
CKAD, Certified Application Developer:

Multi Container Pods: When you need to have 2 or more containers in the same pod. Like a webserver and a loggin agent.

You want both to be in the same pod but separated into two containers.

If you need to scale the number both containers will be scaled at the same time.

They share the same lifecycle, they are created and destroyed together, share the same network space, which means they can refeer to each other as local host, and they have access to the same storage volumes.

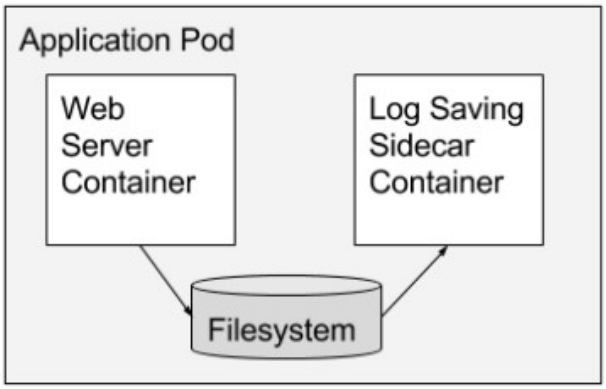
This way you don't have to stablish volume sharing or services between the pods to enable communication between them.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp
  labels:
    name: simple-webapp
spec:
  containers:
  - name: simple-webapp
    image: simple-webapp
    ports:
      - containerPort: 8080

