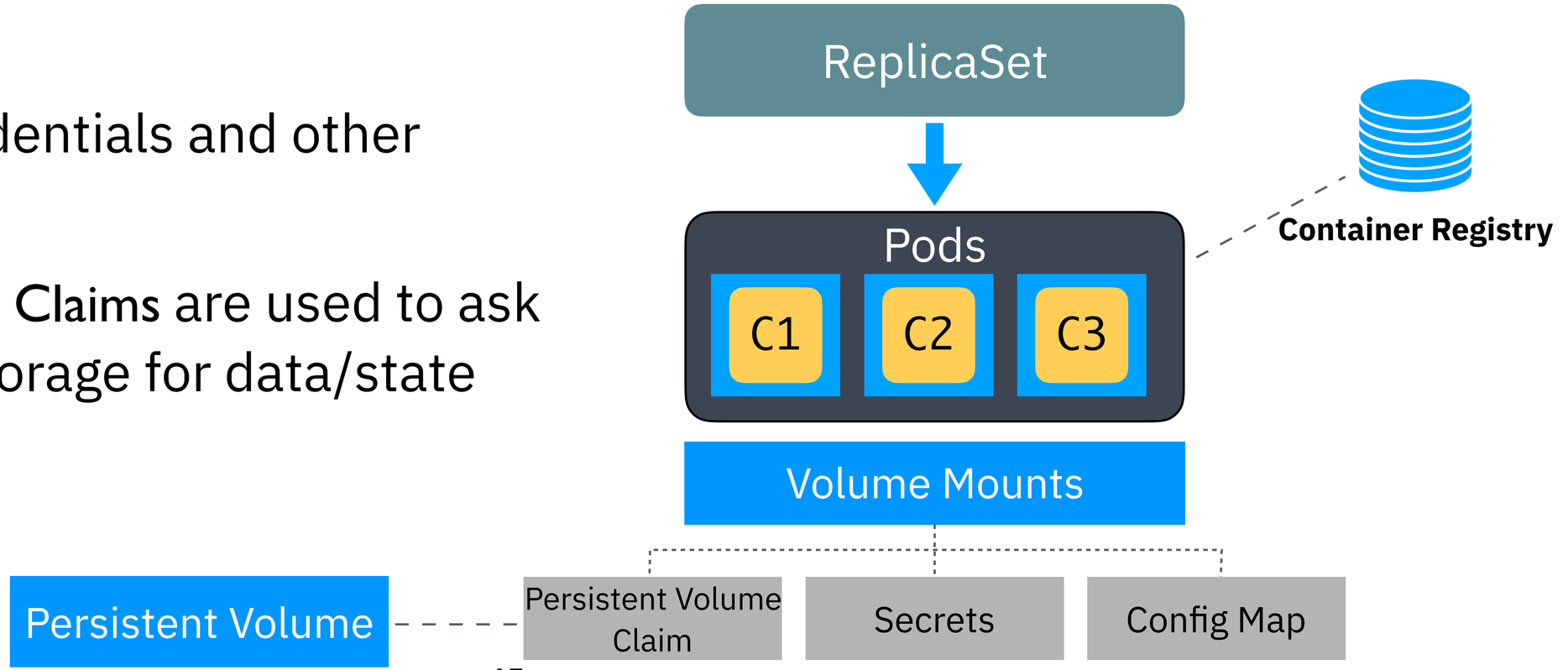
Kubernetes ReplicaSets

- ReplicaSet
  - ReplicaSets allow multiple copies if containers to be deployed
  - Each one in it's own Pod
  - If a container dies, the ReplicaSet will spawn a new one



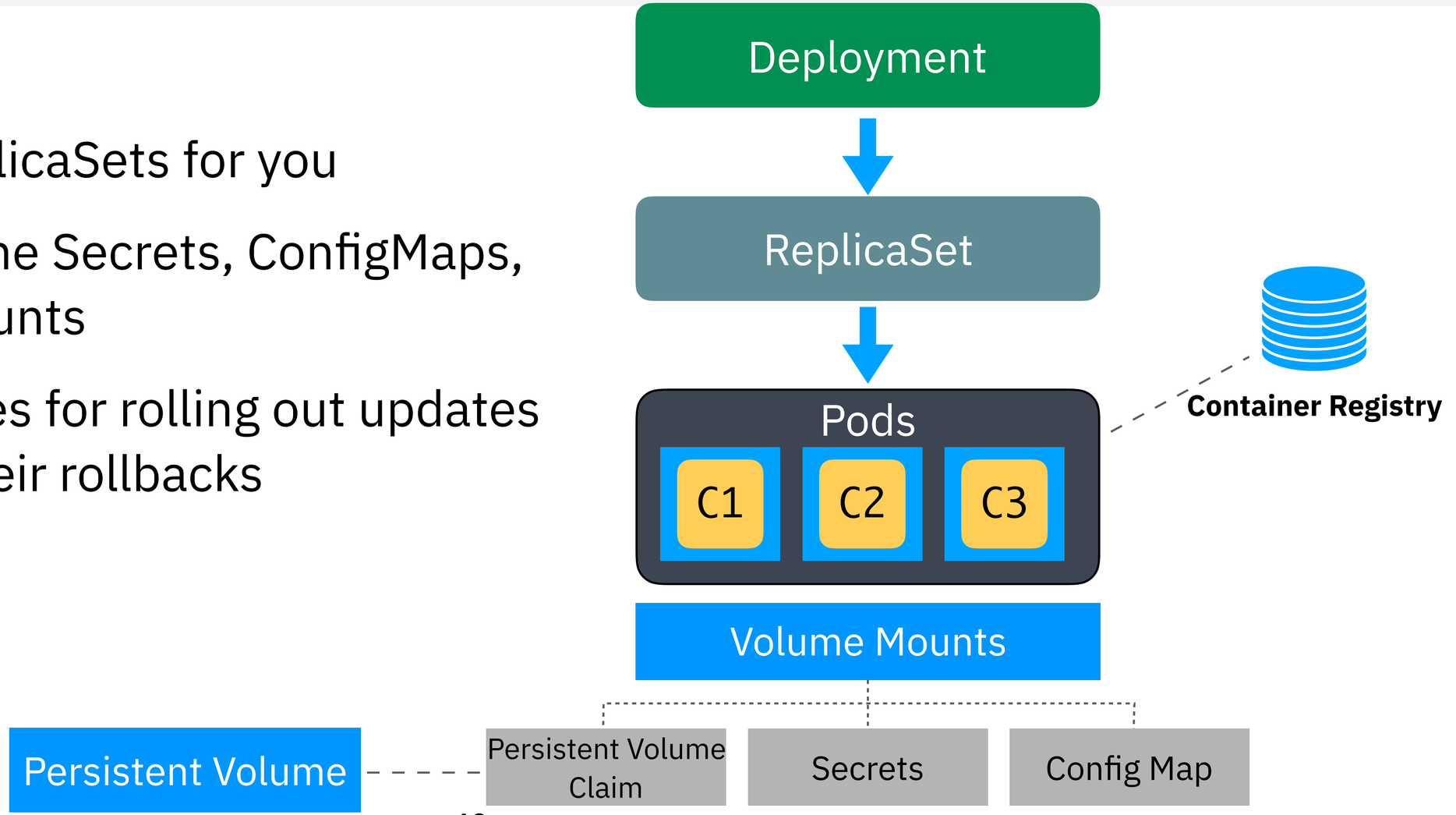
Kubernetes Volume Mounts

- Volumes
  - ConfigMaps hold configuration parameters
  - Secrets hold credentials and other secrets
  - Persistent Volume Claims are used to ask for persistent storage for data/state



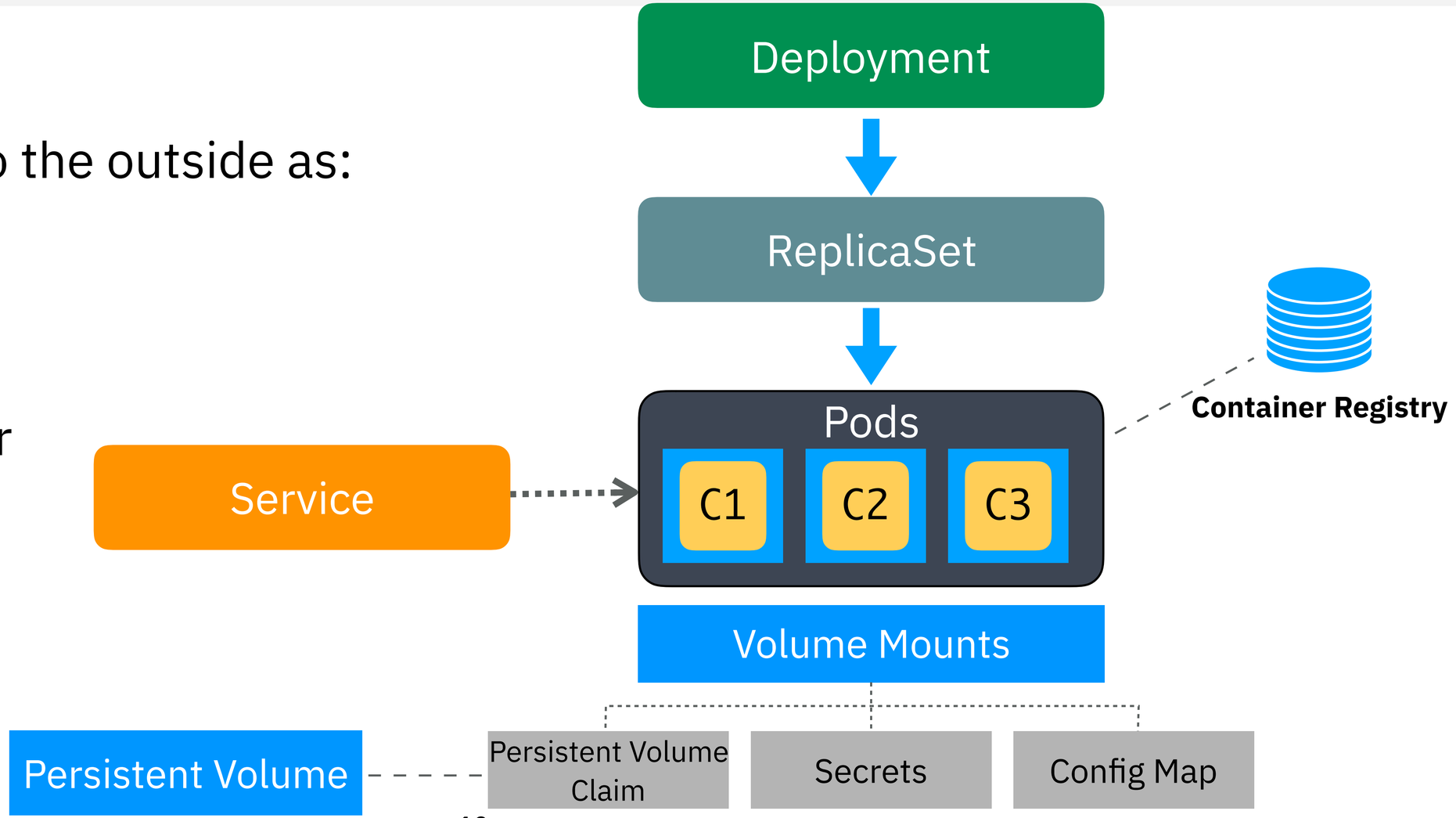
Kubernetes Deployments

- Deployment
  - Sets up the ReplicaSets for you
  - Also specified the Secrets, ConfigMaps, and Volume Mounts
  - Provides features for rolling out updates and handling their rollbacks



Kubernetes Service

- Service
  - Exposes Pods to the outside as:
    - ClusterIP
    - NodePort
    - Load Balancer

