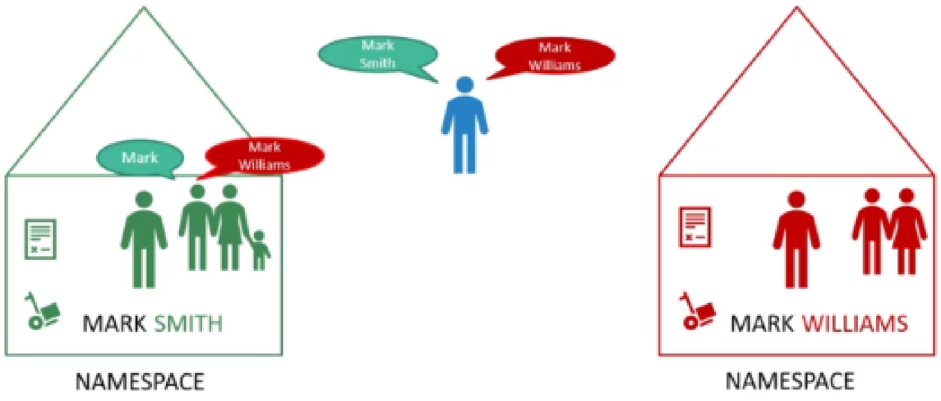
Namespaces

It is basically a environment

Set of rule and resources



These houses correspond to name spaces in Kubernetes.

Default namespace == it created auto byg k8s when we craete cluster

Set of pods and services for their initernal such as networking solution
== kube-system == created at cluster startup

It is isolated from user so that user can't delete

Kube-publice == it is public here resources made available to ll users

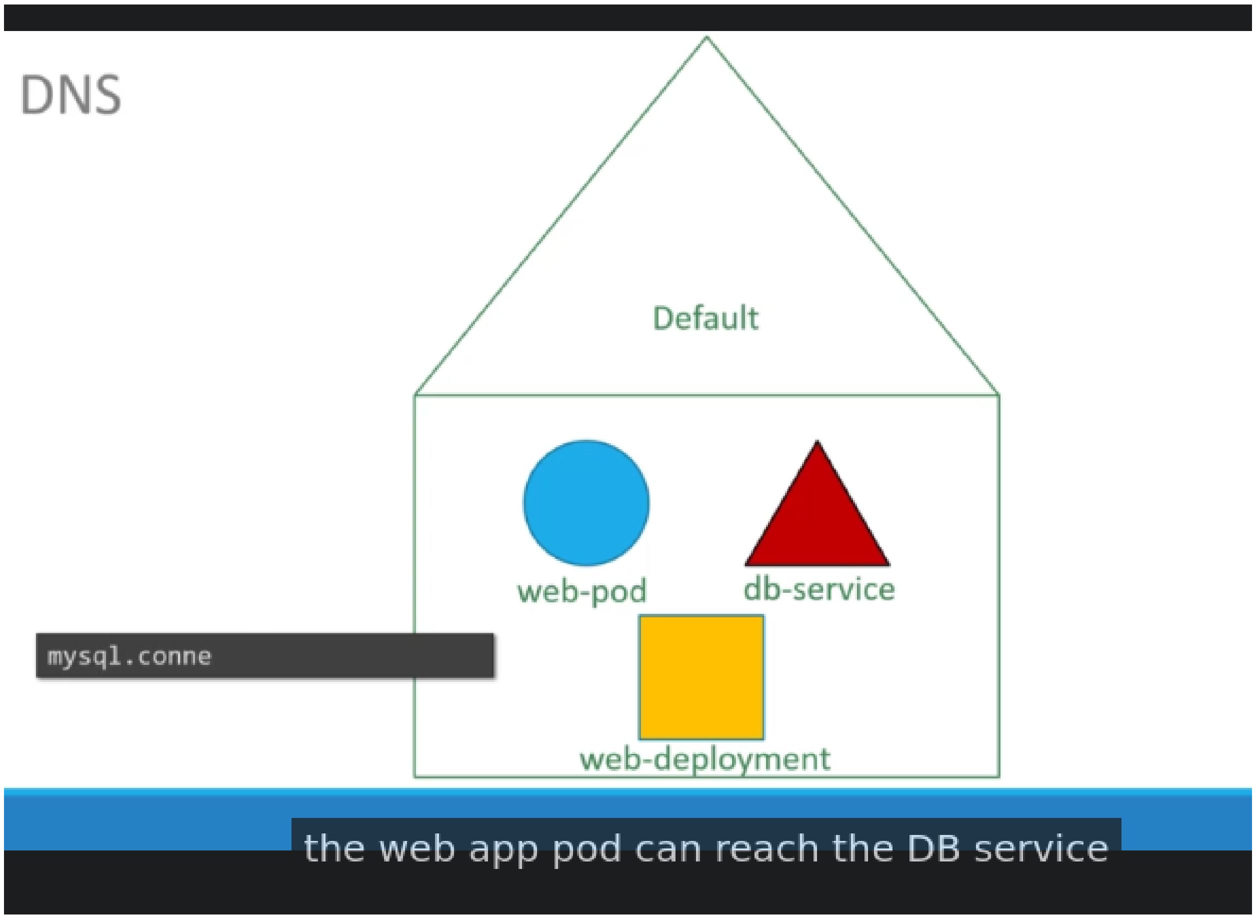
Environment small -== default

For enterprise == use your own created namespavce

Dev and env == set of policies define who can do what

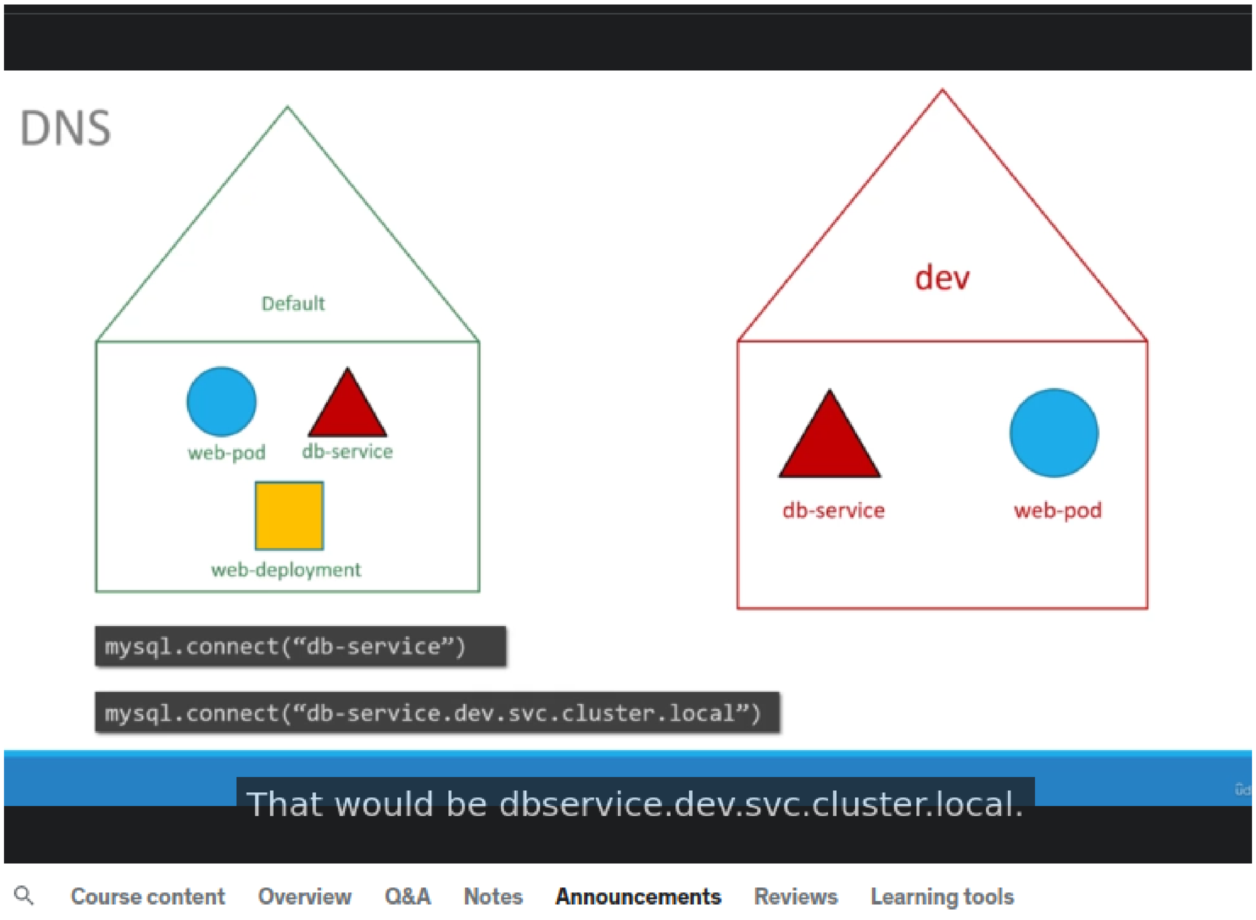
Resource quota to each namespaces

Resources within a namespace can refer by simply by their name



mysql.connect(db-serviuce)

For other namespace we used
Must append the name of namespace to the name of service
‘
mysql.connect(db-service.dev.svc.cluster.local)



```
mysql.connect("db-service")
mysql.connect("db-service.dev.svc.cluster.local")
```