  - name: log-agent
    image: log-agent
```

Design Patterns: More extense explanation

- **Sidecar container:** An example is a web server with a loggin agent to collect logs and send them to a loggin server.



- **Adapter container:** If you have different applications generating logs in multiple formats you can use an adapter container that will convert the logs to a common format and then will send them to the centralized loggin server.

  Example, you have:
  - 1 Pod with a **Ruby** app that writes logs: [DATE] - [HOST] - [DURATION]
  - 1 Pod with a **Node.js** app that writes logs: [HOST] - [START_DATE] - [END_DATE]
  - 1 Adapter container in Node.js pod. Read the explanation before to understand this.

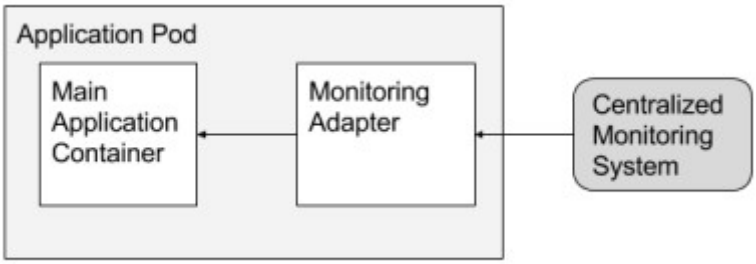
  Let's say the application collecting the logs, like Prometheus or Elastic Stack, only accepts logs in Ruby format, [DATE] - [HOST] - [DURATION]

  You could force the Node.js app to write logs in the Ruby format but that burdens the application developer, and there might be other things depending on the Node.js logs format.

  The better alternative is to provide adapter containers that adjust the output into the desired format.

  The app developer can simply update the pod definition to add the adapter container adding a logic that reformat the output logs from the main container:

Adapter Container:
image: alpine
command: ["/bin/sh"]
args: ["-c", "while true; do (cat /var/log/top.txt | head -1 > /var/log/status.txt) && (cat /var/log/top.txt | head -2 | tail -1 | grep
 -o -E '\\d+\\w' | head -1 >> /var/log/status.txt) && (cat /var/log/top.txt | head -3 | tail -1 | grep -o -E '\\d+%' | head -1 >> /var/log/status.txt); sleep 5; done"]



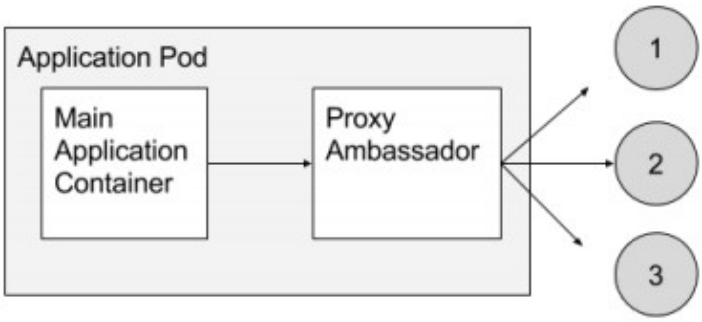
- **Ambassador container:** Is in charge of proxying connections from the application container to other services.

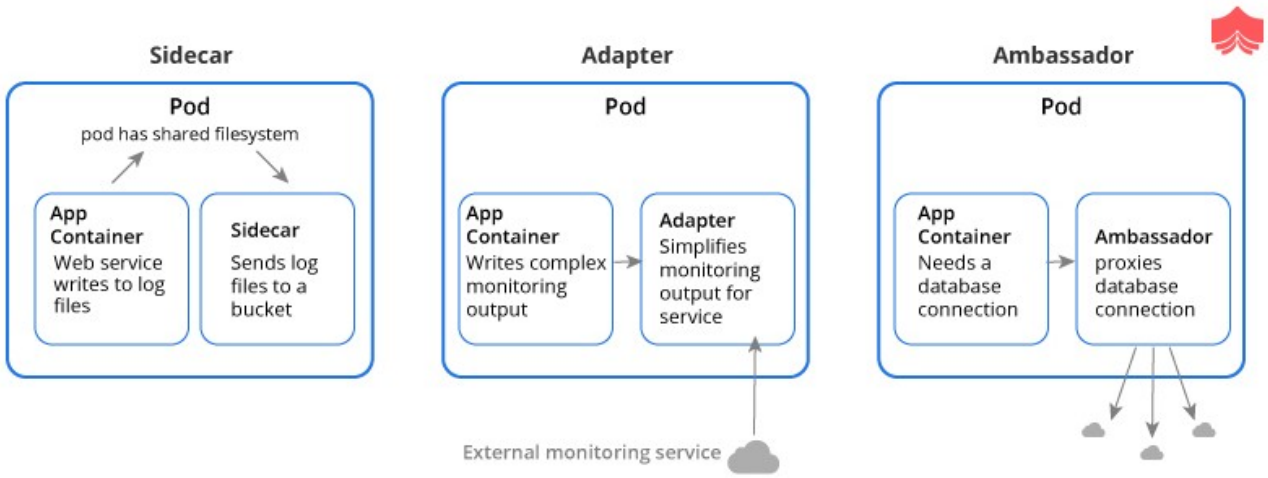
  Acts as a client proxy, it proxies a local connection to the world.

  Best use cases for ambassador pattern is for providing access to a DB.
  When developing locally, you probably want to use your local db.
  While your test or production deployments want a different db.

  You could manage which database to connect to by using environment variables, but will mean that you will need to modify the URL of your application depending on the environment.

A better solution is to always point to localhost and let the mapping responsibility to the right db to the ambassador container.





Clarifying note: When you implement these paterns using a pod definition file it is always the same configuration for all of them.

You simply have multiple containers within the pod definition file.

So all 3 patterns are implemented as a multiple container. All of them will have a name, image, commands, etc.

A thing that could change is in the main container code to use the sidecar container and this may be done by the developers.

Observability:

Stages in the life cycle of a pod:

A pod has a:

- **Pod Status:** Which tells us where the pod is in its lifecycle.