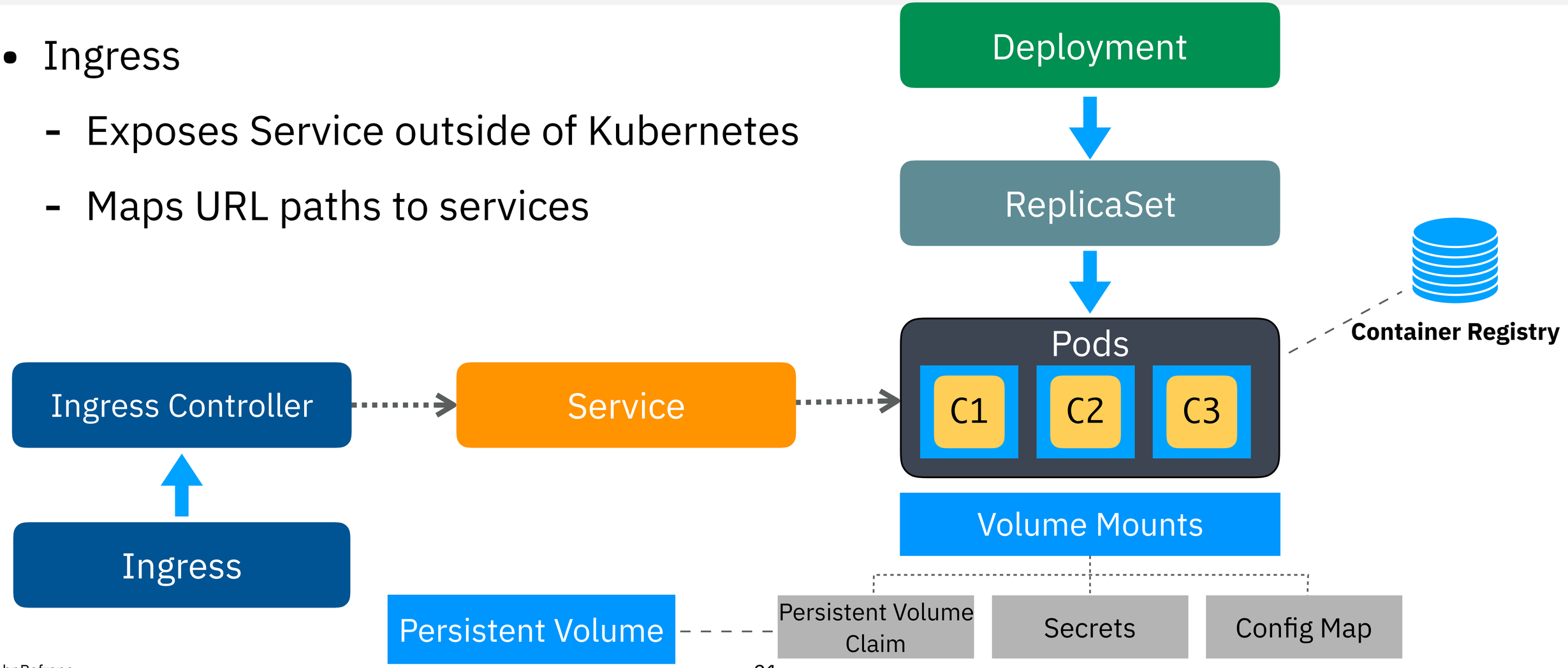


Types of Services

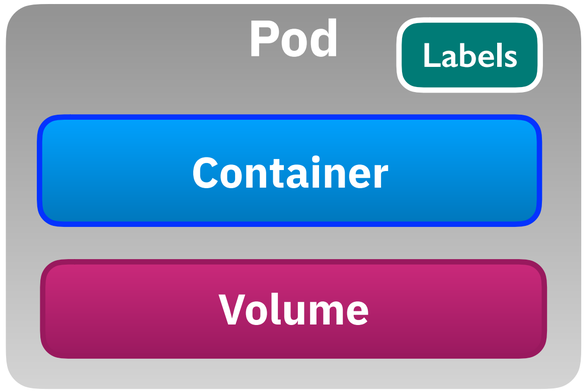
- **LoadBalancer** — Only available via cloud providers. Front end for service that balances the load across multiple backends from a single IP address
- **NodePort** — Exposes the service as an arbitrary port on every worker node in the cluster
- **ClusterIP** — Service is only accessible from other services within the cluster (no external exposure)

Kubernetes Ingress Controller

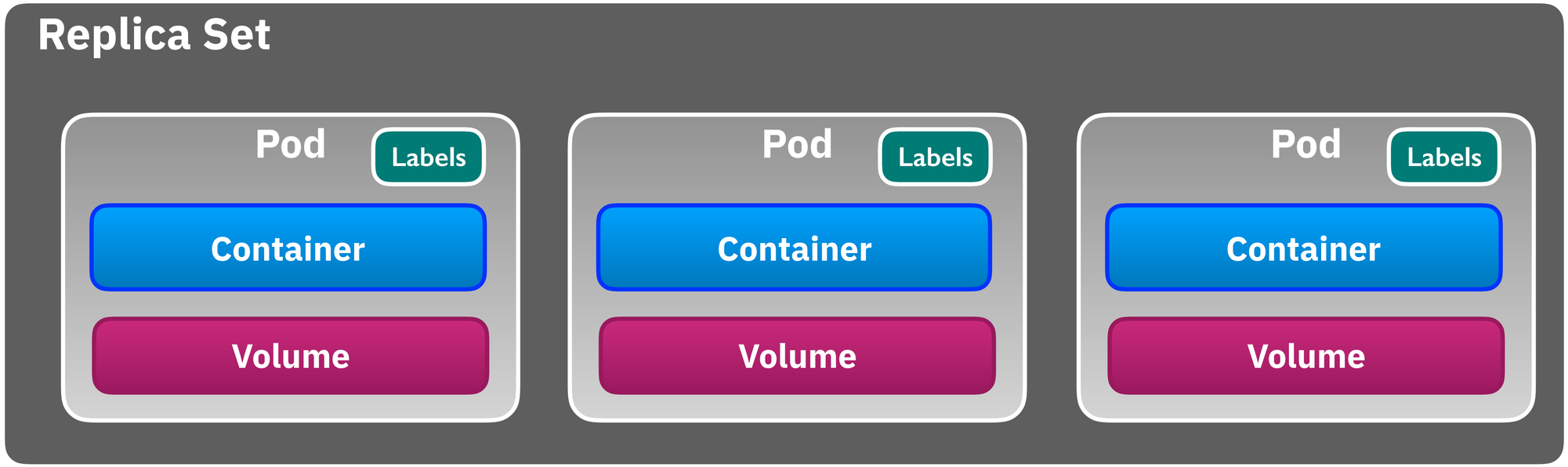
- Ingress
  - Exposes Service outside of Kubernetes
  - Maps URL paths to services