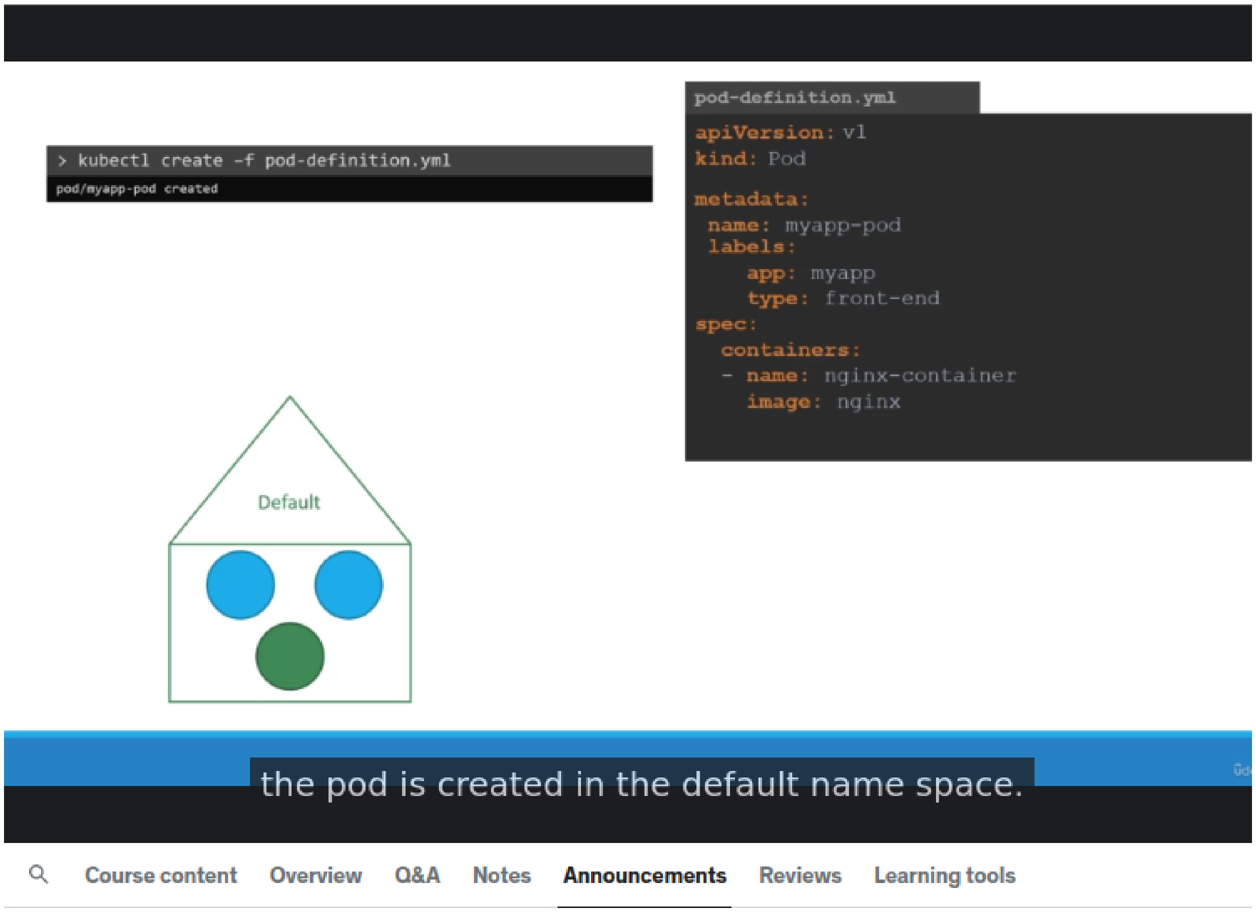
That would be dbservice.dev.svc.cluster.local.

When a service is created a dns name is added automatically

Cluster.local == default domain name of the k8s cluster

Svc is the name of of sub domain od service

Kubectl get pods == default
Kubectl get pods –namespace=kube-system or -n



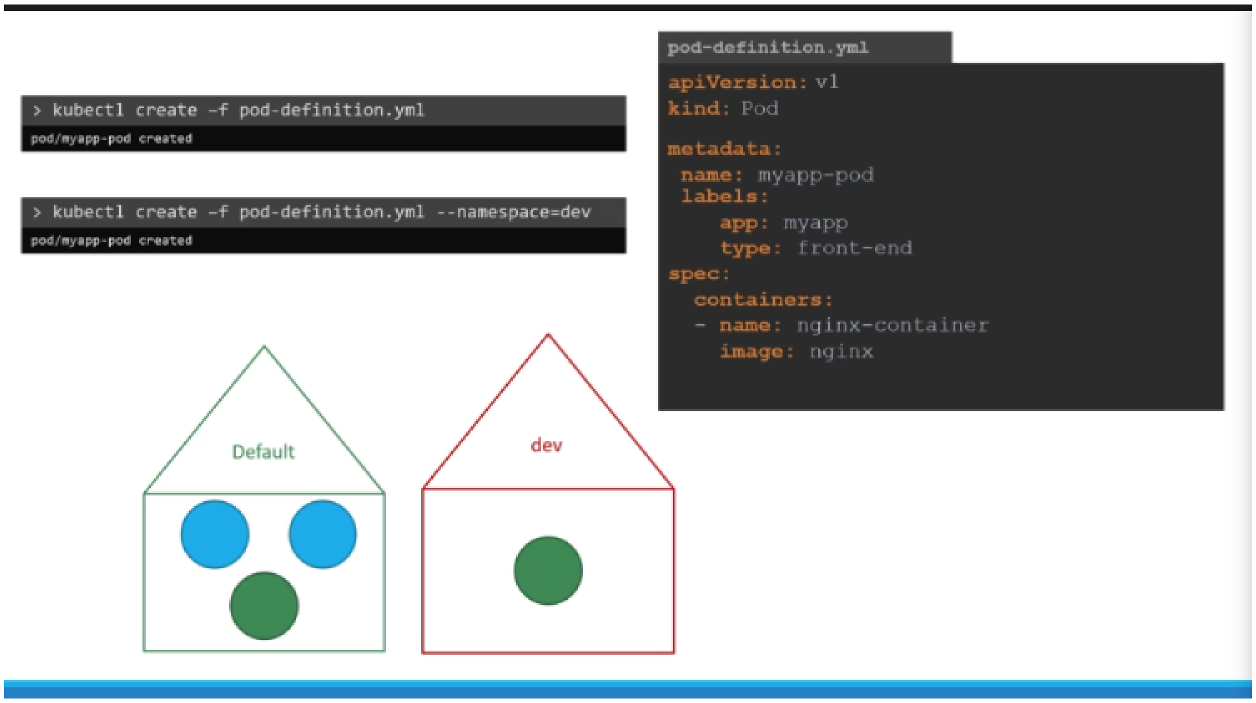
```
> kubectl create -f pod-definition.yml
pod/myapp-pod created
```

pod-definition.yml

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```

the pod is created in the default name space.

To create in other space
Use namespace option



```
> kubectl create -f pod-definition.yml
pod/myapp-pod created
```

```
> kubectl create -f pod-definition.yml --namespace=dev
pod/myapp-pod created
```

pod-definition.yml

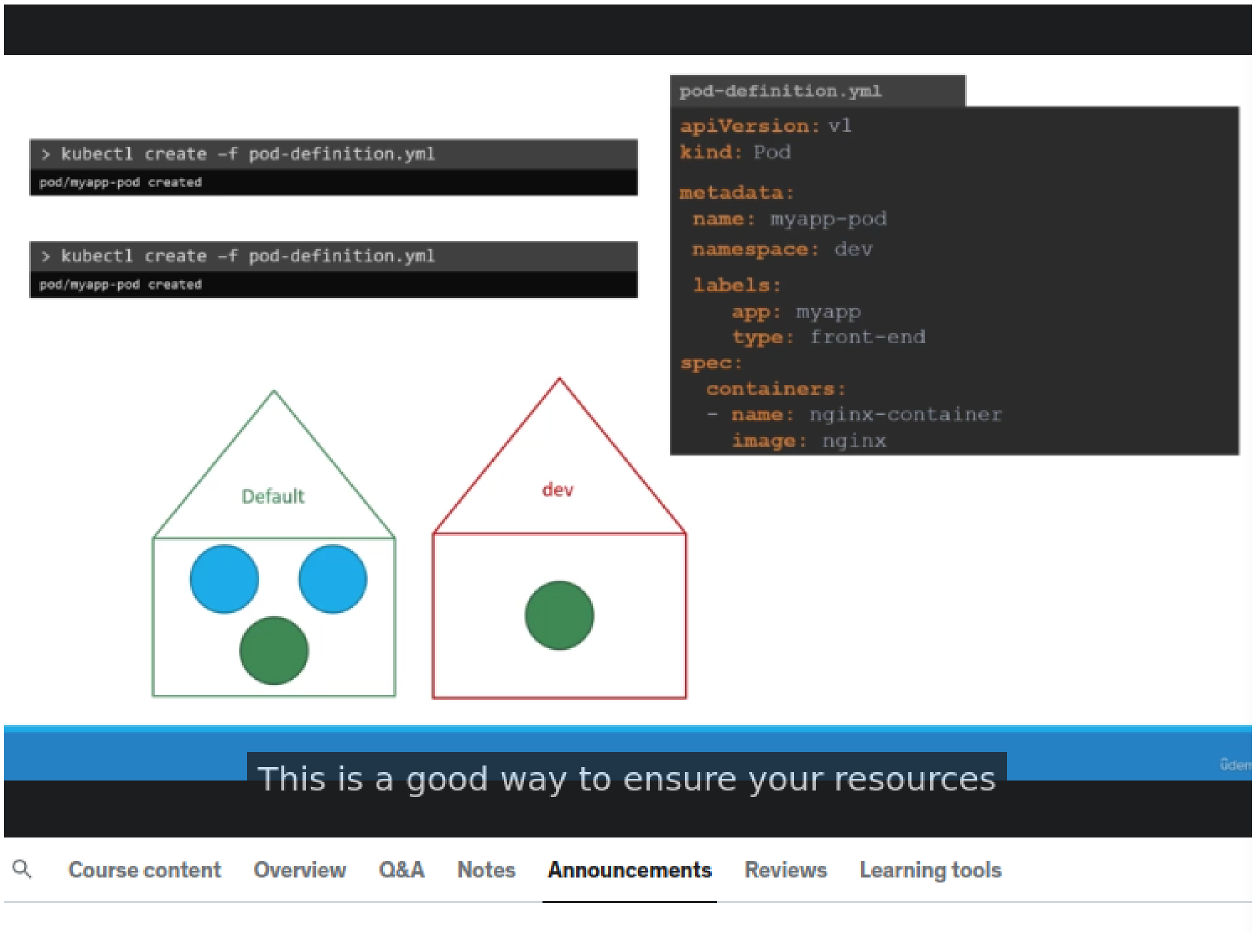
```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```

Metadata:

Namespace: dev or prod

And then create pod

Kubectl create -f yaml-file