    - **Pending State:** When the pod is created is in this state. This is when the scheduler is looking for a node in which to place the pod.

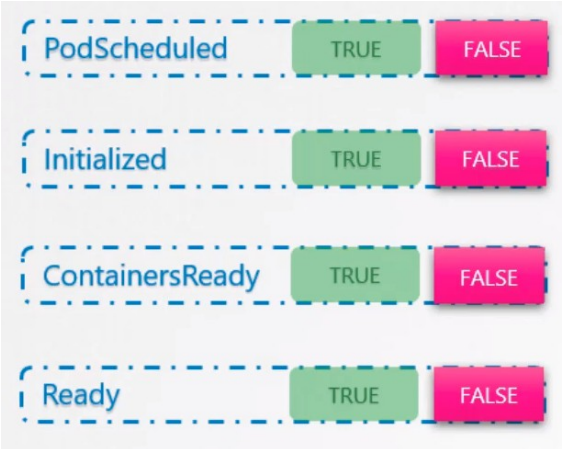
      If the scheduler can not find a node to place the pod it remains in a pending state.

      To find out why the pod is in pending state run k describe pod to know why.
    - **ContainerCreating status:** Where the images required for the application are pulled and the container starts.
    - **Running state:** Once all the container in the pod starts it goes into the running state. Where is continuous to be until the program completes successfully or is terminated.

You can see the container status at k get pods STATUS column. You only can have one status for each pod.

The STATUS column gives us a high level summary of a pod, to go deeper we have pod conditions.

- **Pod Conditions:** Array of true or false values that tell us the state of a pod.



PodScheduled: True when a pod is scheduled in a node.

Initialized: Value set to true when the pod is initialized.

ContainersReady: Set to true when all the containers within a pod are ready

Ready: The pod itself is considered to be ready.

Look for the Conditions section in the k describe pod command or in k get pods in the READY column.

Ready Condition: Indicates that the application/s inside the pod is running and is ready to accept user traffic.

The containers could be running different type of applications in them, like a script that performs a job, a db service, a webserver, etc.

The script may take a few miliseconds to get ready, but db service may take a few seconds to power up.

Some web servers could take several minutes to warm up.

Example, if you try to run an instance of a jenkins server you will notice that it takes between 10 and 15 seconds for the server to initialize before the user can access the Web UI.

Even when the web UI is initialized it takes a few second for the server to warm up and be ready to serve users.

During this wait period if you look at the state of the pod you will see it as ready and running wich is not very true.

Why does it matter if the state is reported incorrectly:
When a container is in ready status, the service attached to it will send client connections.

But if the application running inside the ready container is still powering up the client sent to that container will see a 404 error or something similar.

What we need here is a way to tie ready condition to the actual state of the application inside the container.

The developers of the application know what it needs for the application to be ready.

Ways to determine if an application inside a container is ready: Tests or Proves.

- **HTTP test:** In case of a web application you can set a prove to know then the API server is up and running. You can run a HTTP test to see if the API server response.
- **TCP socket test:** In case of a DB you can see if a tcp socket is listening.
- **Custom script:** You can execute a command within the container to run a custom script that would exit successfully if the application is ready.

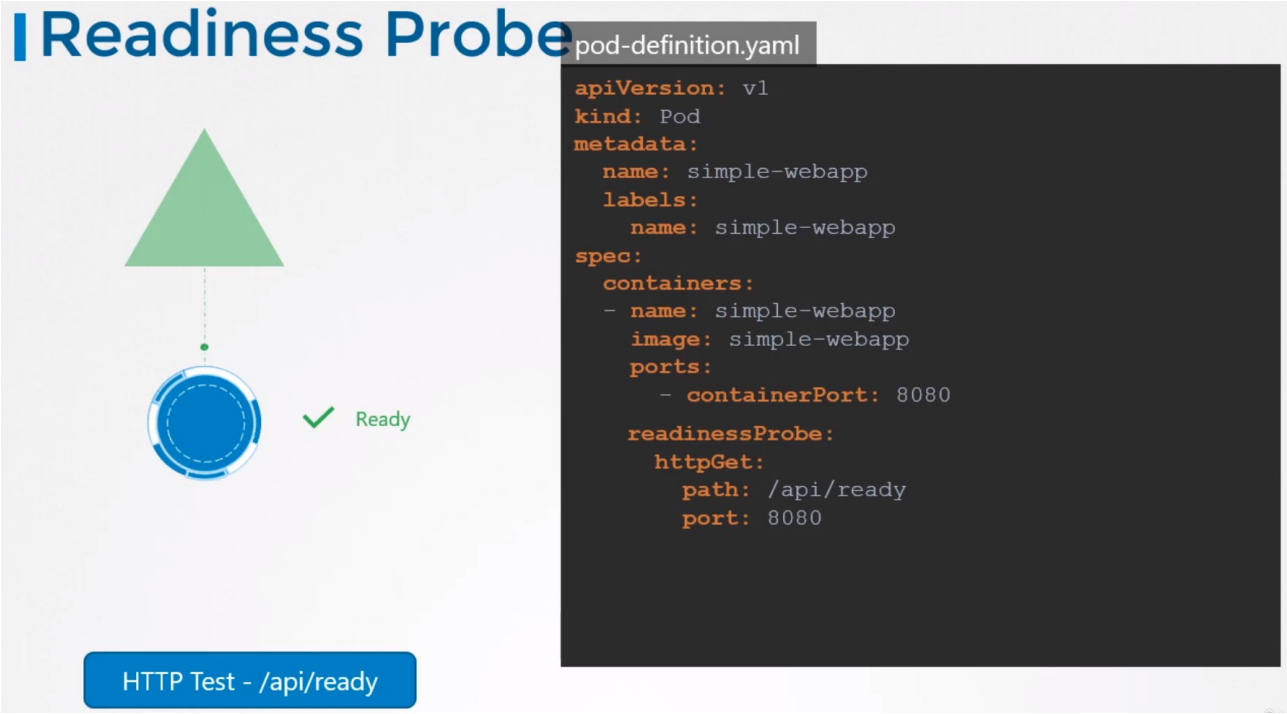