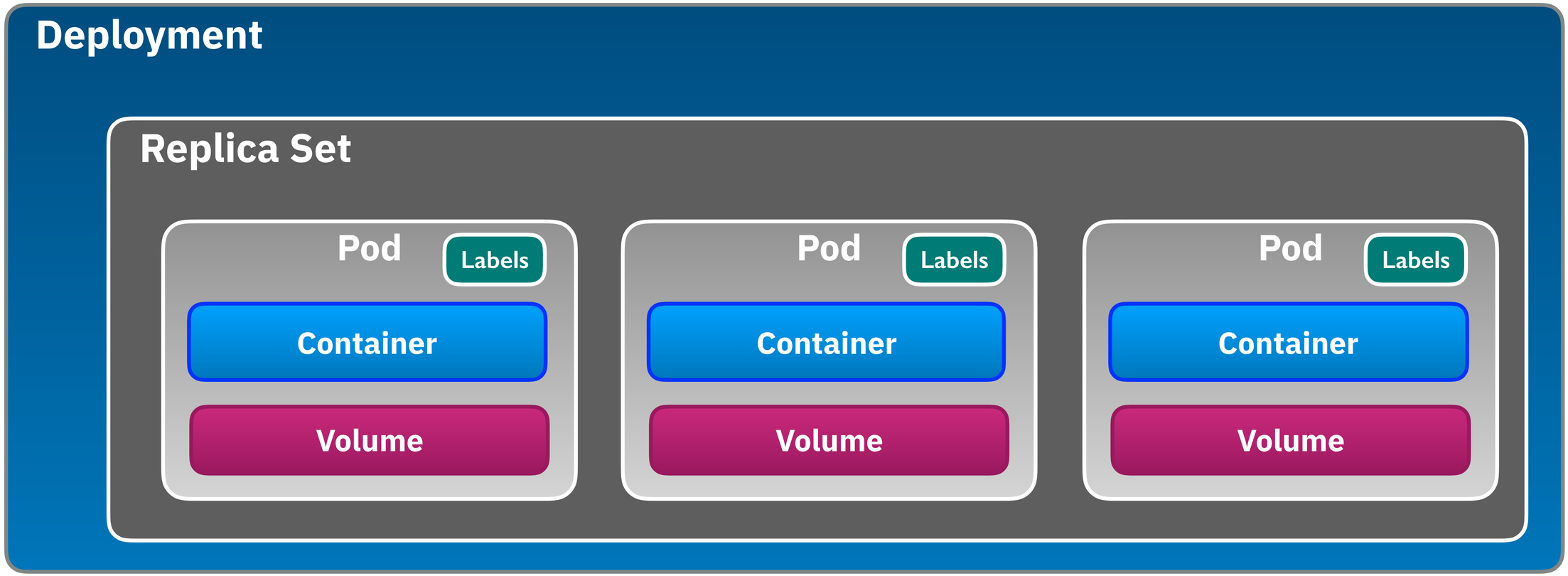
Services Map to Pods via Labels



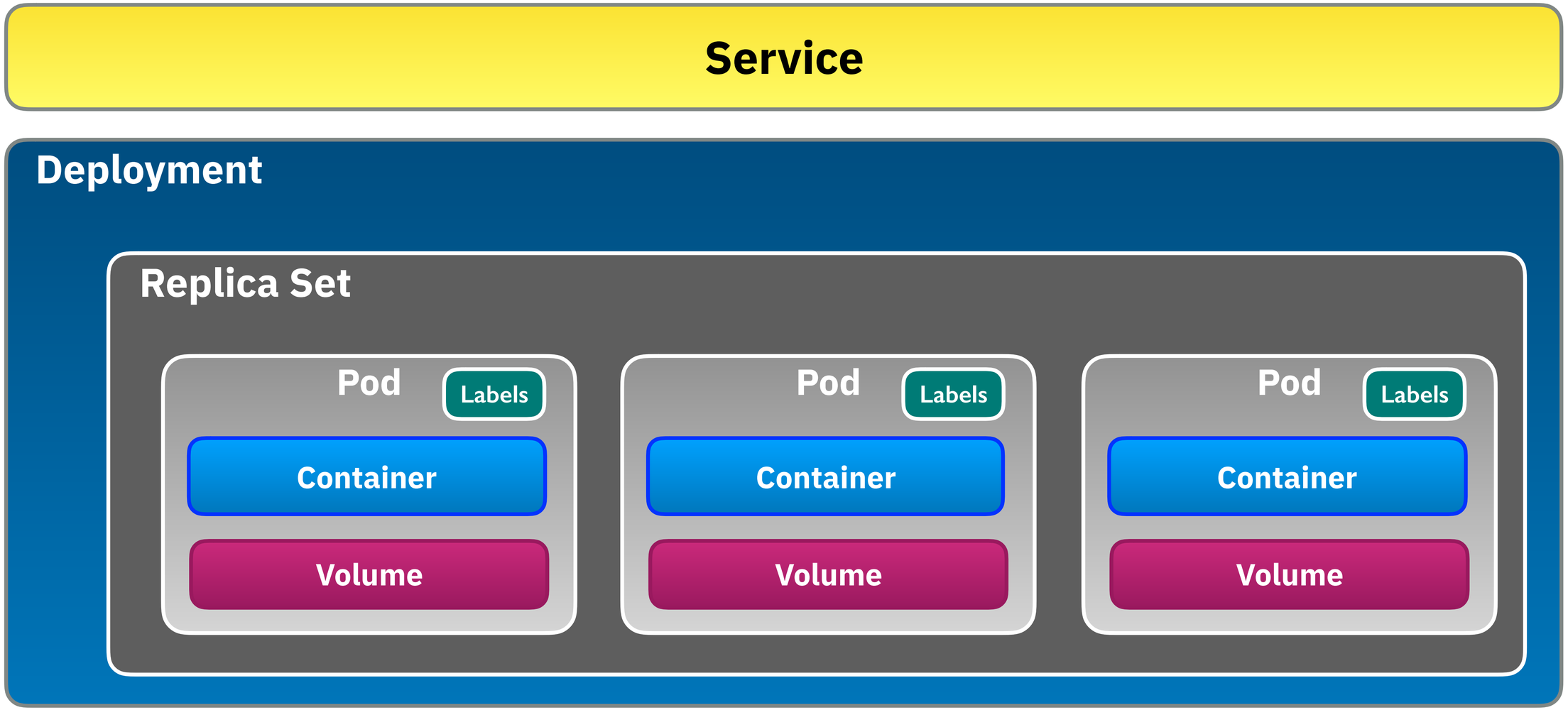
Services Map to Pods via Labels



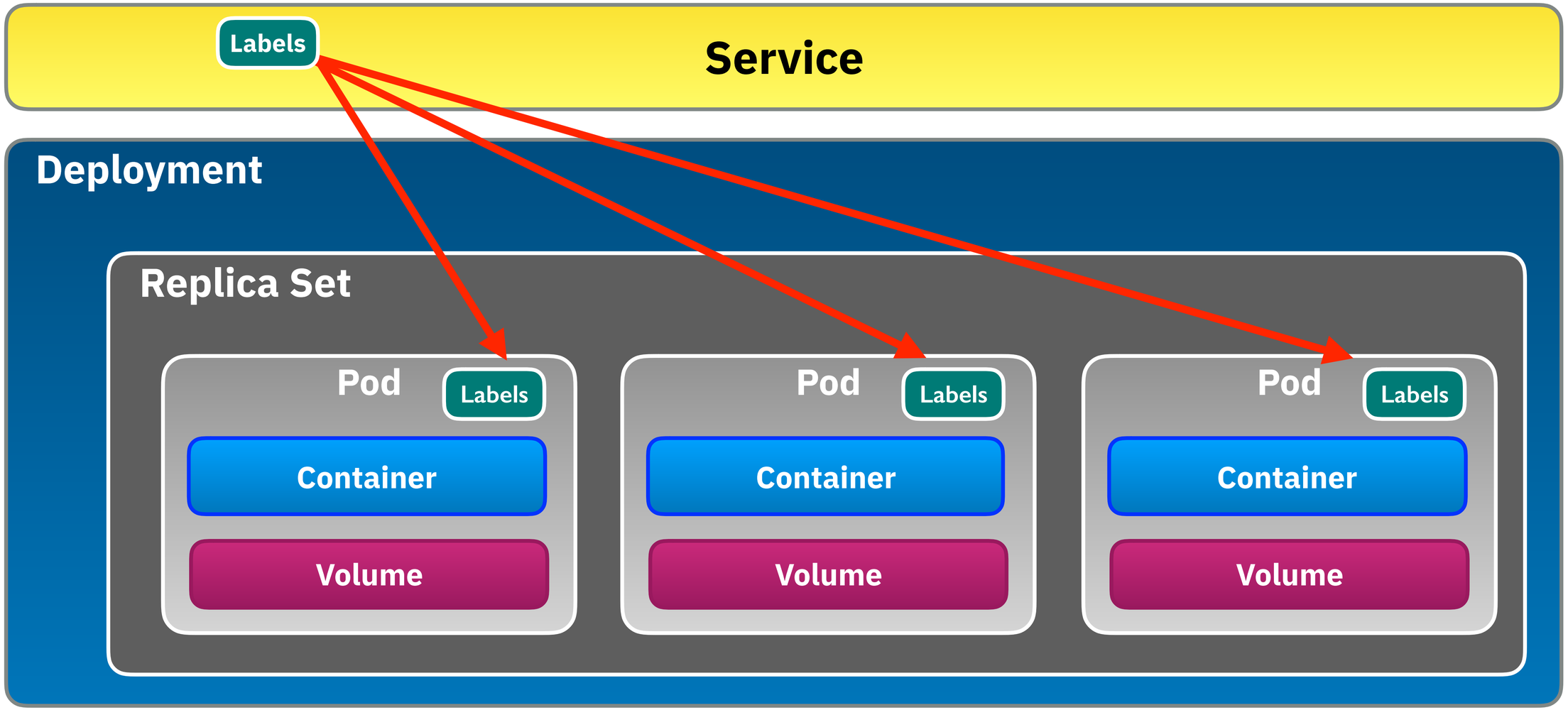
Services Map to Pods via Labels



Services Map to Pods via Labels



Services Map to Pods via Labels



Use Case for Multiple Containers in a Pod

- Pods can be used to host vertically integrated application stacks (e.g. LAMP), but their primary motivation is to support co-located, co-managed helper programs, such as:
  - Content management systems, file and data loaders, local cache managers, etc.
  - Log and checkpoint backup, compression, rotation, snapshotting, etc.
  - Data change watchers, log tailers, logging and monitoring adapters, event publishers, etc.
  - Proxies, bridges, and adapters
  - Controllers, managers, configurators, and updaters

https://kubernetes.io/docs/concepts/workloads/pods/pod/#uses-of-pods

Deployment using kubectl

- A containerized application can be deployed on Kubernetes using a deployment definition by executing a simple CLI command as follows:

```
kubectl create deployment <application-name> --image=<container-image>
```

```
$ kubectl create deployment hello-service --image=hello:v1
```

Create a Service

- You can expose a deployment as a service using `kubectl expose`:

```
kubectl expose deployment <application-name> --type=<service-type> \
                                --port=<port_number>
```

```
$ kubectl expose deployment hello-service --type=NodePort --port=3000
```

Valid <service-type>s are:
- LoadBalancer
- NodePort
- ClusterIP

Deploy using YAML

- You can create a deployment and a service using `yaml` files

```
$ kubectl create -f deployment.yaml
$ kubectl create -f service.yaml
```

- Update them

```
$ kubectl apply -f deployment.yaml
$ kubectl apply -f service.yaml
```

- And just as easily delete them

```
$ kubectl delete -f deployment.yaml
$ kubectl delete -f service.yaml
```

Deployment YAML

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pet-demo
spec:
  replicas: 2
  selector:
    matchLabels:
      app: pet-demo
  template:
    metadata:
      labels:
        app: pet-demo
    spec:
      containers:
      - name: pet-demo
        image: pet-demo:v1
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 5000
          protocol: TCP
      restartPolicy: Always
```

Service YAML

```yaml
apiVersion: v1
kind: Service
metadata:
  name: pet-demo-service
spec:
  type: NodePort
  selector:
    App: pet-demo
  ports:
    - name: primary
      protocol: TCP
      port: 5000
```

Linking Service to Pods

```yaml
apiVersion: v1
kind: Service
metadata:
  name: pet-demo-service
spec:
  type: NodePort
  selector:
    app: pet-demo
  ports:
    - name: primary
      protocol: TCP
      port: 5000
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pet-demo
spec:
  replicas: 2
  selector:
    matchLabels:
      app: pet-demo
  template:
    metadata:
      labels:
        app: pet-demo
    spec:
      containers:
      - name: pet-demo
        image: pet-demo:v1
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 5000
          protocol: TCP
      restartPolicy: Always
```

Linking Service to Pods

```yaml
apiVersion: v1
kind: Service
metadata:
  name: pet-demo-service
spec:
  type: NodePort
  selector:
    app: pet-demo
  ports:
    - name: primary
      protocol: TCP
      port: 5000
```

Look for Pods with this Label

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pet-demo
spec:
  replicas: 2
  selector:
    matchLabels:
      app: pet-demo
  template:
    metadata:
      labels:
        app: pet-demo
    spec:
      containers:
      - name: pet-demo
        image: pet-demo:v1
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 5000
          protocol: TCP
      restartPolicy: Always
```

Linking Service to Pods

Will match this Pod

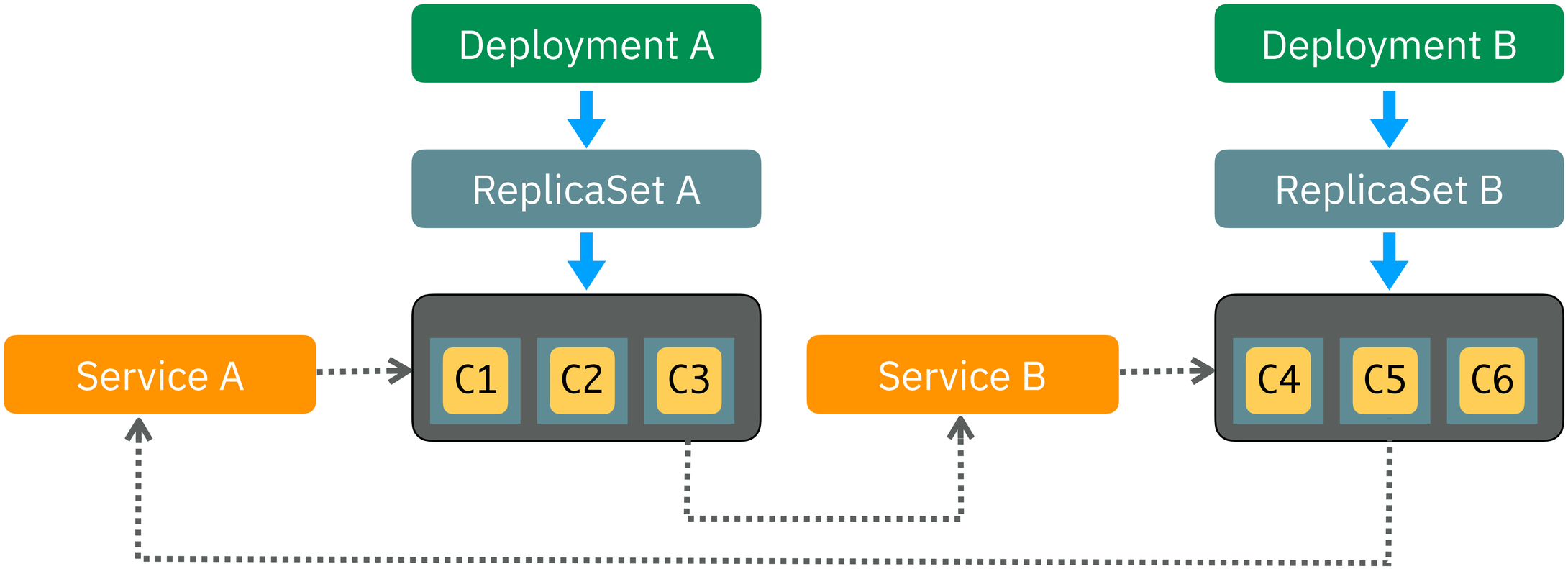
Look for Pods with this Label

```yaml
apiVersion: v1
kind: Service
metadata:
  name: pet-demo-service
spec:
  type: NodePort
  selector:
    app: pet-demo
  ports:
    - name: primary
      protocol: TCP
      port: 5000
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pet-demo
spec:
  replicas: 2
  selector:
    matchLabels:
      app: pet-demo
  template:
    metadata:
      labels:
        app: pet-demo
    spec:
      containers:
      - name: pet-demo
        image: pet-demo:v1
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 5000
          protocol: TCP
      restartPolicy: Always
```