```
> kubectl create -f pod-definition.yml
pod/myapp-pod created
```

```
> kubectl create -f pod-definition.yml
pod/myapp-pod created
```

pod-definition.yml

```
apiVersion: v1
kind: Pod

metadata:
  name: myapp-pod
  namespace: dev
  labels:
     app: myapp
     type: front-end
spec:
   containers:
   - name: nginx-container
     image: nginx
```

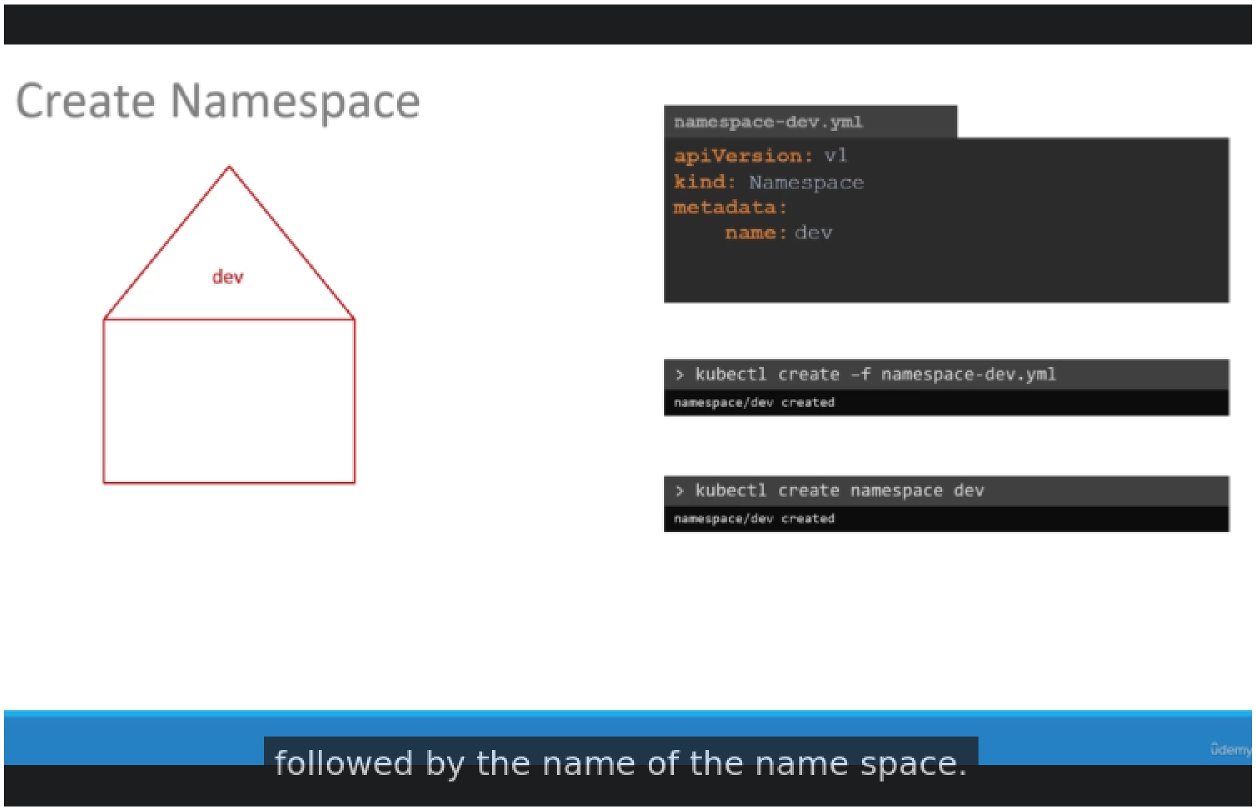
How to create namespace

apiVersion: v1
Kind: Namespace
Metadata:
        Name:dev

Kubectl create -f namespace-file.yaml

2 . 2ns way

Kubectl create namespace namespace-name


Create Namespace

```
namespace-dev.yml
apiVersion: v1
kind: Namespace
metadata:
    name: dev
```

```
> kubectl create -f namespace-dev.yml
namespace/dev created
```

```
> kubectl create namespace dev
namespace/dev created
```

followed by the name of the name space.

We have 3 ns dev , default , prod by default we are in default

How to switch

Kubectl get pods –namespace=dev
Kubectl get pods == output is default ns pod
Kubectl get pods –namespace=prod

But we dont want to menation ns always

Switch to another ns permanenlty

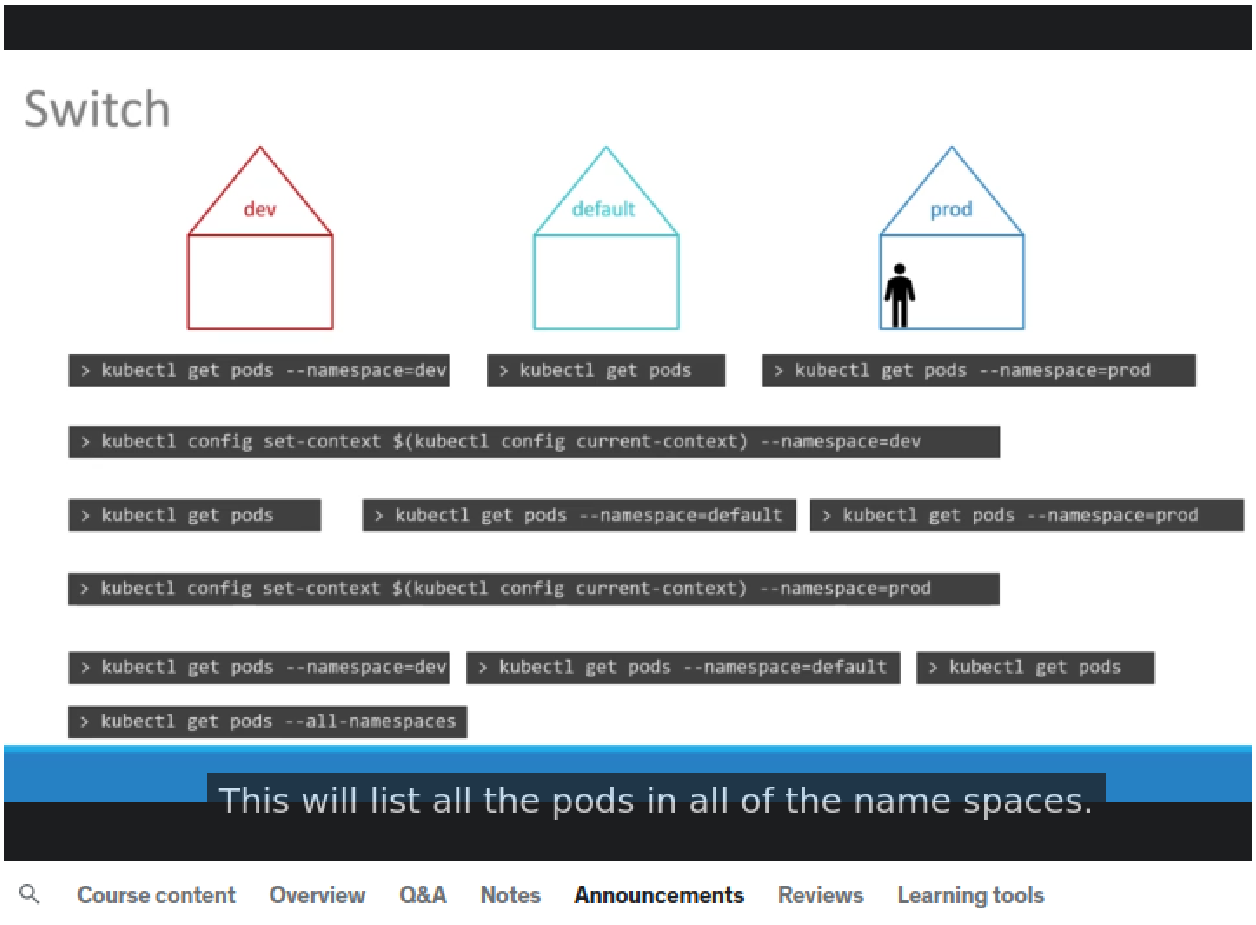
—---------
Kubectl config set-context $(kubectl config current-context) –namespace=namespace-name

And now only run

Kubectl get pods == output is only the dev environment

Pods in all namespaces

Kubectl get pods –all-namespaces
Kubectl get pods -A