Readiness Proves: To configure those tests add the field readinessProbe inside the container definition.

E.g., for a web server use the HTTP test specifying httpGet option. Specify the port and the /api/ready path.

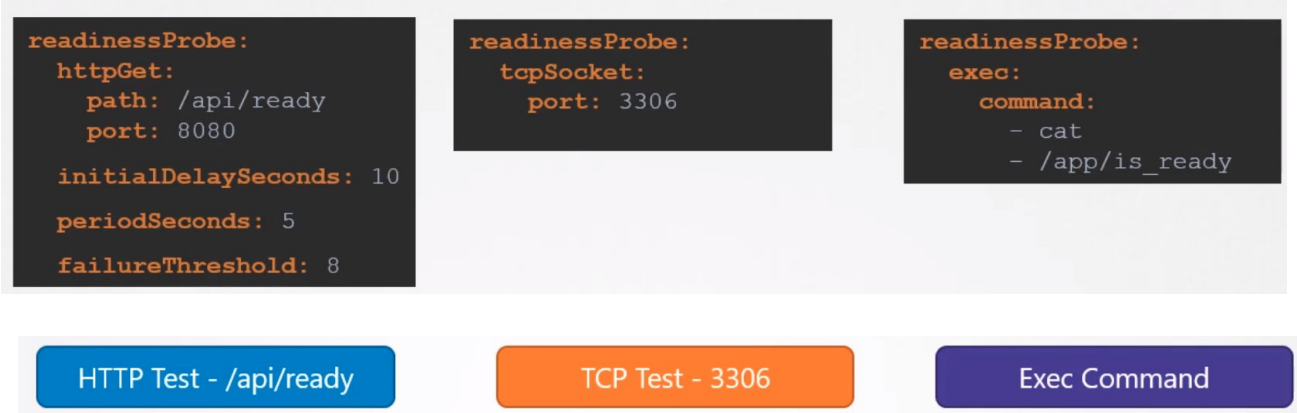
Now when the container is created k8s does not set immediately the ready condition on the container to True.

Instead it performs a test to see if the API response positivily.

Until then the service does not forward any traffic to the pod as it sees that the pod is not ready.



Ways a Probe can be configured:



HTTP: Use httpGet with the path and the port

TCP: Use the tcpSocket with the port.

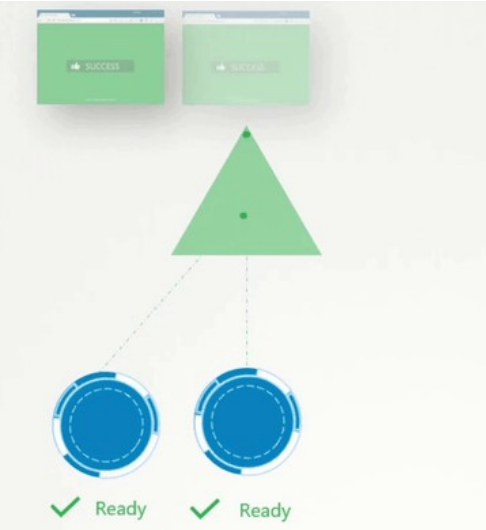
Command: Specify the exec option with the command options in an array format.

Additional options:

- **initialDelaySeconds:** If you know your application will take 10 seconds to warm up you can add them and the probe will start the test after these 10 seconds.
- **periodSeconds:** To specify how often to probe.
- **failureThreshold:** By default if the application is not ready after 3 probe attempts, the probe will stop. If you want to make more attempts use this option.
- There are more more options in the documentation

How readiness probes are useful into a multiple pod set up:

Without readiness probe:

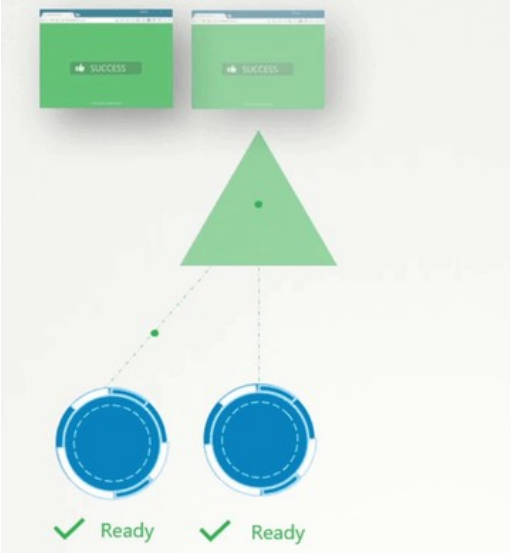


If a new pod is created the service will send clients to him once is in ready status.

But the application inside the container, a webserver in this example, is not ready yet, it needs a minute to power up.

Without readiness probe the client that is sent to the not ready yet server will face a 404 error or similar.

With readiness probe:



If readiness probe is set in the container definition the application will tell when is truly ready and won't show any error to any client connecting to it.

Liveness Probes: Used to know if the application within the container is healthy.

K8s will restart a container if detects that is failing.

But the application could be failing due a bug and to k8s concern the container is still up and running properly.

So you could have a failing application and k8s not noticing it and therefore not restarting the container.

To avoid this we have liveness probe, which checks periodically the state of the application running inside the container.

If the test fails the container is considered unhealthy and is destroyed and recreated.

The developers are the ones who define what it means for an application to be healthy.

Ways a Liveness Probe can be configured:

- **HTTP Test:** In case of a web application you could check if the /api/healthy is up and running.
- **TCP Socket:** In a db for instance, you can check if a particular tcp socket is listening.
- **Exec Command:** You can execute a command to perform a test.

Liveness Probe is configured like the readiness one:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp
  labels:
    name: simple-webapp
spec:
  containers:
  - name: simple-webapp
    image: simple-webapp
    ports:
      - containerPort: 8080
    livenessProbe:
      httpGet:
        path: /api/healthy
        port: 8080
```