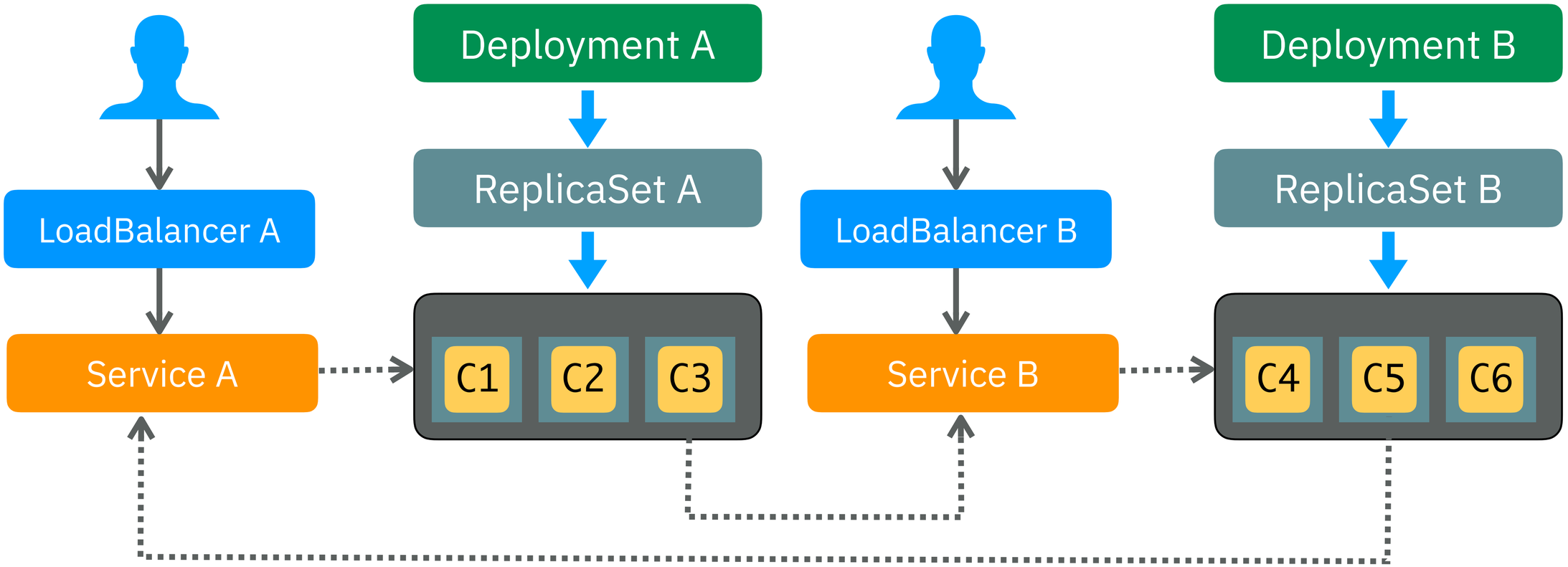
Service Discovery

- Kubernetes provides internal routing so services can find each other



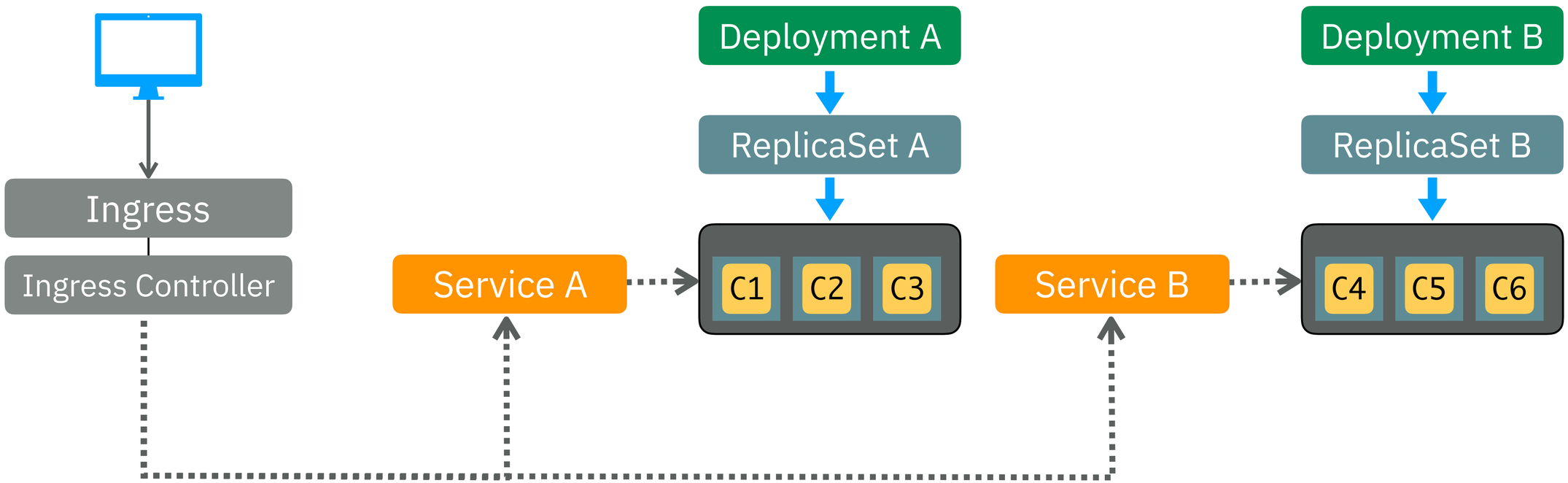
External Service Access

- LoadBalancers provide external access for a single service



External Routing

- Kubernetes provides an Ingress Controller to allow external network access or you can use NodePorts



Ingress Example

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ecommerce
spec:
  # tls:
  # - secretName: tls
  rules:
  - host: ecommerce.mycompany.com
    http:
      paths:
      - path: /shopcarts
        backend:
          serviceName: shopcart-service
          servicePort: 5000
      - path: /catalog
        backend:
          serviceName: catalog-service
          servicePort: 5000
      - path: /orders
        backend:
          serviceName: order-service
          servicePort: 5000
      - path: /recommedations
        backend:
          serviceName: recommedation-service
          servicePort: 5000
```

Ingress Example

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ecommerce
spec:
  # tls:
  # - secretName: tls
  rules:
  - host: ecommerce.mycompany.com
    http:
      paths:
      - path: /shopcarts
        backend:
          serviceName: shopcart-service
          servicePort: 5000
      - path: /catalog
        backend:
          serviceName: catalog-service
          servicePort: 5000
      - path: /orders
        backend:
          serviceName: order-service
          servicePort: 5000
      - path: /recommedations
        backend:
          serviceName: recommedation-service
          servicePort: 5000
```

Ingress Example

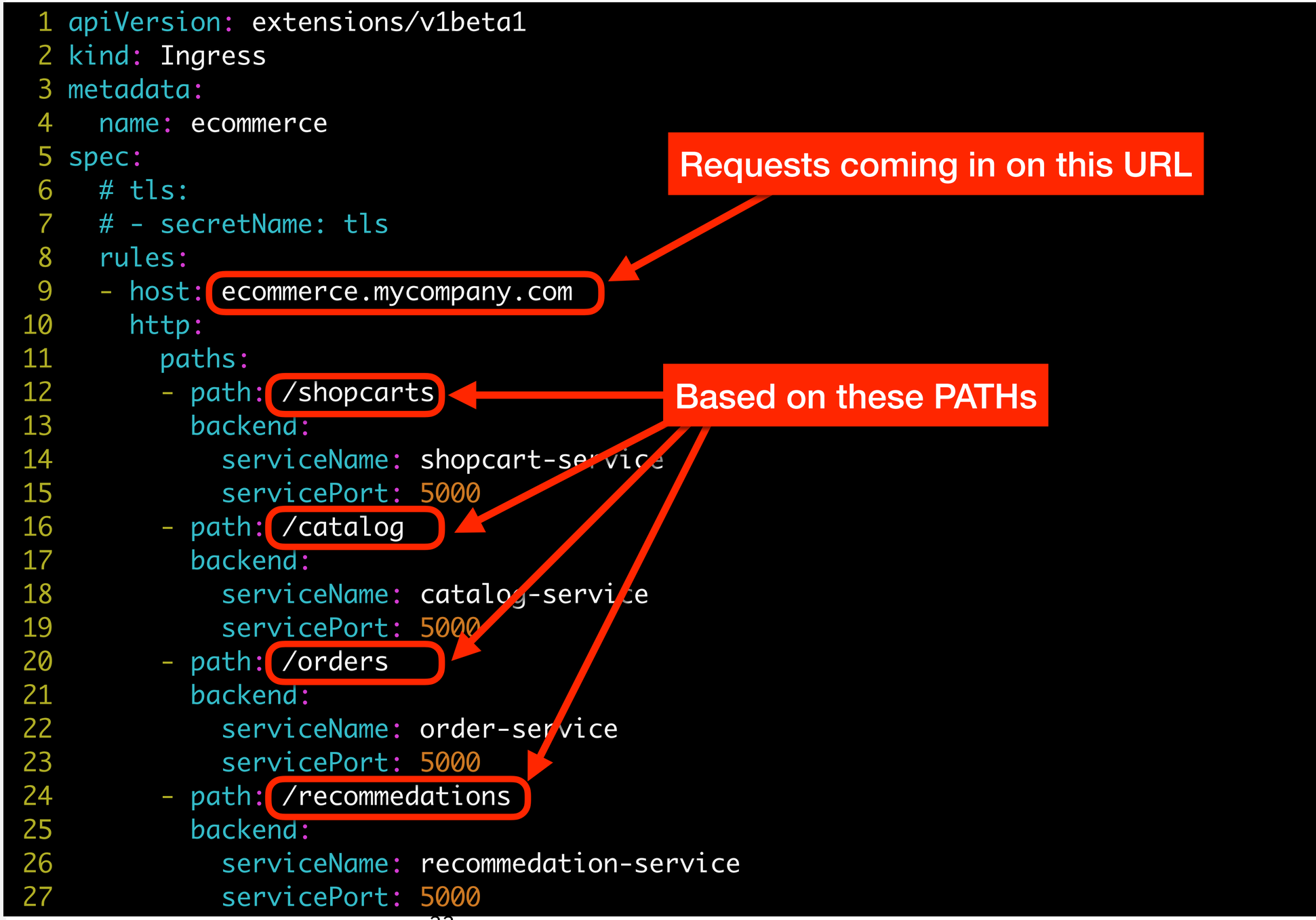
```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ecommerce
spec:
  # tls:
  # - secretName: tls
  rules:
  - host: ecommerce.mycompany.com
    http:
      paths:
      - path: /shopcarts
        backend:
          serviceName: shopcart-service
          servicePort: 5000
      - path: /catalog
        backend:
          serviceName: catalog-service
          servicePort: 5000
      - path: /orders
        backend:
          serviceName: order-service
          servicePort: 5000
      - path: /recommedations
        backend:
          serviceName: recommedation-service
          servicePort: 5000
```