```
> kubectl get pods --namespace=dev
> kubectl get pods
> kubectl get pods --namespace=prod
> kubectl config set-context $(kubectl config current-context) --namespace=dev
> kubectl get pods
> kubectl get pods --namespace=default
> kubectl get pods --namespace=prod
> kubectl config set-context $(kubectl config current-context) --namespace=prod
> kubectl get pods --namespace=dev
> kubectl get pods --namespace=default
> kubectl get pods
> kubectl get pods --all-namespaces
```

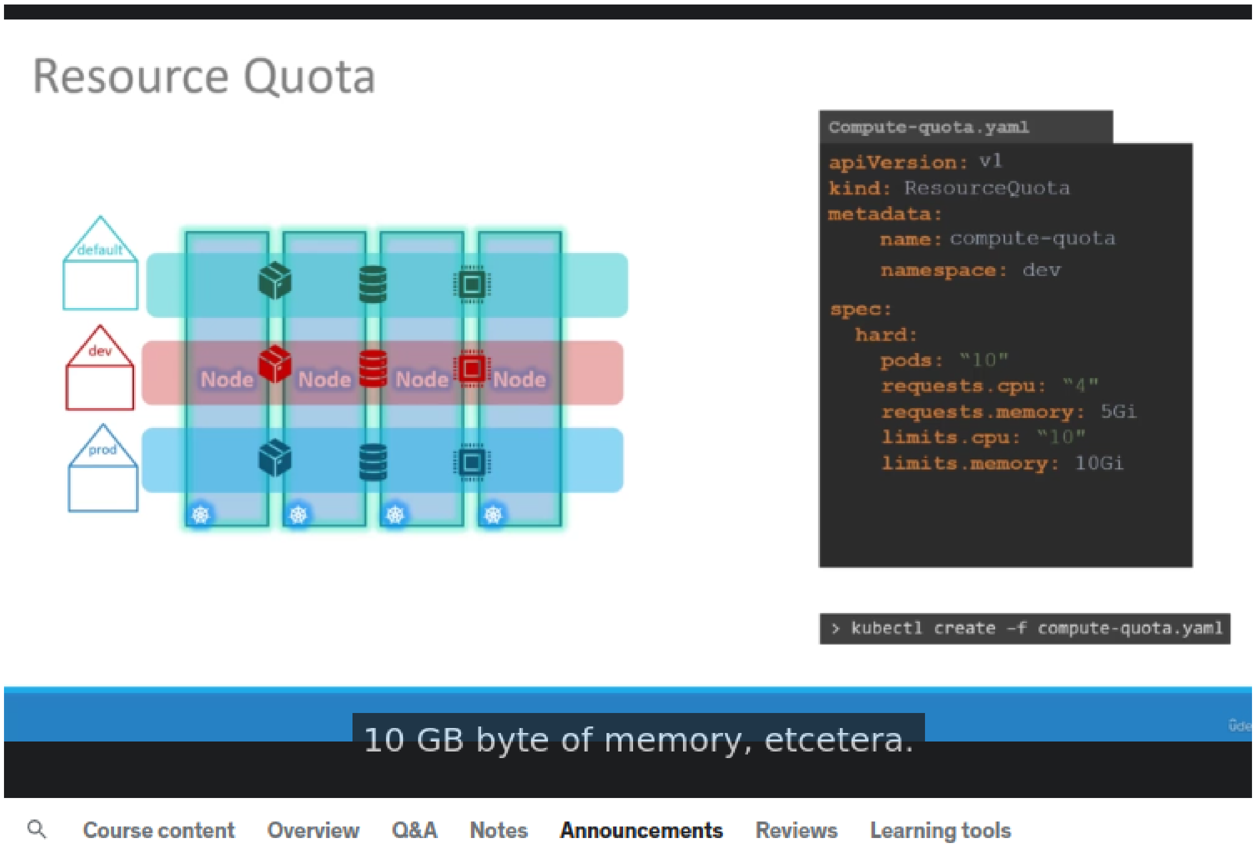
First find out current context and then add the naespace for that context

Limit resource

apiVersion: v1
Kind: ResourceQuota
Metadata:
    Name: compute-quota
    Namespace: dev

Spec: // provides your init
 Hard:
	Pods: “10”
	Requests.cpu: “4”
	requests.memory: 5Gi
limits.cpu: “10”
Limits.memory: 10Gi

Kubectl create -f resource-quota-file-name.yaml



Resource Quota

Compute-quota.yaml

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
    name: compute-quota
    namespace: dev

spec:
  hard:
    pods: "10"
    requests.cpu: "4"
    requests.memory: 5Gi
    limits.cpu: "10"
    limits.memory: 10Gi
```

```
> kubectl create -f compute-quota.yaml
```

10 GB byte of memory, etcetera.

Kubectl get ns or namespaces

Kubectl get ns or namespaces
Kubectl get pods – namespace=namespace-name
Kubectl get pods – n=namespace-name

Imperative and Declarative

Steps by step —-- final destination

How to do what to do —------ what to do

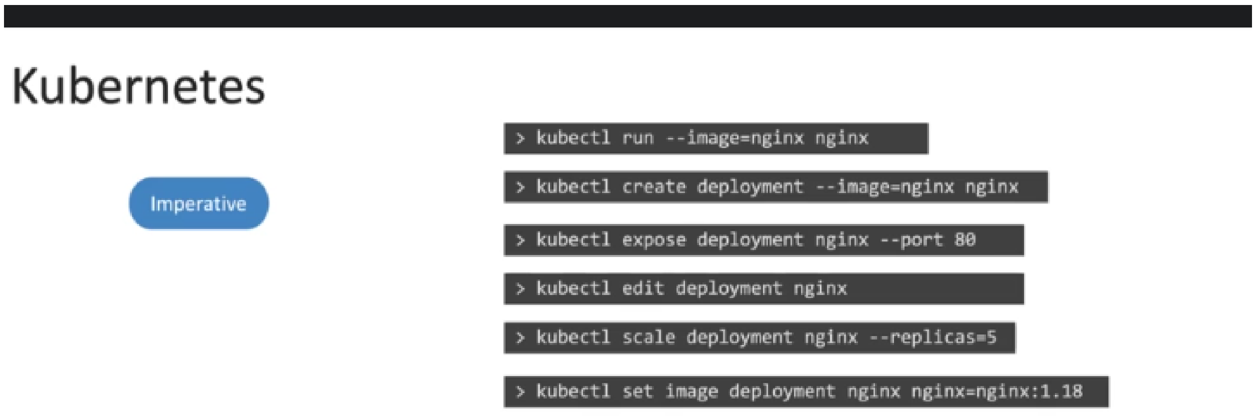
In iaac == a ser of instruction -== command ==Imperatibve

In iaac declare only requirements == Declarative === all things are done by software

There are 7 steps
In first run only 4 step execute
Then in next run
We need to provide checks that if this happen then dont apply

Im

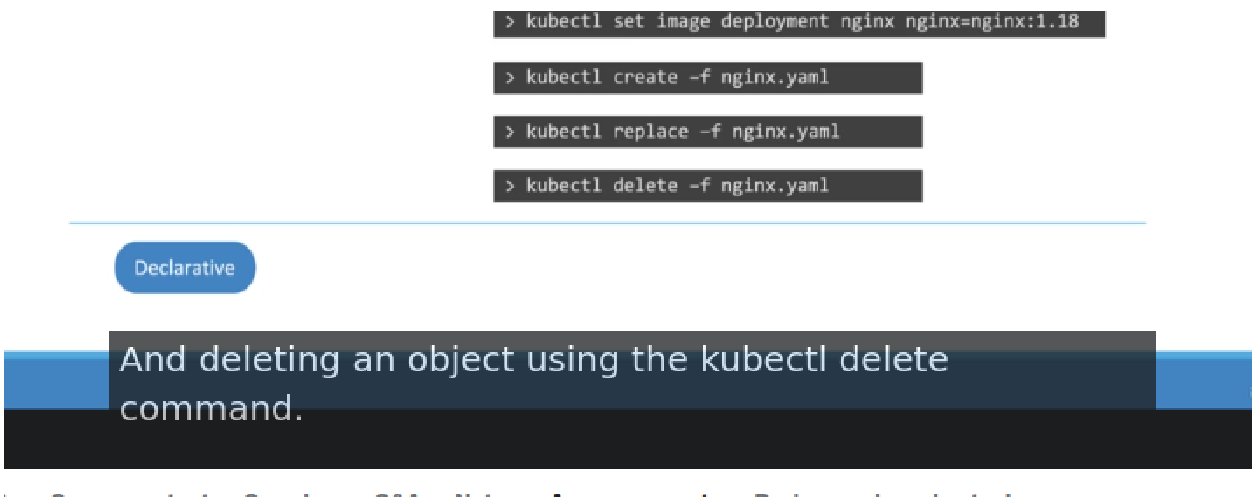
Imperative



```
> kubectl run --image=nginx nginx
> kubectl create deployment --image=nginx nginx
> kubectl expose deployment nginx --port 80
> kubectl edit deployment nginx
> kubectl scale deployment nginx --replicas=5
> kubectl set image deployment nginx nginx=nginx:1.18
```

We can also use file

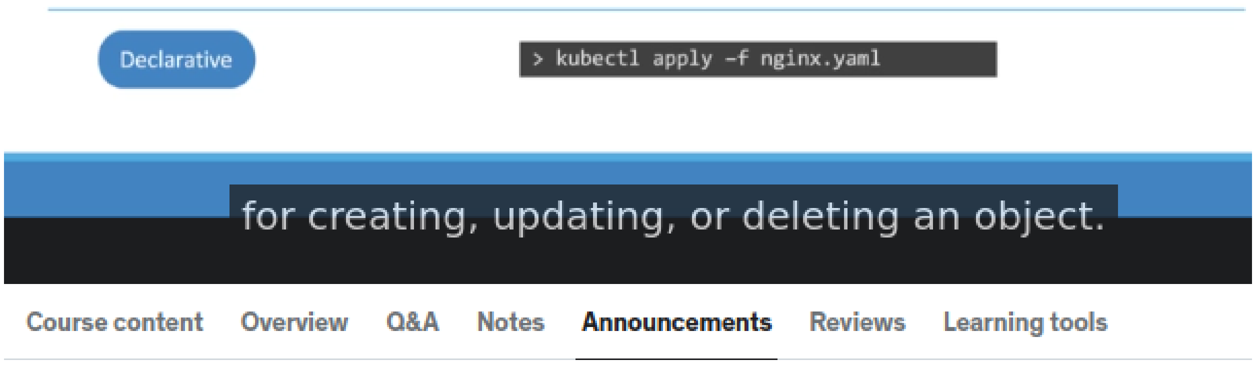
Create , replace delete



```
> kubectl set image deployment nginx nginx=nginx:1.18
> kubectl create -f nginx.yaml
> kubectl replace -f nginx.yaml
> kubectl delete -f nginx.yaml
```

And deleting an object using the kubectl delete command.