Logs:

k logs -f pod_name → stream (live) the logs of a single container pod

k logs -f pod_name container_name → stream the logs of the specific container in a multi container pod

Monitoring and Debug Applications:

For the CKAD you only need to know the Metrics Server, which is an in-memory solution, doesn't stores data.

The kublet service in each node contains a subcomponent known as **cAdvisor** which retrieve performance metrics from pods and expose them to the kubelet API to make the metrics available for the Metrics Server.

Install Metrics Server in the cluster:
git clone https://github.com/kubernetes-sigs/metrics-server

cd the metrics server folder

k create -f .

Run commands: MS will need a few minutes to have data to show.
k top node
k top pods

Pod Design:

Annotations: Are used to record other details for information purposes. Like version details, contact email or phone, etc.

Jobs: A job is use to run a set of pods to perform a given task to completion.

K8s wants your applications to live forever. The default behaviour of pods is to attempt to restart the container in an effort to keep it running.

RestartPolicy: This behaviour is defined by the RestartPolicy set on the pod, by default is set to always.

- OnFailure
- Never

You can change this policy to Never, which won't restart the container on finish, or OnFailure.

spec.BackoffLimit: To specify the number of retries before considering a job failed.
By default is 6. Info here

spec.completions: The number of completed jobs that we require. The job won't stop until you achieve the completion count.

Create a Job:

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: math-add-job
spec:
  template:
    spec:
      containers:
        - name: math-add
          image: ubuntu
          command: ['expr', '3', '+', '2']

      restartPolicy: Never
```

Create the job. You will see the status as completed when is done and will have 0 restarts because the restart policy is set to 0.

If you check the logs of the pod you will see the number 5 which is 3 + 2.

To delete the job run: k delete job name_of_job

Deleting the job will also result in deleting the pods created by the job.

Run several Jobs:

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: math-add-job
spec:
  completions: 3
  template:
    spec:
      containers:
        - name: math-add
          image: ubuntu
          command: ['expr', '3', '+', '2']

      restartPolicy: Never
```

```
kubectl create -f job-definition.yaml
```

```
kubectl get jobs
NAME           DESIRED   SUCCESSFUL   AGE
math-add-job   3         3            38s
```