Requests coming in on this URL

Will be routed to these microservices

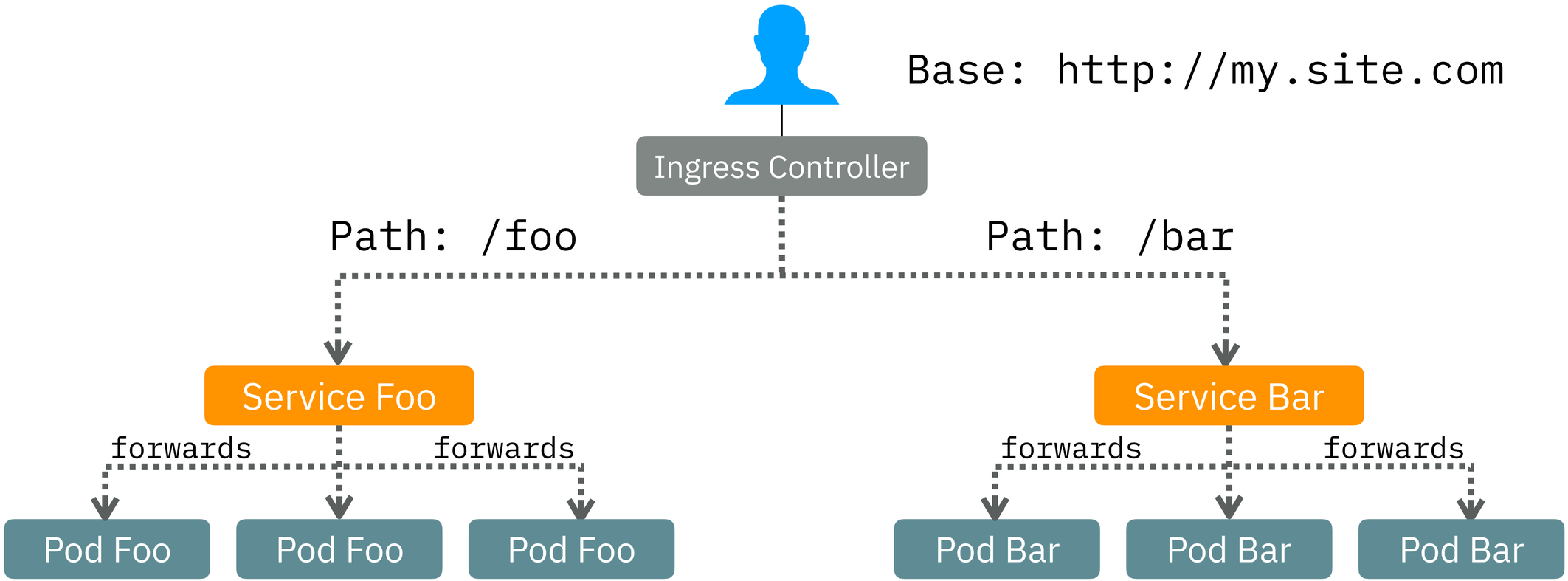
Ingress Example

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ecommerce
spec:
  # tls:
  # - secretName: tls
  rules:
  - host: ecommerce.mycompany.com
    http:
      paths:
      - path: /shopcarts
        backend:
          serviceName: shopcart-service
          servicePort: 5000
      - path: /catalog
        backend:
          serviceName: catalog-service
          servicePort: 5000
      - path: /orders
        backend:
          serviceName: order-service
          servicePort: 5000
      - path: /recommedations
        backend:
          serviceName: recommedation-service
          servicePort: 5000
```



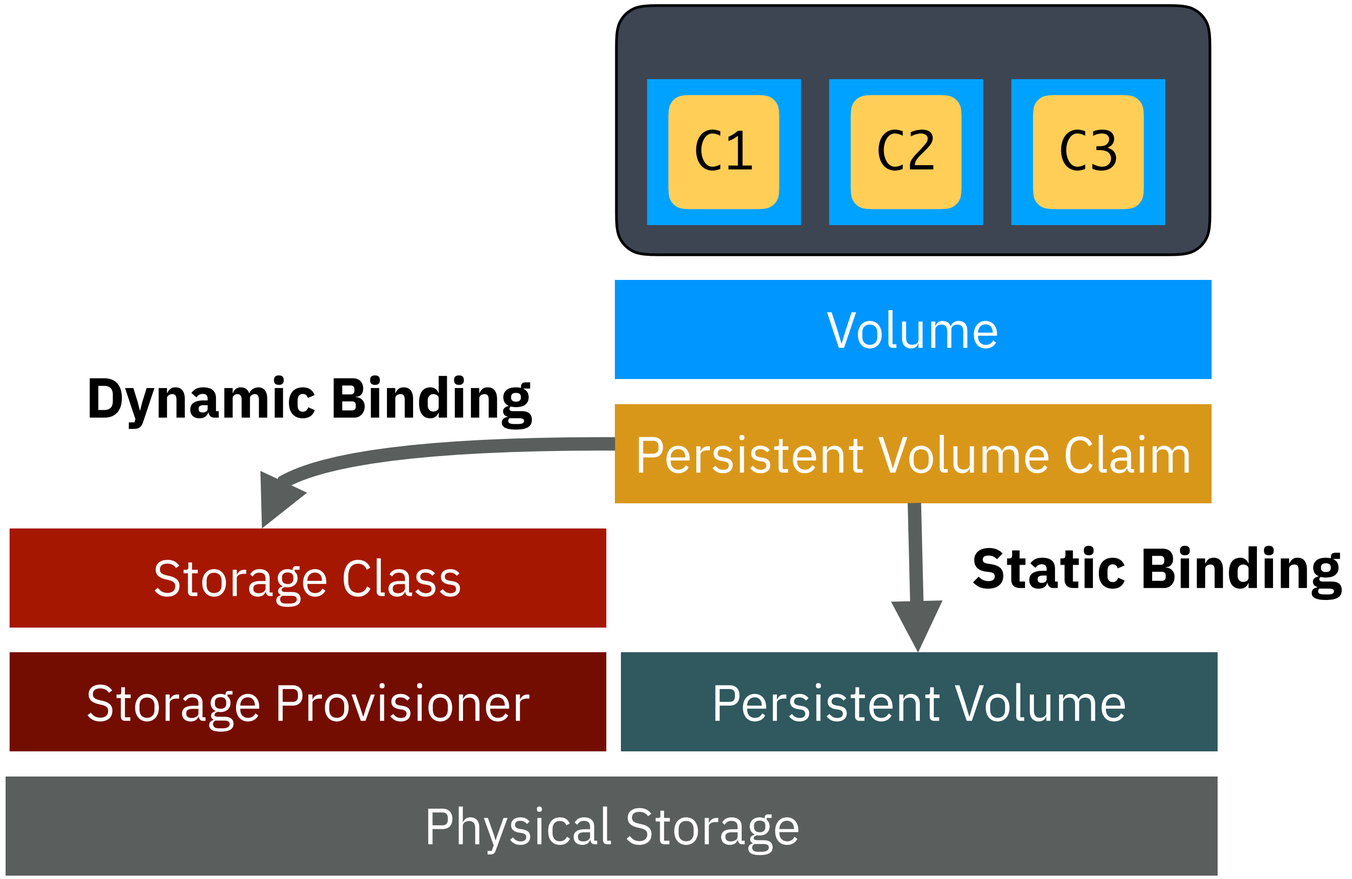
Ingress Controller

- Single entry point into multiple kubernetes services



Persistent Volumes

- Kubernetes loosely couples physical storage devices with containers by introducing an intermediate resource called persistent volume claims (PVCs).
- A PVC defines the disk size, disk type (ReadWriteOnce, ReadOnlyMany, ReadWriteMany) and dynamically links a storage device to a volume defined against a pod
- The binding process can either be done in a static way using PVs or dynamically be using a persistent storage provider



Example Volume Mount

- There is a volume named `redis-storage` and that is connected to the `redis` container via the VolumeMount: `/data/redis`

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis
    volumeMounts:
    - name: redis-storage
      mountPath: /data/redis
  volumes:
  - name: redis-storage
    emptyDir: {}
~
```

Example Volume Mount

- There is a volume named `redis-storage` and that is connected to the `redis` container via the VolumeMount: `/data/redis`

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis
    volumeMounts:
    - name: redis-storage
      mountPath: /data/redis
  volumes:
  - name: redis-storage
    emptyDir: {}
~
```

Volume named `redis-storage`

Example Volume Mount

- There is a volume named `redis-storage` and that is connected to the `redis` container via the VolumeMount: `/data/redis`

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis
    volumeMounts:
    - name: redis-storage
      mountPath: /data/redis
  volumes:
  - name: redis-storage
    emptyDir: {}
~
```

Mounted at: `/data/redis`

Volume named `redis-storage`

Configurations Management

(ConfigMap)

- Containers generally use environment variables for parameterizing their runtime configurations
- Kubernetes provides a way of managing more complex configuration files using a simple resource called ConfigMaps
- ConfigMaps can be created using directories, files or literal values using following CLI command:

```
$ kubectl create configmap <map-name> <data-source>

# map-name: name of the config map
# data-source: directory, file or literal value
```

Credentials Management

(Secrets)

- Similar to ConfigMaps, Kubernetes provides another valuable resource called Secrets for managing sensitive information such as passwords, OAuth tokens, and ssh keys.
- A secret can be created for managing basic auth credentials using the following way:

```
# write credentials to two files
$ echo -n 'admin' > ./username.txt
$ echo -n '1f2d1e2e67df' > ./password.txt

# create a secret
$ kubectl create secret generic app-credentials --from-file=./
username.txt --from-file=./password.txt
```

Secret Yaml Example

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: ecommerce-apikey
  namespace: default
data:
  secret: <place base64 encoded secret here>
```

```
$ echo -n "this is my secret" | base64
dGhpcyBpcyBteSBzZWNyZXQ=
```

Secret Yaml Example

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: ecommerce-apikey
  namespace: default
data:
  secret: <place base64 encoded secret here>
```

Place the base64 encode string in secret

```
$ echo -n "this is my secret" | base64
dGhpcyBpcyBteSBzZWNyZXQ=
```

Using Secrets

```yaml
apiVersion: apps/v1
kind: Deployment
...
    spec:
        containers:
        - name: pet-demo
          image: pet-demo:v1
          imagePullPolicy: IfNotPresent
          ports:
          - containerPort: 5000
            protocol: TCP
          env:
            - name: DATABASE_URI
              valueFrom:
                secretKeyRef:
                  name: pet-creds
                  key: binding
```

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: pet-creds
  namespace: default
data:
  binding: dGhpcyBpcyBteSBzZWNyZXQ=
```

Using Secrets

```
apiVersion: apps/v1
kind: Deployment
...
    spec:
        containers:
        - name: pet-demo
          image: pet-demo:v1
          imagePullPol
          ports:
          - containerPort: 5000
            protocol: TCP
          env:
            - name: DATABASE_URI
              valueFrom:
                secretKeyRef:
                  name: pet-creds
                  key: binding
```

Secrets exposed as environment variables

```
apiVersion: v1
kind: Secret
metadata:
  name: pet-creds
  namespace: default
data:
  binding: dGhpcyBpcyBteSBzZWNyZXQ=
```

Kubernetes Rolling Updates (Zero Downtime Deployments)

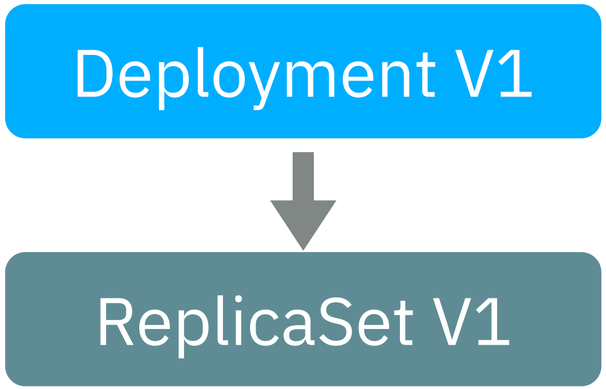
```
$ kubectl set image deployment/<application-name> <container-name>=<container-image-name>:<new-version>
```

Deployment V1

Deploying Version 1

Kubernetes Rolling Updates (Zero Downtime Deployments)

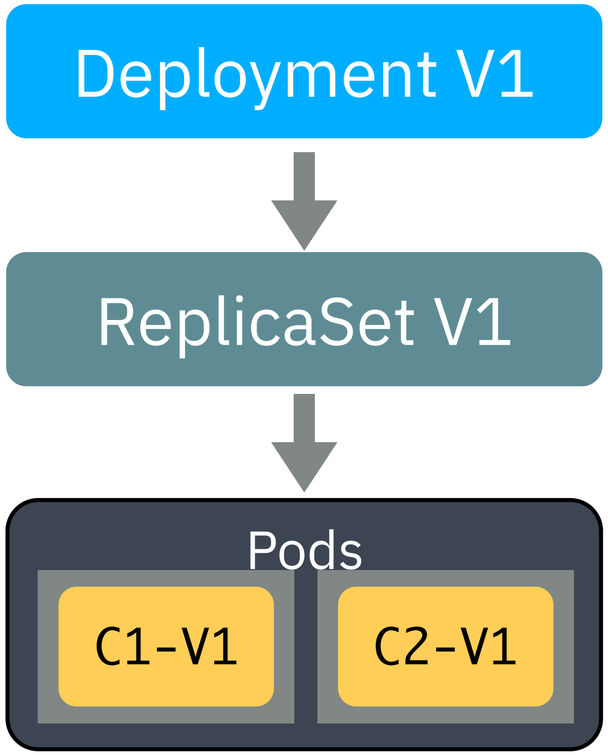
```
$ kubectl set image deployment/<application-name> <container-name>=<container-image-name>:<new-version>
```



Deploying Version 1

Kubernetes Rolling Updates (Zero Downtime Deployments)

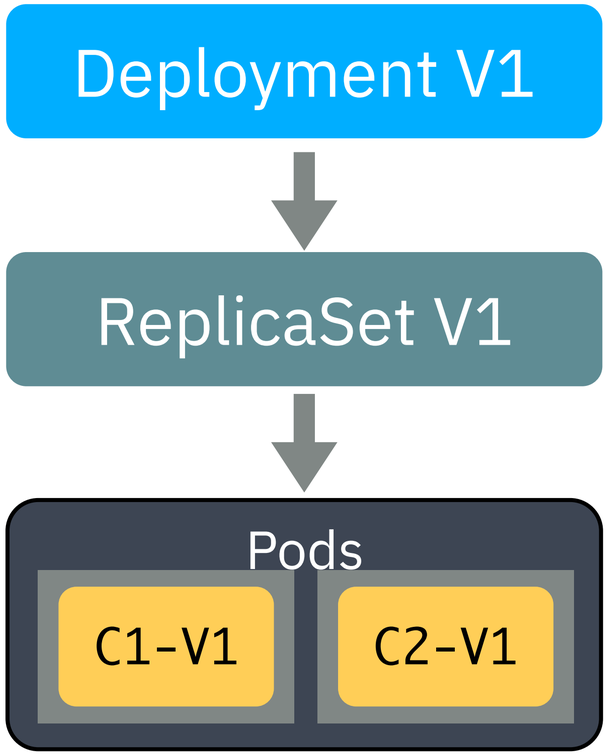
```
$ kubectl set image deployment/<application-name> <container-name>=<container-image-name>:<new-version>
```



Deploying Version 1

Kubernetes Rolling Updates (Zero Downtime Deployments)