Declarative

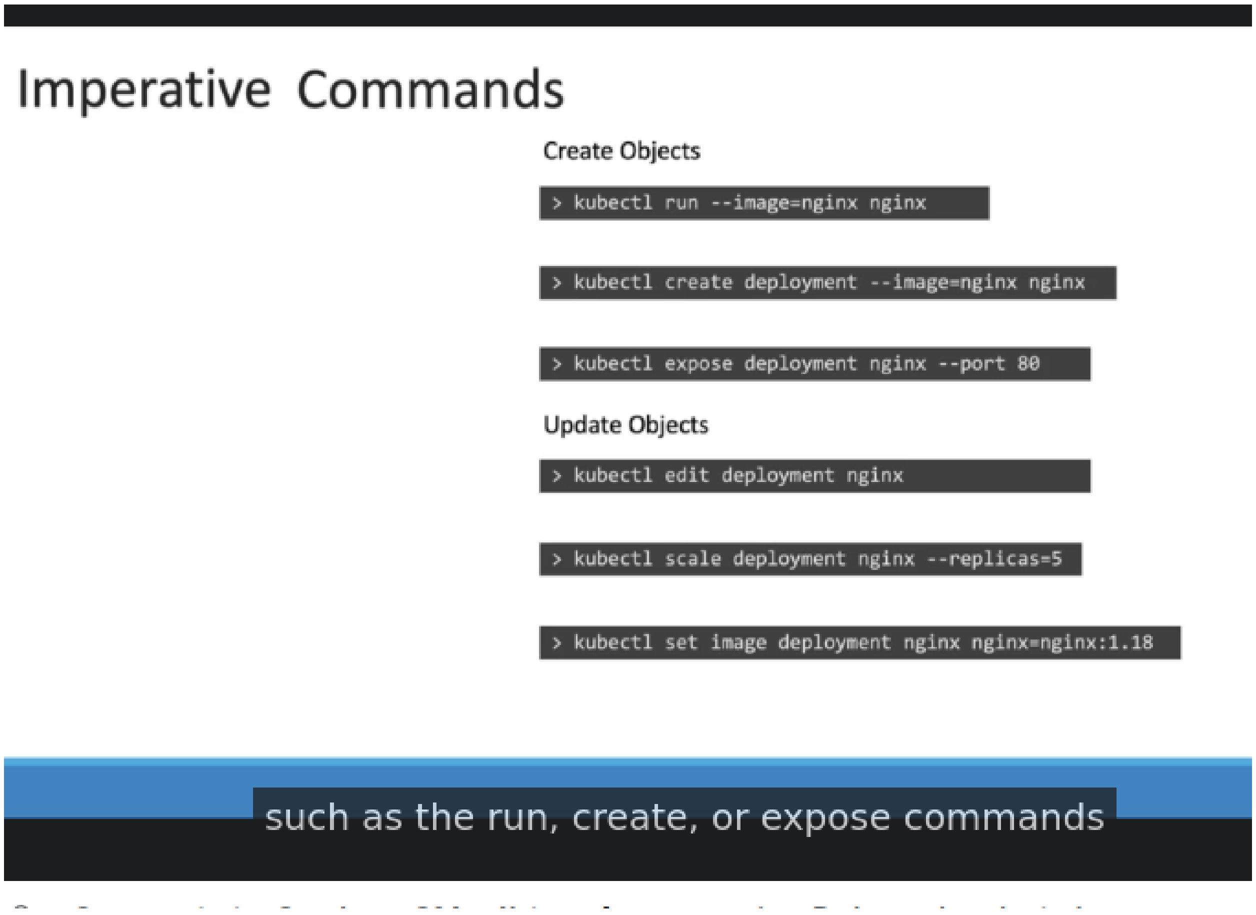


```
> kubectl apply -f nginx.yaml
```

for creating, updating, or deleting an object.

Apply command look existing configuration and figure out what changes need to be done to the system

Imperative commands



Imperative Commands

Create Objects

```
> kubectl run --image=nginx nginx
```

```
> kubectl create deployment --image=nginx nginx
```

```
> kubectl expose deployment nginx --port 80
```

Update Objects

```
> kubectl edit deployment nginx
```

```
> kubectl scale deployment nginx --replicas=5
```

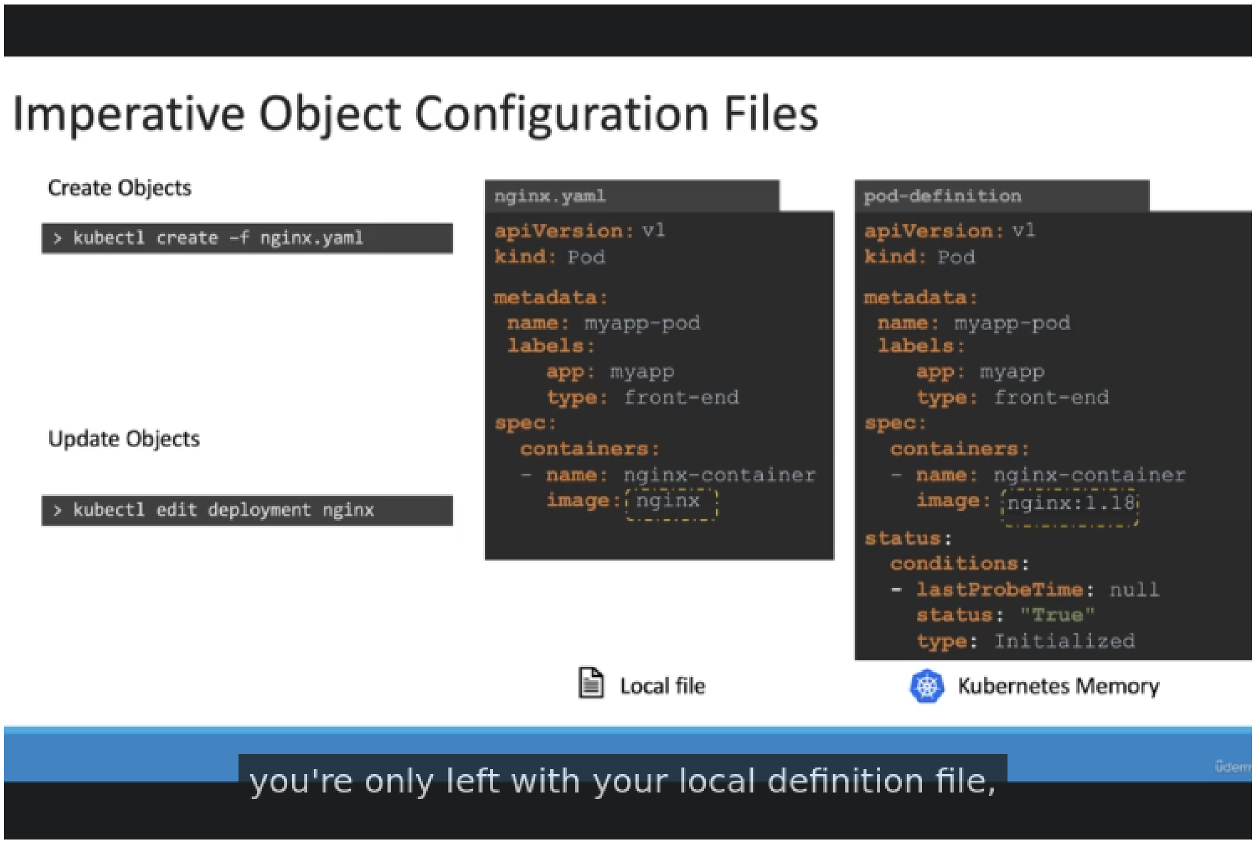
```
> kubectl set image deployment nginx nginx=nginx:1.18
```