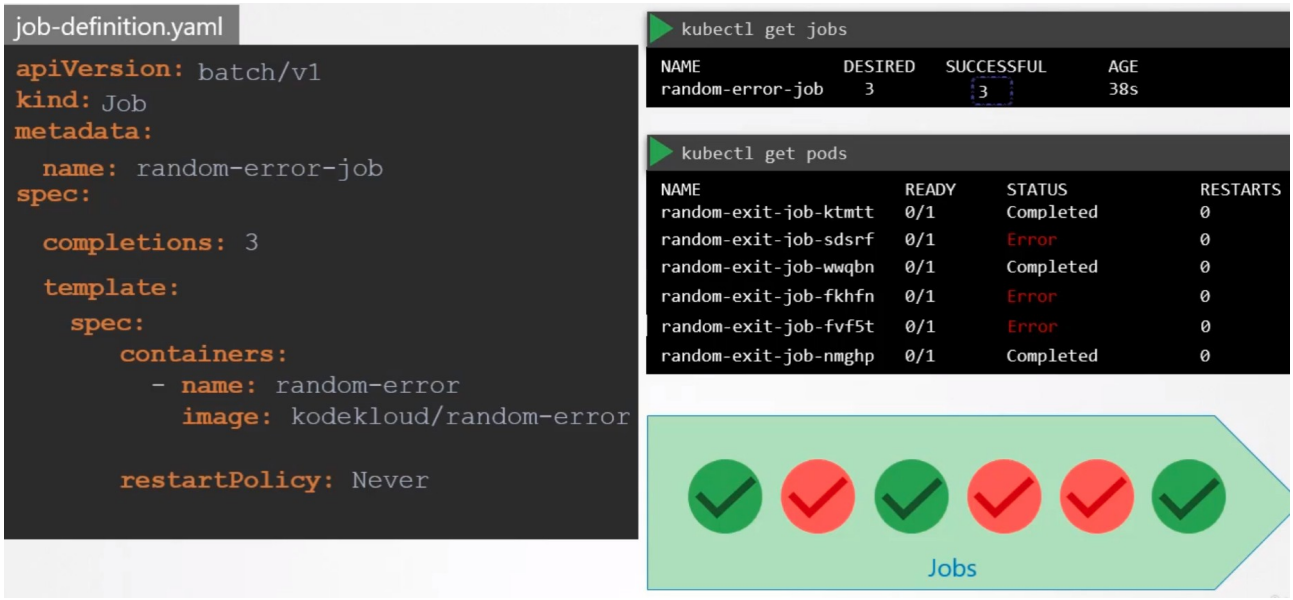
```
kubectl get pods
NAME                 READY   STATUS      RESTARTS   AGE
math-add-job-25j9p   0/1     Completed   0          2m
math-add-job-87g4m   0/1     Completed   0          2m
math-add-job-d5z95   0/1     Completed   0          2m
```

We will create 3 pods here, see completions: 3

By default the pods are created one after the other. The second pod is created only after the first is finished.

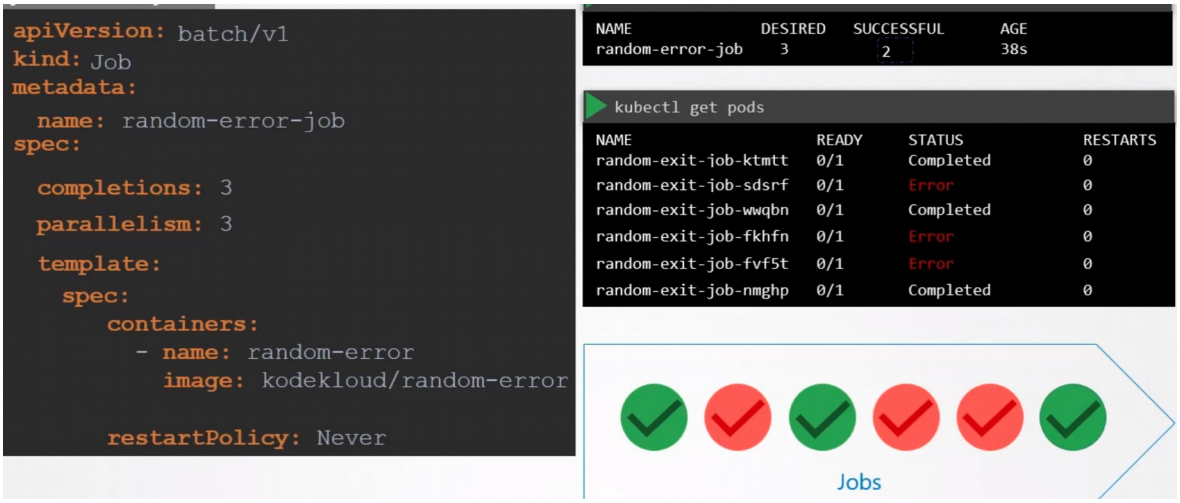
Another example:

If the Job have 3 **completions** it won't stop until it have all 3 completed.