```
$ kubectl set image deployment/<application-name> <container-name>=<container-image-name>:<new-version>
```

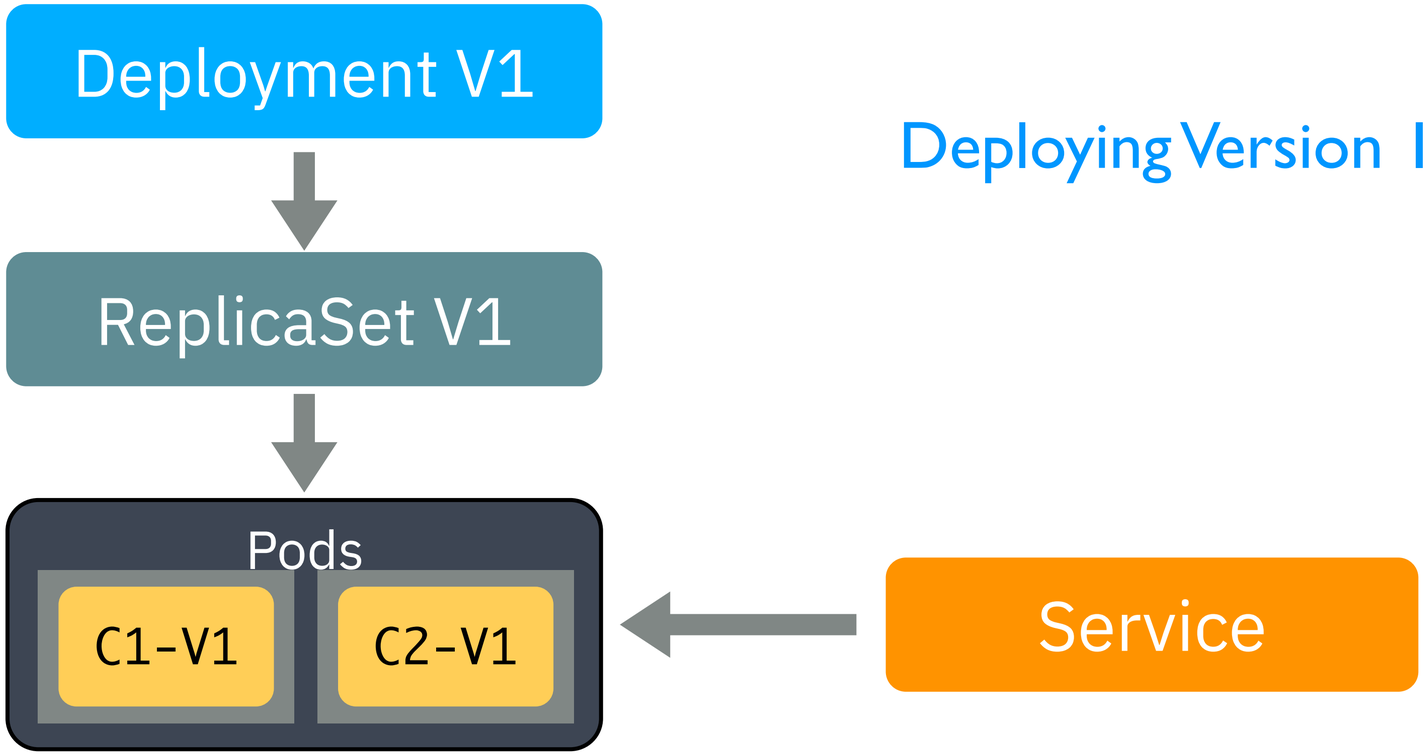


Deploying Version 1

Service

Kubernetes Rolling Updates (Zero Downtime Deployments)

```
$ kubectl set image deployment/<application-name> <container-name>=<container-image-name>:<new-version>
```



Deploying Version 1

Kubernetes Rolling Updates (Zero Downtime Deployments)

```
$ kubectl set image deployment/<application-name> <container-name>=<container-image-name>:<new-version>
```

Kubernetes Rolling Updates (Zero Downtime Deployments)

```
$ kubectl set image deployment/<application-name> <container-name>=<container-image-name>:<new-version>
```

Kubernetes Rolling Updates (Zero Downtime Deployments)

```
$ kubectl set image deployment/<application-name> <container-name>=<container-image-name>:<new-version>
```

Kubernetes Rolling Updates (Zero Downtime Deployments)

```
$ kubectl set image deployment/<application-name> <container-name>=<container-image-name>:<new-version>
```

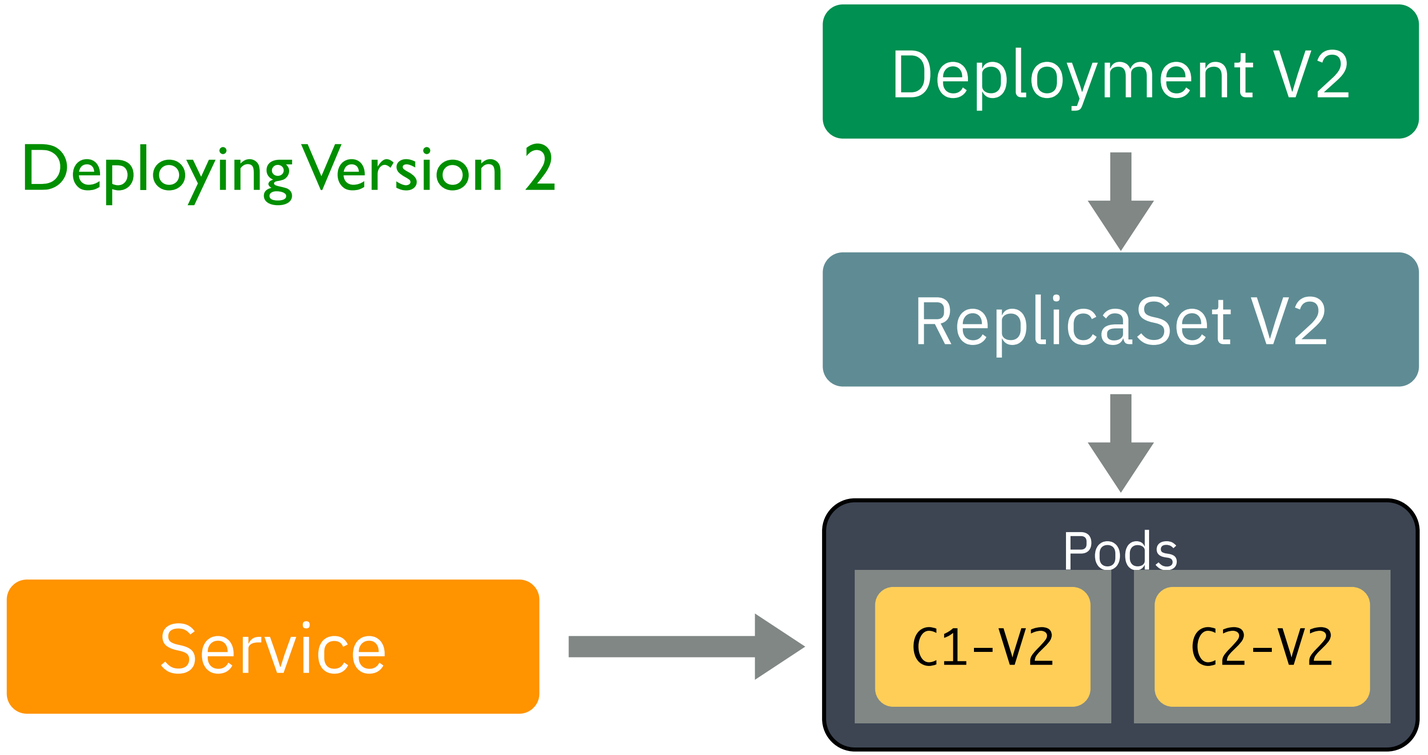
Kubernetes Rolling Updates (Zero Downtime Deployments)

```
$ kubectl set image deployment/<application-name> <container-name>=<container-image-name>:<new-version>
```

Kubernetes Rolling Updates (Zero Downtime Deployments)

```
$ kubectl set image deployment/<application-name> <container-name>=<container-image-name>:<new-version>
```

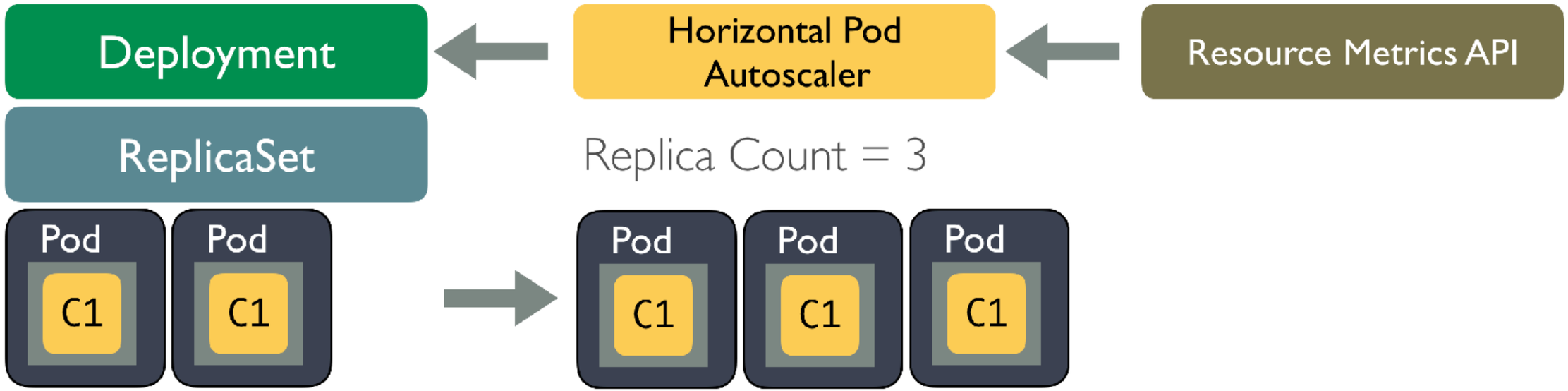
Deploying Version 2



Kubernetes Autoscaling

Kubernetes allows pods to be manually scaled either using ReplicaSets or Deployments. This can be achieved using the following CLI command:

```
$ kubectl scale --replicas=<desired-instance-count> deployment/<application-name>
```



Hands On

"live session"

Configure your Kubernetes Cluster

- We need to set permissions on the metadata that minikube needs:

```
$ sudo chown -R $USER $HOME/.kube $HOME/.minikube

$ sudo minikube addons enable ingress
```

Confirm that you cluster Works

- Make sure that your cluster is configured correctly

```
$ kubectl get all


NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   3m44s
```

Deploy Nginx

- Create a Deployment

```
$ kubectl create deployment my-nginx --image=nginx:alpine
deployment.apps/my-nginx created
```

- Check that the deployment and pods were created

```
$ kubectl get deployments
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
my-nginx   1/1     1            1           3h42m

$ kubectl get pods
NAME                        READY   STATUS    RESTARTS   AGE
my-nginx-b49cfc867-q7mjn    1/1     Running   0          3h42m
```

Deploy Nginx

- Create a Service

```
$ kubectl expose deploy my-nginx --type=NodePort --port=80
service/my-nginx exposed
```

- Check that the service was created

```
$ kubectl get service
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
kubernetes   ClusterIP   172.21.0.1      <none>        443/TCP
317d
my-nginx     NodePort    172.21.12.226   <none>        80:30371/TCP
3h42m
```

Kubectl get all

```
vagrant@ubuntu-xenial:~$ kubectl get all
NAME                              READY   STATUS    RESTARTS   AGE
pod/my-nginx-b49cfc867-q7mjn      1/1     Running   0          3h33m

NAME                 TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
service/kubernetes   ClusterIP   172.21.0.1      <none>        443/TCP        317d
service/my-nginx     NodePort    172.21.12.226   <none>        80:30371/TCP   3h30m

NAME                              READY   UP-TO-DATE   AVAILABLE   AGE
deployment.extensions/my-nginx    1/1     1            1           3h33m

NAME                                         DESIRED   CURRENT   READY   AGE
replicaset.extensions/my-nginx-b49cfc867     1         1         1       3h33m

NAME                          READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/my-nginx      1/1     1            1           3h33m