such as the run, create, or expose commands

Also edit scale set commands to update existinf object

Run once and availableony in session history

Yaml file



Imperative Object Configuration Files

Create Objects

```
> kubectl create -f nginx.yaml
```

Update Objects

```
> kubectl edit deployment nginx
```

nginx.yaml

```
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```

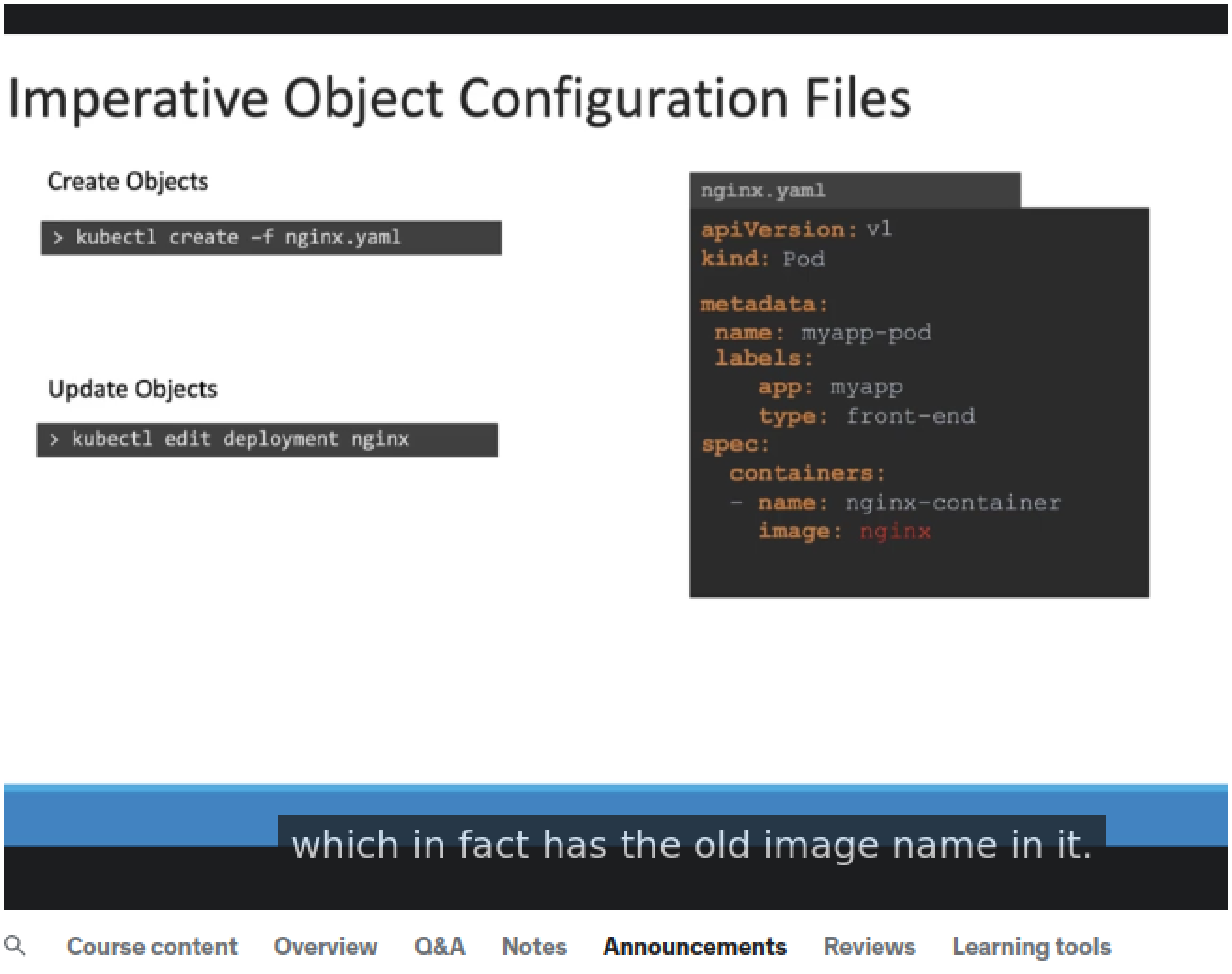
Local file

pod-definition

```
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx:1.18
status:
  conditions:
  - lastProbeTime: null
    status: "True"
    type: Initialized
```

Kubernetes Memory

you're only left with your local definition file,



Imperative Object Configuration Files

Create Objects

```
> kubectl create -f nginx.yaml
```

Update Objects

```
> kubectl edit deployment nginx
```

nginx.yaml

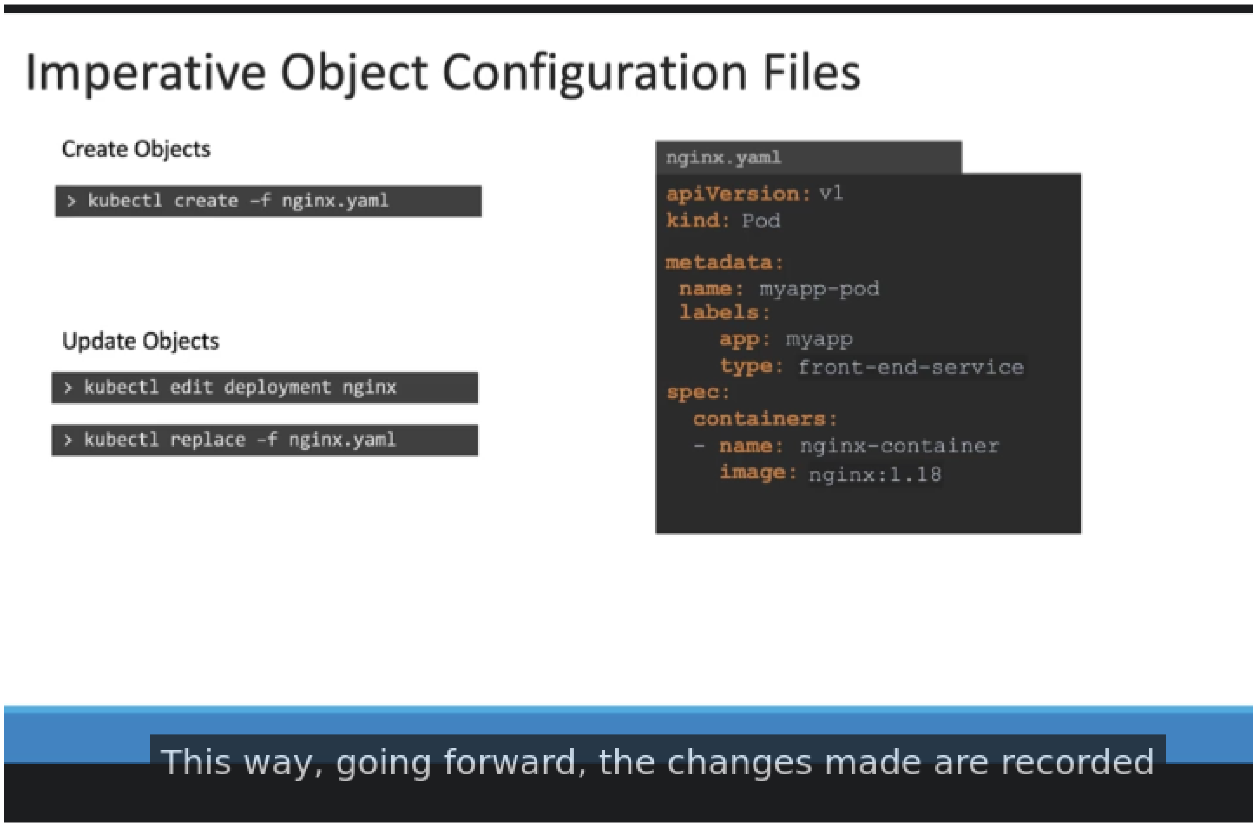
```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```

which in fact has the old image name in it.

But this will not change local file

So to change in local diectl change and run following command

Kubectl replace -f yaml-file



Imperative Object Configuration Files

Create Objects

```
> kubectl create -f nginx.yaml
```

Update Objects

```
> kubectl edit deployment nginx
```

```
> kubectl replace -f nginx.yaml
```