If some jobs are failing, like in the image above, it will spin up pods until the 3 completion are succesfull.

Parallelism: Instead of creating the pods secuencially we can create them in parallel. Just add the parallelim property to the job.



In this example we have a parallelism of 3, which means that will create 3 pods at the same time.

But if some pods fail the job is intelligent enough to create the remaining ones until we complete the number requested, 3 in this example.

Commands:
Create a job in imperative way:
k create job name_job --image nginx -oyaml > job.yaml

k -n neptune create job neb-new-job --image=busybox:1.31.0 $do > /opt/course/3/job.yaml -- sh -c "sleep 2 && echo done"

Cron Jobs: Job that can be scheduled.

```yaml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: reporting-cron-job
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      completions: 3
      parallelism: 3
      template:
        spec:
          containers:
            - name: reporting-tool
              image: reporting-tool

          restartPolicy: Never
```

The jobTemplate comes from the Job, in this case from the previous Job example.

We have now 3 spec sections: one for the CronJob, the other for the Job and the last for the Pod.

Create the cronjob, k get cronjob to see it's status.

Command: Create a cronjob to be executed every day at 21:30:
k create cronjob throw-dice-cron-job --image kodekloud/throw-dice --schedule "30 21 * * *"

k create cronjob throw-dice-cron-job --image=kodekloud/throw-dice --restart=Never --schedule="30 21 * * *" $do > cron.yaml

---

State Persistence:

Stateful Sets:

Are similar to deployments in:

- They create pods based on a template.
- They scale up and down.
- Can perform rolling updates and rollbacks.

Different to deployments in:

- Pods are created in an ordered, graceful deployment.
  After the first pod is created it must be running and in ready state before the next pod is deployed.
- Stable, unique DNS record on the network identifier.
- Assign a unique ordinal index to each pod in numbers starting from 0 for the first pod and increments by one.
- Each pod gets a unique name derived from this index, combined with the stateful set name.The first pod will always be name_stateful_set-0.
  Example: mysql-0, mysql-1, mysql-2, etc
- Even if the pod fails and needs to be recreated it will have the same name. Sticky identity.

- Scaling. Each new instance will clone from the previous one. Scaling down works the same, the last created will be the first removed.
- podManagementPolicy: Parallel → you scale up or down all the pods at the same time, but you keep the other advantages of stateful sets, like unique name. Default value of this field is: ordered ready

Why Stateful Sets: Example, we wan't to create a mysql cluster with 1 master and 2 slaves.

First we create the master, then the slave 1.
We ask slave 1 to clone data from master and enable continuous replication from master to slave 1
We wait for slave 1 to be ready
We create slave 2 and we clone data from slave 1.
Enable continuous replication from master to slave 2
Configure master address on slave

To do all this we will need to have permanent hostnames, not randomly created that will change if the pod is destroyed.

We will need the pods to be created in order, secuencially, first the master, then the slave, then the other slave, etc.

You can do all this with Stateful Sets.

Creating Stateful Sets: Like creating a deployment. Just change the kind to StatefulSet

You will need to specify as well a serviceName, which have to be a HeadLess service.

Headless Services:

What is it: Is like a service but does NOT have an IP on its own (like ClusterIP) and does NOT perform load balancing.

All it has is create DNS entries for each pod using the pod name and the subdomain.

When you create a Headless Service, e.g., mysql-h, each pod gets a DNS record created in the form of:

1. podName. → mysql-0 , mysql-1, mysql-2, etc
2. name of headless service. → mysql-h.
3. namespace.
4. svc.
5. cluster-domain → cluster.local

mysql-**0.**mysql-h.default.svc.cluster.local

mysql-**1**.mysql-h.default.svc.cluster.local

mysql-**2.**mysql-h.default.svc.cluster.local

Platform example:
You have 3 mysql servers. One have to be the master with read/write access, the others are the slaves with read access.

You have a web server that have to access the mysql master to write, and have to access the slaves to read.
You can not use a normal Service because it does load balancing, so will send randomly to all the mysql server writing requests.

So you create a Headless Service which does not do lb and doesn't have it's own IP.

The headless service create a DNS record of all the mysql servers and depending on the webserver needs will send its requests to the master or the slaves.