NAME                                    DESIRED   CURRENT   READY   AGE
replicaset.apps/my-nginx-b49cfc867      1         1         1       3h33m
```

Naming Convention

- We made a deployment called `my-nginx` and it created:

```
Deployment: my-nginx

ReplicaSet: my-nginx-b49cfc867

       Pod: my-nginx-b49cfc867-q7mjn
```

Naming Convention

- We made a deployment called `my-nginx` and it created:

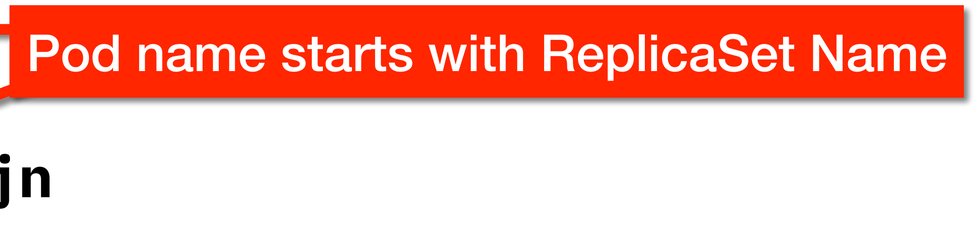
Naming Convention

- We made a deployment called `my-nginx` and it created:

Deployment: my-nginx

ReplicaSet: my-nginx-b49cfc867

Pod: my-nginx-b49cfc867-q7mjn



Get the NodePort

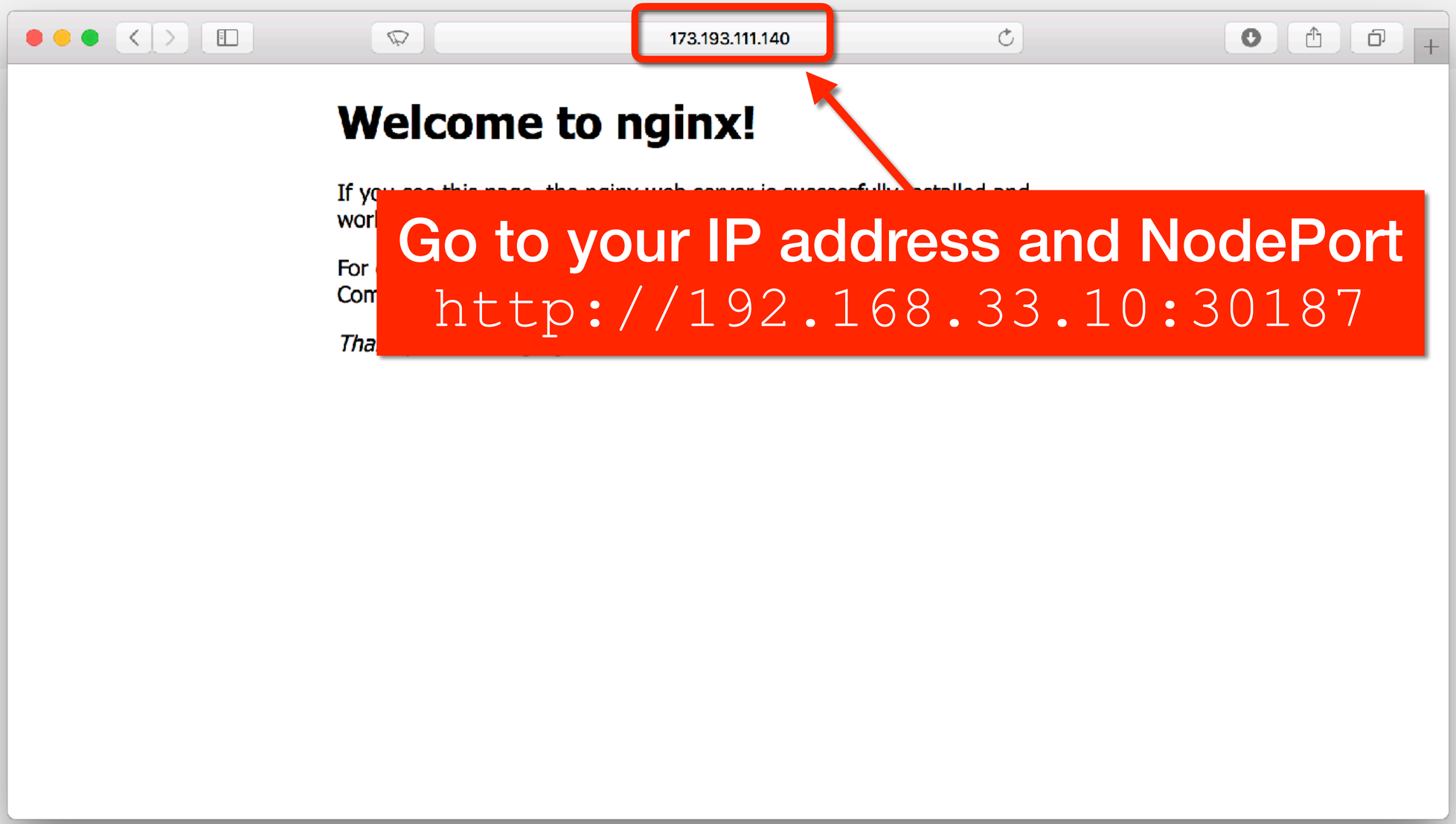
```
$ kubectl get service my-nginx
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
my-nginx      NodePort    172.21.9.54     <none>        80:30187/TCP   37d
```

Get the Public IP address

The IP address of the Vagrant Virtual Machine is in the Vagrantfile

192.168.33.10

Access NodePort



Create your Credentials Secret

- Use base64 to encode your secret and add it to the `secret.yaml` file:

```
$ echo "postgres://admin:s3cr3t@postgres:5432/postgres" | base64
cG9zdGdyZXM6Ly9hZG1pbjpzM2NyM3RAcG9zdGdyZXM6NTQzMi9wb3N0Z3Jlcwo=

$ vi secret.yaml

$ kubectl apply -f secret.yaml

$ kubectl apply -f postgres.yaml
```

Create A PostgreSQL Service*

- The pet demo application requires Postgres to store it's state
- We can deploy a Postgres container from the `postgres.yaml` file in the `./deploy` folder of the current repository

```
$ cd /vagrant/deploy

$ kubectl apply -f postgres.yaml
deployment.apps/postgres created
service/postgres created
```

* Note: in a "real" deployment we would attach a volume to the Redis container to persist it's data beyond the life of the container

postgres.yaml

Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: postgres
spec:
  type: NodePort
  selector:
    app: postgres
  ports:
    - name: primary
      protocol: TCP
      port: 5432
```

Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
      - name: postgres
        image: postgres:alpine
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 5432
          protocol: TCP
        env:
        - name: POSTGRES_USER
          value: "admin"
        - name: POSTGRES_PASSWORD
          value: "s3cr3t"
```

postgres.yaml

Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: postgres
spec:
  type: NodePort
  selector:
    app: postgres
  ports:
    - name: primary
      protocol: TCP
      port: 5432
```

Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
      - name: postgres
        image: postgres:alpine
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 5432
          protocol: TCP
        env:
        - name: POSTGRES_USER
          value: "admin"
        - name: POSTGRES_PASSWORD
          value: "s3cr3t"
```

Don't try this at home
...it's only a demo!

Now we can Deploy Our Image

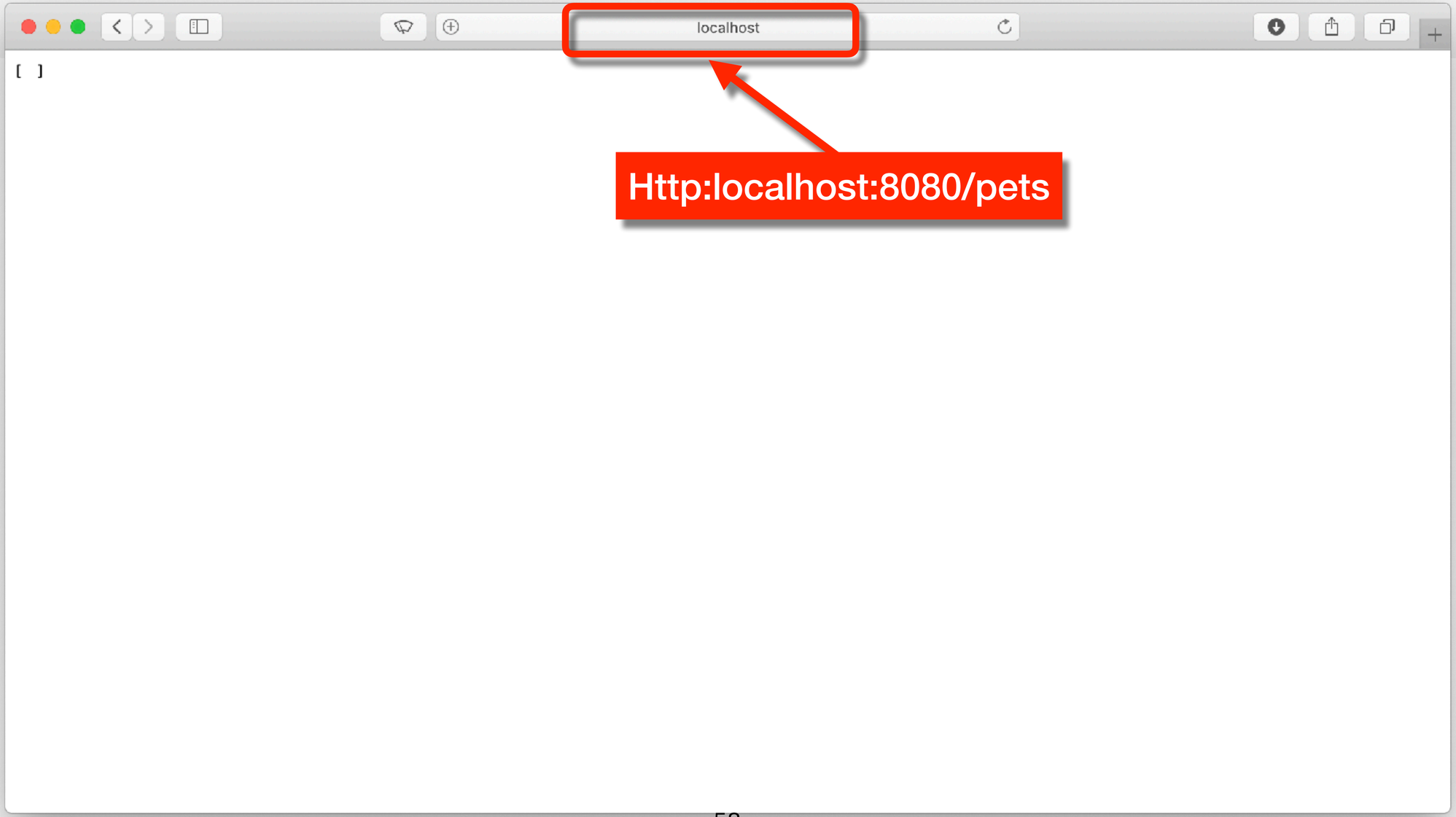
- You can use `kubectl apply` with the files under `/vagrant/deploy`