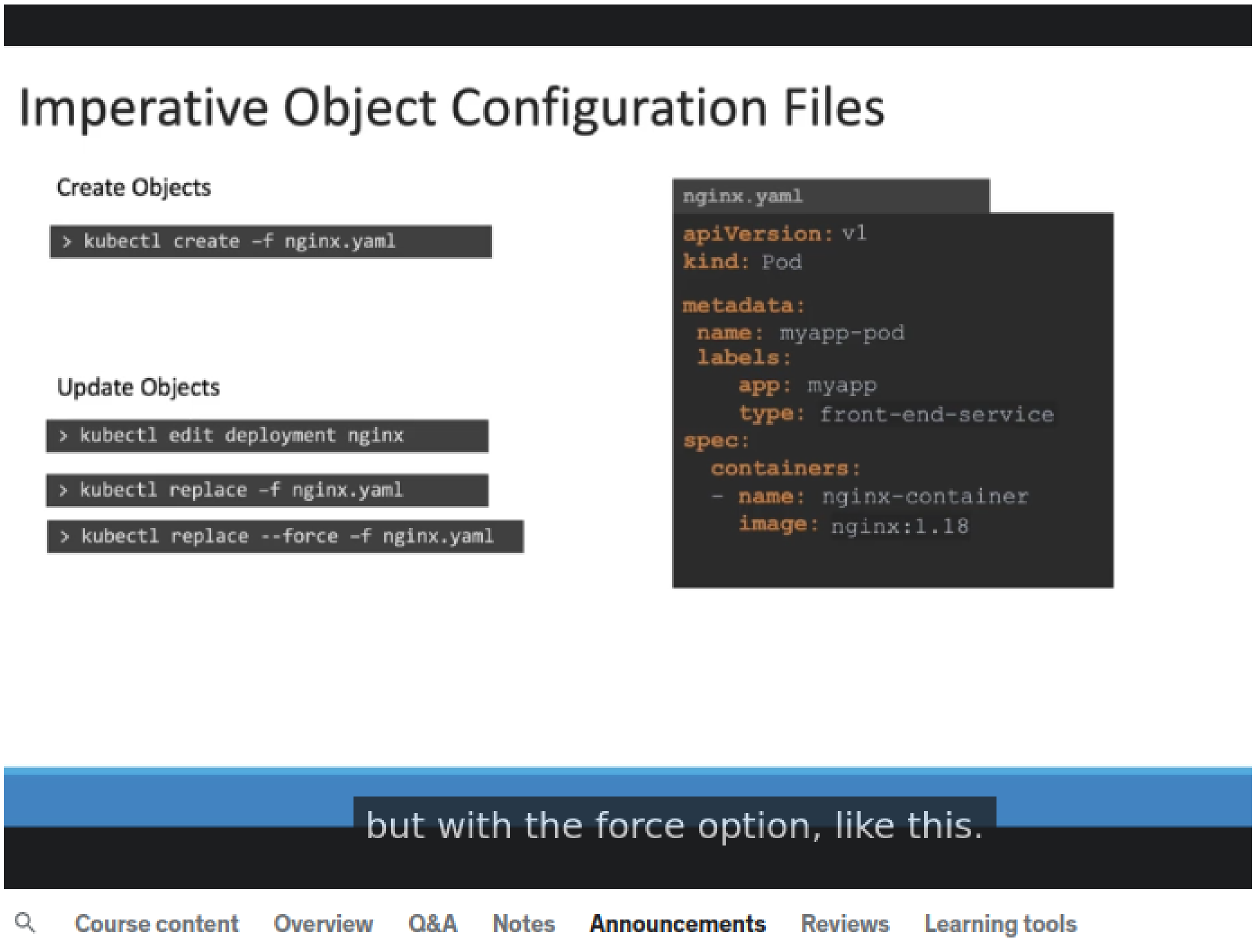
nginx.yaml

```
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end-service
spec:
  containers:
  - name: nginx-container
    image: nginx:1.18
```

This way, going forward, the changes made are recorded

Completely delete and recreate objects

Kubectl replace –force -f yaml-file



Imperative Object Configuration Files

Create Objects

```
> kubectl create -f nginx.yaml
```

Update Objects

```
> kubectl edit deployment nginx
```

```
> kubectl replace -f nginx.yaml
```

```
> kubectl replace --force -f nginx.yaml
```

nginx.yaml

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end-service
spec:
  containers:
  - name: nginx-container
    image: nginx:1.18
```

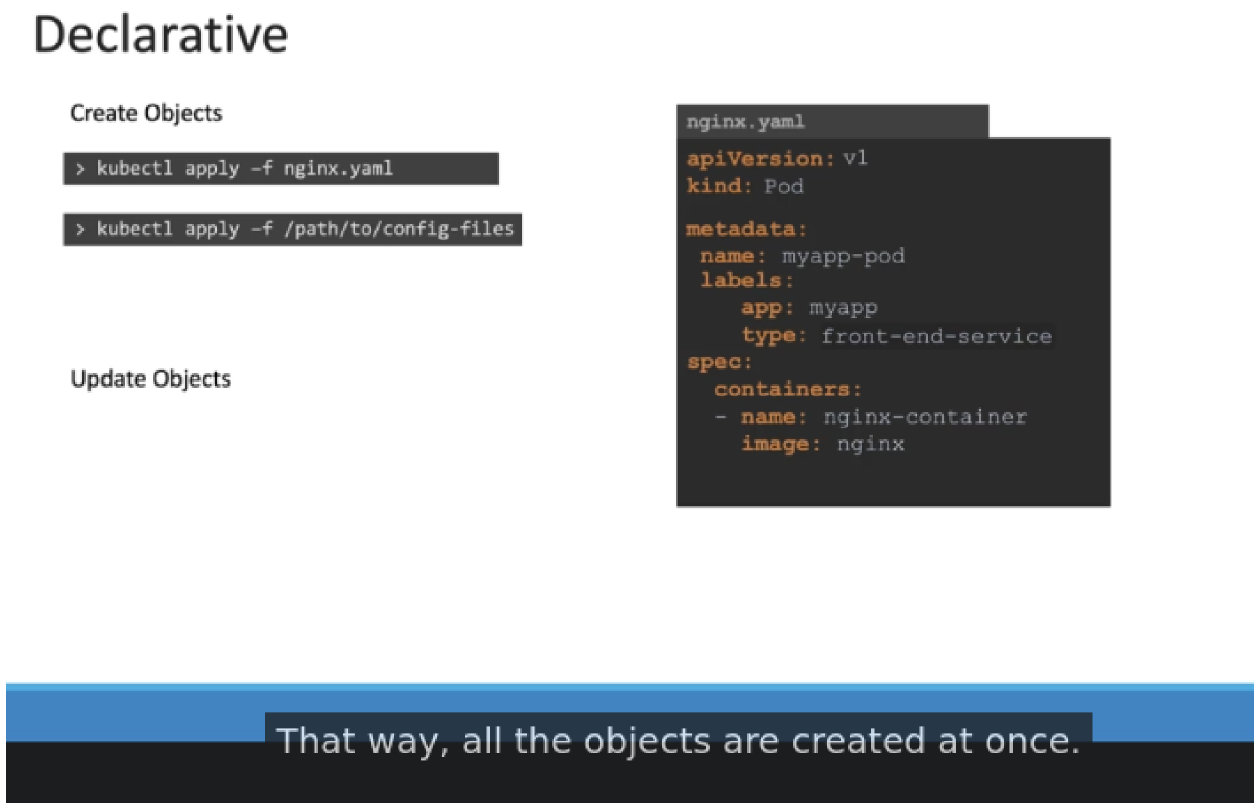
but with the force option, like this.

In imperative approach

If we create object and it already exists it gives error
When we run replace command object must be available

So to overcome all this problem we use declarative approach

Declarative



Declarative

Create Objects

```
> kubectl apply -f nginx.yaml
```

```
> kubectl apply -f /path/to/config-files
```

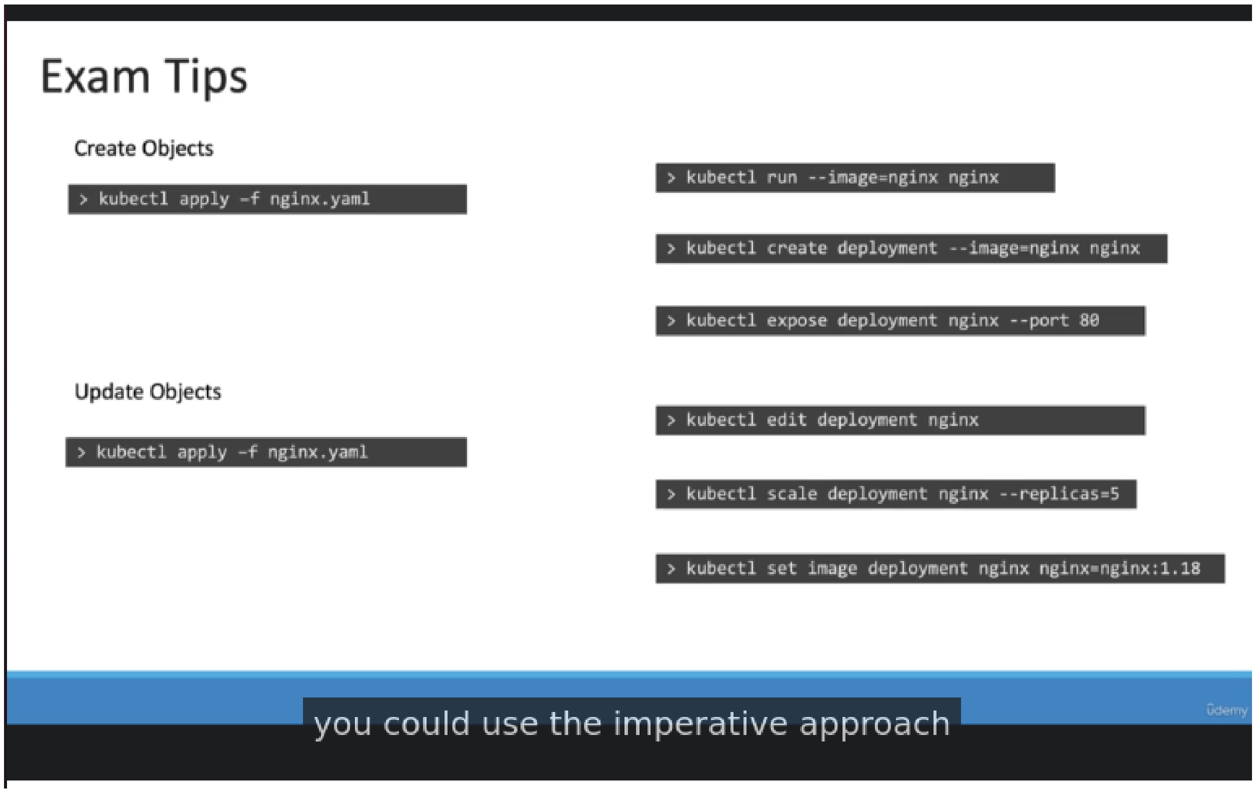
Update Objects

nginx.yaml

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end-service
spec:
  containers:
  - name: nginx-container
    image: nginx
```

That way, all the objects are created at once.

And now when changes need to be n=made we need to change only local file and run apply command again
Apply command knows that object exits then he only update the objects with new changes



Exam Tips

Create Objects

```
> kubectl apply -f nginx.yaml
```

```
> kubectl run --image=nginx nginx
```

```
> kubectl create deployment --image=nginx nginx
```

```
> kubectl expose deployment nginx --port 80
```

Update Objects

```
> kubectl apply -f nginx.yaml
```

```
> kubectl edit deployment nginx
```

```
> kubectl scale deployment nginx --replicas=5
```

```
> kubectl set image deployment nginx nginx=nginx:1.18
```

you could use the imperative approach

Certification Tips - Imperative Commands with Kubectl

While you would be working mostly the declarative way - using definition files, imperative commands can help in getting one time tasks done quickly, as well as generate a definition template easily. This would help save considerable amount of time during your exams.

Before we begin, familiarize with the two options that can come in handy while working with the below commands:

--dry-run: By default as soon as the command is run, the resource will be created. If you simply want to test your command , use the --dry-run=client option. This will not create the resource, instead, tell you whether the resource can be created and if your command is right.

-o yaml: This will output the resource definition in YAML format on screen.

Use the above two in combination to generate a resource definition file quickly, that you can then modify and create resources as required, instead of creating the files from scratch.

POD

Create an NGINX Pod

```
kubectl run nginx --image=nginx
```

Generate POD Manifest YAML file (-o yaml). Don't create it(--dry-run)

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

Deployment

Create a deployment

```
kubectl create deployment --image=nginx nginx
```

Generate Deployment YAML file (-o yaml). Don't create it(--dry-run)

```
kubectl create deployment --image=nginx nginx --dry-run=client -o yaml
```

Generate Deployment with 4 Replicas

```
kubectl create deployment nginx --image=nginx --replicas=4
```

You can also scale a deployment using the kubectl scale command.

```
kubectl scale deployment nginx --replicas=4
```

Another way to do this is to save the YAML definition to a file and modify

```
kubectl create deployment nginx --image=nginx --dry-run=client -o yaml >
nginx-deployment.yaml
```

You can then update the YAML file with the replicas or any other field before creating the deployment.

Service

Create a Service named redis-service of type ClusterIP to expose pod redis on port 6379

kubectl expose pod redis --port=6379 --name redis-service --dry-run=client -o yaml

(This will automatically use the pod's labels as selectors)

Or

kubectl create service clusterip redis --tcp=6379:6379 --dry-run=client -o yaml (This will not use the pods labels as selectors, instead it will assume selectors as **app=redis.** You cannot pass in selectors as an option. So it does not work very well if your pod has a different label set. So generate the file and modify the selectors before creating the service)

Create a Service named nginx of type NodePort to expose pod nginx's port 80 on port 30080 on the nodes:

kubectl expose pod nginx --type=NodePort --port=80 --name=nginx-service --dry-run=client -o yaml

(This will automatically use the pod's labels as selectors, but you cannot specify the node port. You have to generate a definition file and then add the node port in manually before creating the service with the pod.)

Or

kubectl create service nodeport nginx --tcp=80:80 --node-port=30080 --dry-run=client -o yaml

(This will not use the pods labels as selectors)

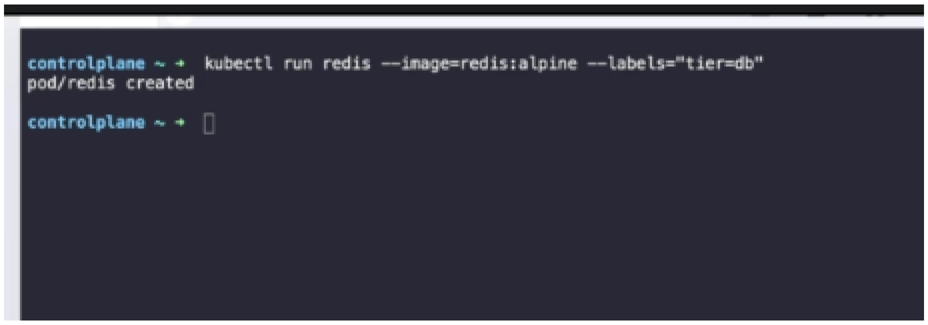
Both the above commands have their own challenges. While one of it cannot accept a selector the other cannot accept a node port. I would recommend going with the kubectl expose command. If you need to specify a node port, generate a definition file using the same command and manually input the nodeport before creating the service.

Reference:

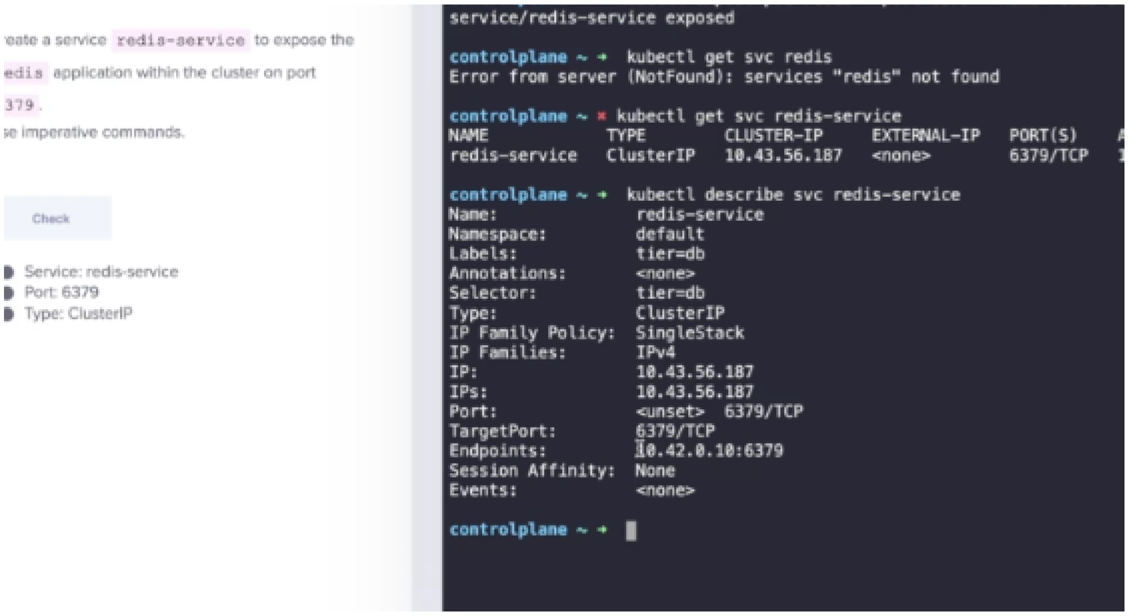
https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands

https://kubernetes.io/docs/reference/kubectl/conventions/

–labelsl=”app=run”



```
controlplane ~ ➜  kubectl run redis --image=redis:alpine --labels="tier=db"
pod/redis created

controlplane ~ ➜  
```

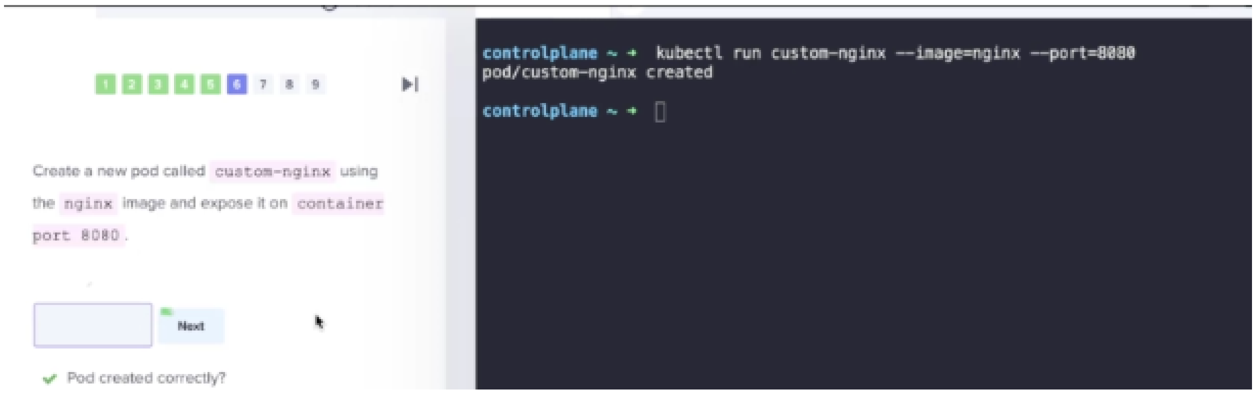


eate a service `redis-service` to expose the `edis` application within the cluster on port `379`.

e imperative commands.

- Service: redis-service
- Port: 6379
- Type: ClusterIP

```
service/redis-service exposed

controlplane ~ ➜  kubectl get svc redis
Error from server (NotFound): services "redis" not found

controlplane ~ ✖ kubectl get svc redis-service
NAME            TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)
redis-service   ClusterIP   10.43.56.187   <none>        6379/TCP

controlplane ~ ➜  kubectl describe svc redis-service
Name:              redis-service
Namespace:         default
Labels:            tier=db
Annotations:       <none>
Selector:          tier=db
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.43.56.187
IPs:               10.43.56.187
Port:              <unset>  6379/TCP
TargetPort:        6379/TCP
Endpoints:         10.42.0.10:6379
Session Affinity:  None
Events:            <none>

controlplane ~ ➜  
```



Create a new pod called `custom-nginx` using the `nginx` image and expose it on `container port 8080`.

✓ Pod created correctly?

```
controlplane ~ ➜  kubectl run custom-nginx --image=nginx --port=8080
pod/custom-nginx created

controlplane ~ ➜  
```

Kubectl run httpd –image=httpd –port=80 –expose=true

Apply command

Local last applied live object