```
$ cd /vagrant/deploy

$ kubectl apply -f deployment.yaml
deployment.apps/pet-demo created

$ kubectl apply -f service.yaml
service/pet-demo-service created

$ kubectl apply -f ingress.yaml
ingress.extensions/pet-demo-ingress created
```

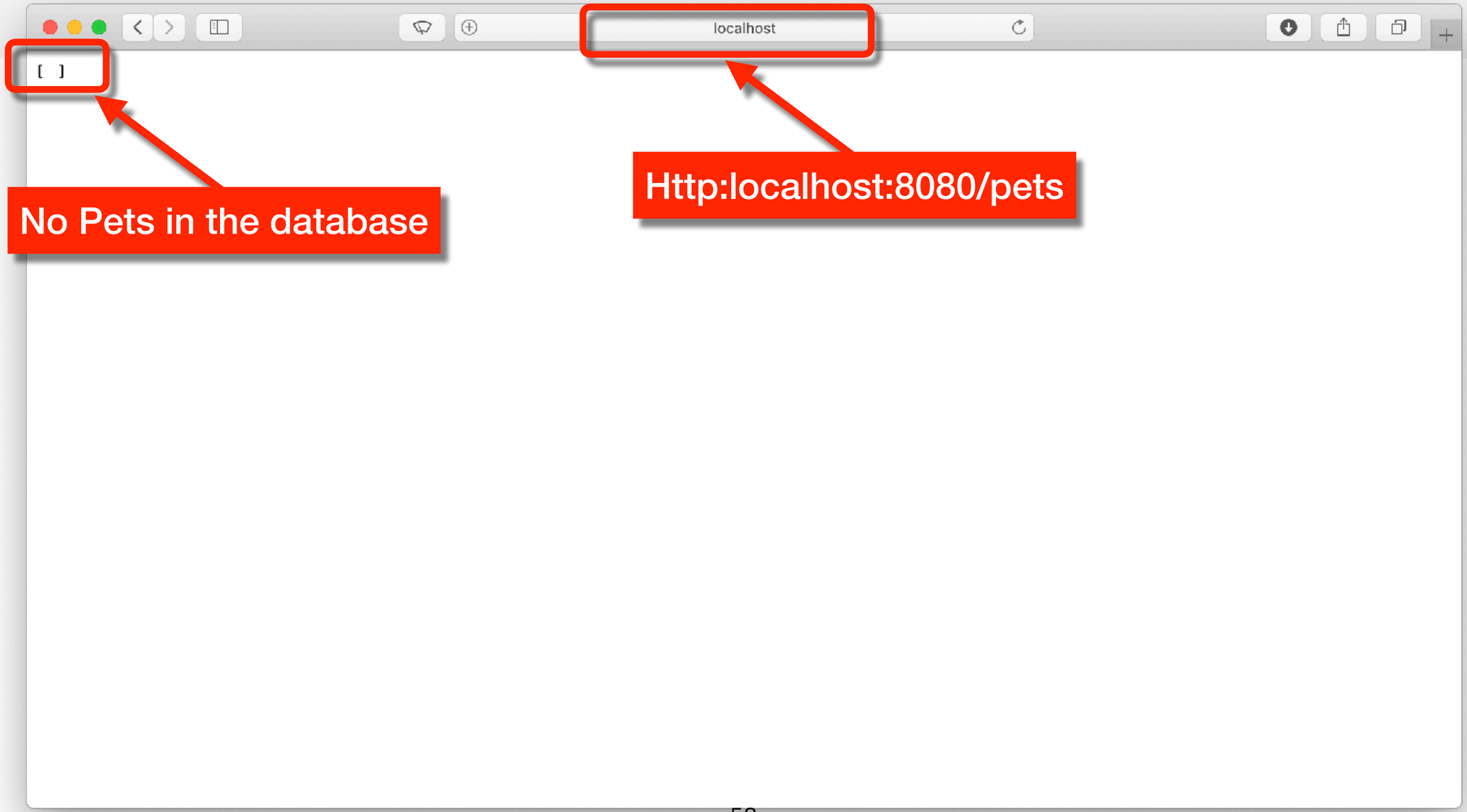
Check Kubernetes

```
$ kubectl get all
NAME                                 READY   STATUS    RESTARTS   AGE
pod/pet-demo-b8f9cc44d-7dq5w         1/1     Running   0          2m12s
pod/postgres-6565c9f5b8-bdbm4        1/1     Running   0          65m


NAME                       TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
service/kubernetes         ClusterIP   10.96.0.1        <none>        443/TCP          76m
service/pet-demo-service   NodePort    10.98.93.70      <none>        5000:31489/TCP   2m3s
service/postgres           NodePort    10.102.252.68    <none>        5432:31967/TCP   51m


NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/pet-demo        1/1     1            1           2m12s
deployment.apps/postgres        1/1     1            1           65m

NAME                                       DESIRED   CURRENT   READY   AGE
replicaset.apps/pet-demo-b8f9cc44d         1         1         1       2m12s
replicaset.apps/postgres-6565c9f5b8        1         1         1       65m
```

Your App on Kubernetes



Your App on Kubernetes



Use HTTP command to create some data

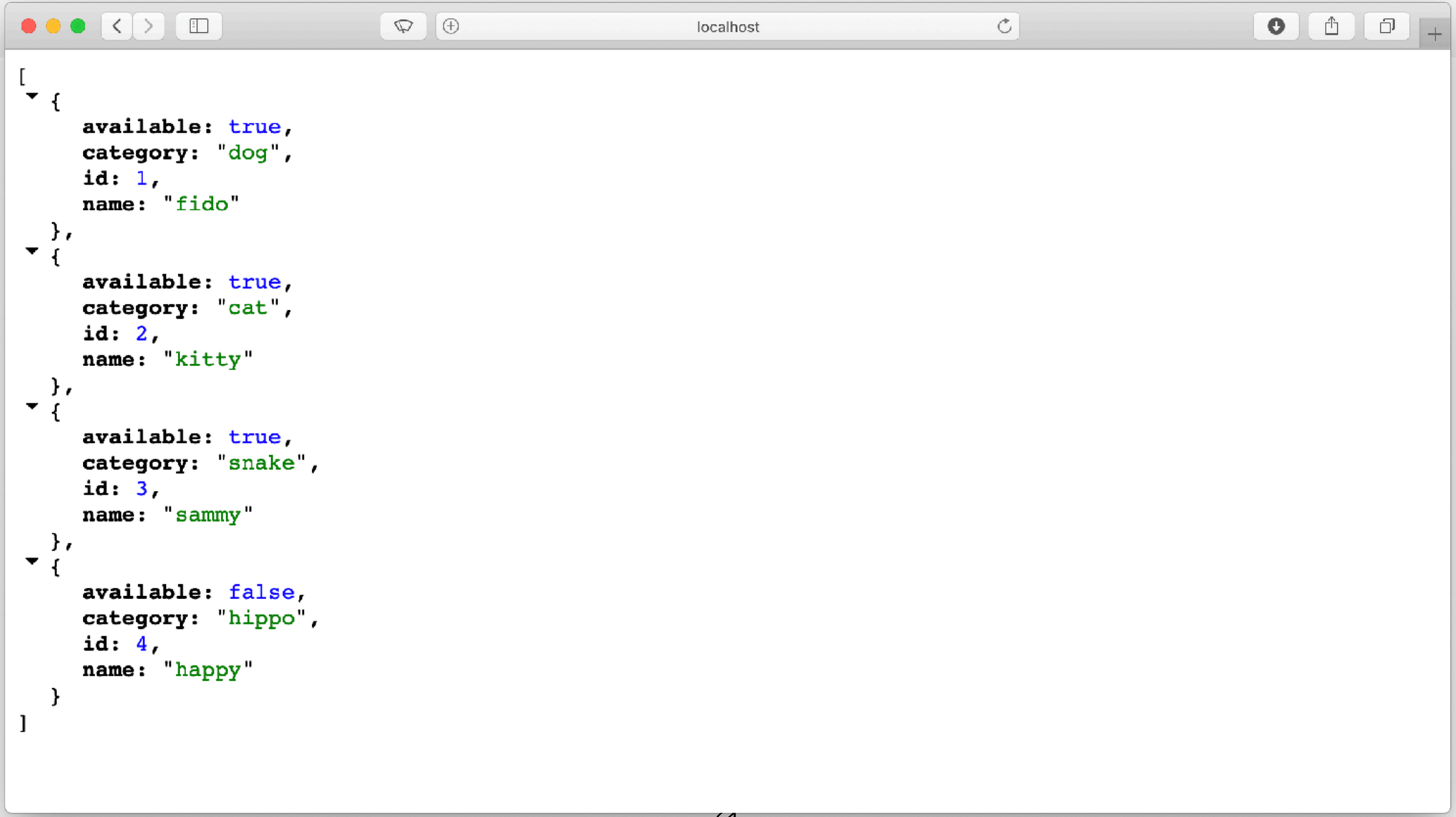
```
$ http post localhost/pets name='fido' category='dog' available:=true
HTTP/1.1 201 CREATED
Connection: keep-alive
Content-Length: 57
Content-Type: application/json
Date: Thu, 04 Jul 2019 21:37:46 GMT
Location: http://localhost/pets/3
Server: nginx/1.15.9

{
    "available": true,
    "category": "dog",
    "id": 3,
    "name": "fido"
}
```

Add these pets

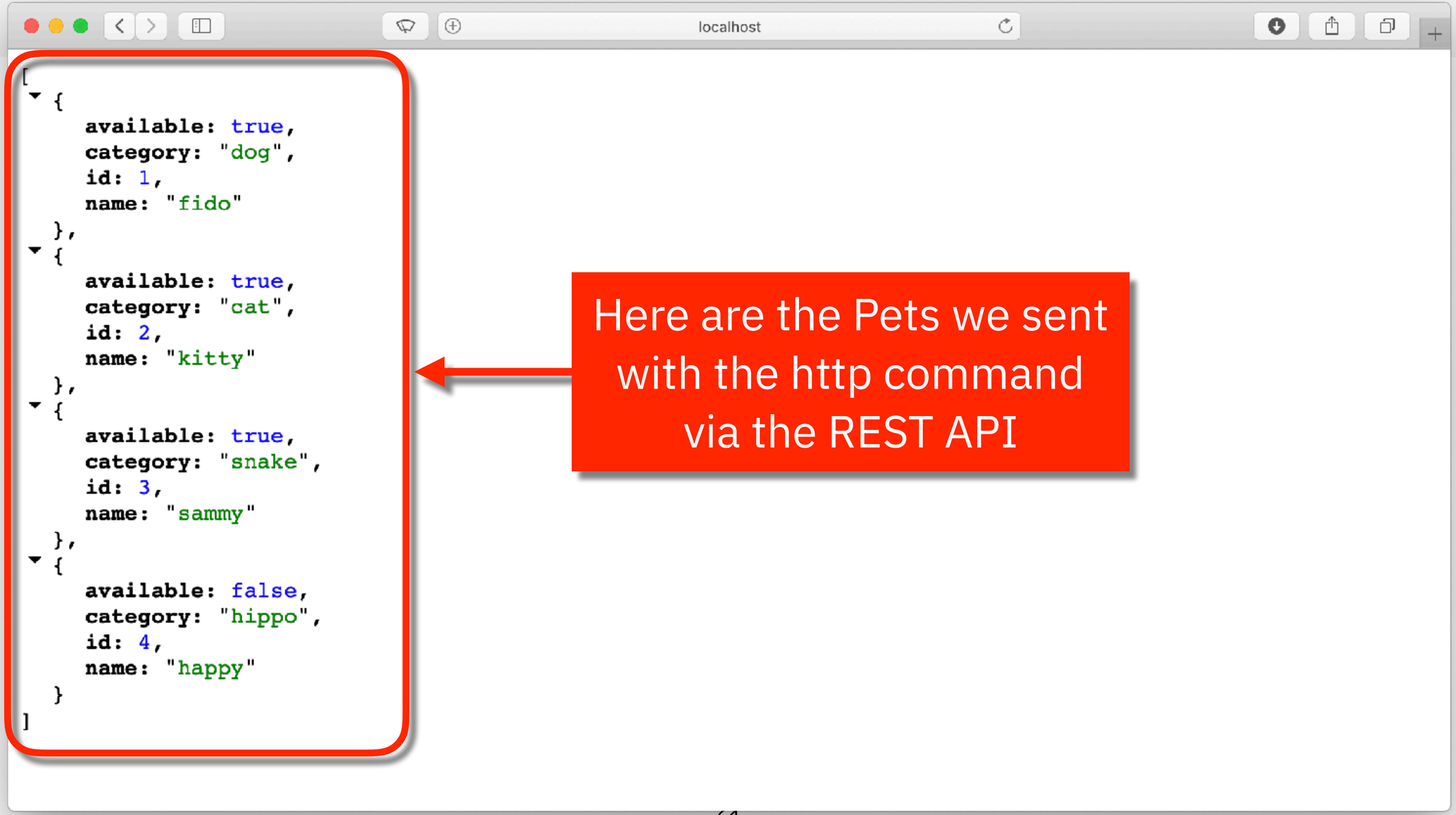
- Use http to these pets

```
$ http post localhost/pets name='fido' category='dog' available:=true

$ http post localhost/pets name='kitty' category='cat' available:=true

$ http post localhost/pets name='sammy' category='snake' available:=true

$ http post localhost/pets name='harry' category='hippo' available:=false
```

Refresh for browser



Refresh for browser



Summary

You just created your first Kubernetes deployment

You learned how to create deployments

You learned how to expose deployments as services

You can now deploy your microservices on any Kubernetes cloud