How to create a Headless Service: Create a service definition file as usual. What differentiates the Headless Service from a normal service is that you have to set the clusterIP to none.

By doing so it creates a headless service.

```yaml
apiVersion: v1
kind: Service
metadata:
   name: mysql-h
spec:
  ports:
   - port: 3306
  selector:
      app: mysql
  clusterIP: None
```

Kind is StatefulSet, there is a typo in the Image.

See **spec.serviceName:** Is how you map the headless service to the Statefull Set

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql-deployment
  labels:
      app: mysql

spec:
  serviceName: mysql-h
  replicas:  3
  matchLabels:
     app: mysql
  template:
     metadata:
      name: myapp-pod
      labels:
        app: mysql
     spec:
       containers:
       - name: mysql
         image: mysql
```

With the spec.serviceName: mysql-h in this example, you map the StatefulSet to the Headless service.

When creating a stateful set you don't need to specify a hostname or subdomain.

Stateful Set automatically assigns the right hostname for each Pod based on the hostname and it automatically assigns the right subdomain based on the Headless Service name.

The only thing you have to remember is to specify spec.serviceName to map it to the headless service. That's how it knows what subdomain to assign to the pod.

The stateful set takes the names that we've specified and adds that to as a subdomain property when the pod is created.

All pods now get a separate DNS record created.

Recap on storage:

What we know about storage in k8s until now is that with pv we create volume objets which are then claimed by pvc and finally used in Pod definition files within pods.

That's a single pv mapped to a single pvc to a single pod definition file.

With dynamic provisioning with the sc definition we take out the manual creation of pv and use storage provisioners to automatically provision volumes on cloud providers.

The pv is created automatically. We still create a pvc manually and associate that to a pod.

All this works fine with a single pod with a volume.

How does that change with deployments of stateful sets:

With Stateful Sets when you specify the same pvc under the pod definition, all pods created by that stateful set tries to use the same volume.

If you want multiple pods or multiple instances of your application to share and access the same storage its fine, that's how you configured. That also depends on the kind of volume and the provisioner used.

Note that NOT all the storage types supports that operation, ReadWriteMany (read and write from multiple instances at the same time).

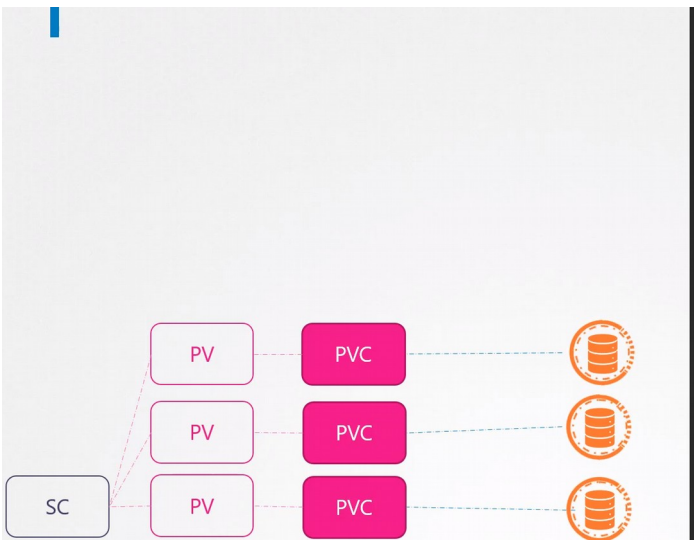
But, what if you want separated volumes for each pod, as the Mysql replication use case that we have been talking about.

The pods don't want to share data, instead each pod needs its own local storage.

Each instance has its own database and the replication of data between the databases is done at the database level.

So each pod needs a pvc for itself. A pvc is bound to a pv, so each pvc needs a pv and the pv can be created by a single or different storage classes.

How do you automatically create a pvc for each pod in a stateful set?




```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
  labels:
    app: mysql
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql
        volumeMounts:
         - mountPath: /var/lib/mysql
           name: data-volume
      volumes:
        - name: data-volume
          persistentVolumeClaim:
            claimName: data-volume
```

Storage in StatefulSets: If you need to have a different pvc for each pod of the stateful set you can achieve this by using a VolumeClaimTemplate.

VolumeClaimTemplates: Is really a pvc templetized and injected into the pod definition on the stateful set.

Instead of creating a pvc manually and then call it from the stateful set definition file, you move the entire pvc definition into a section named VolumeClaimTemplates under the stateful set specification.

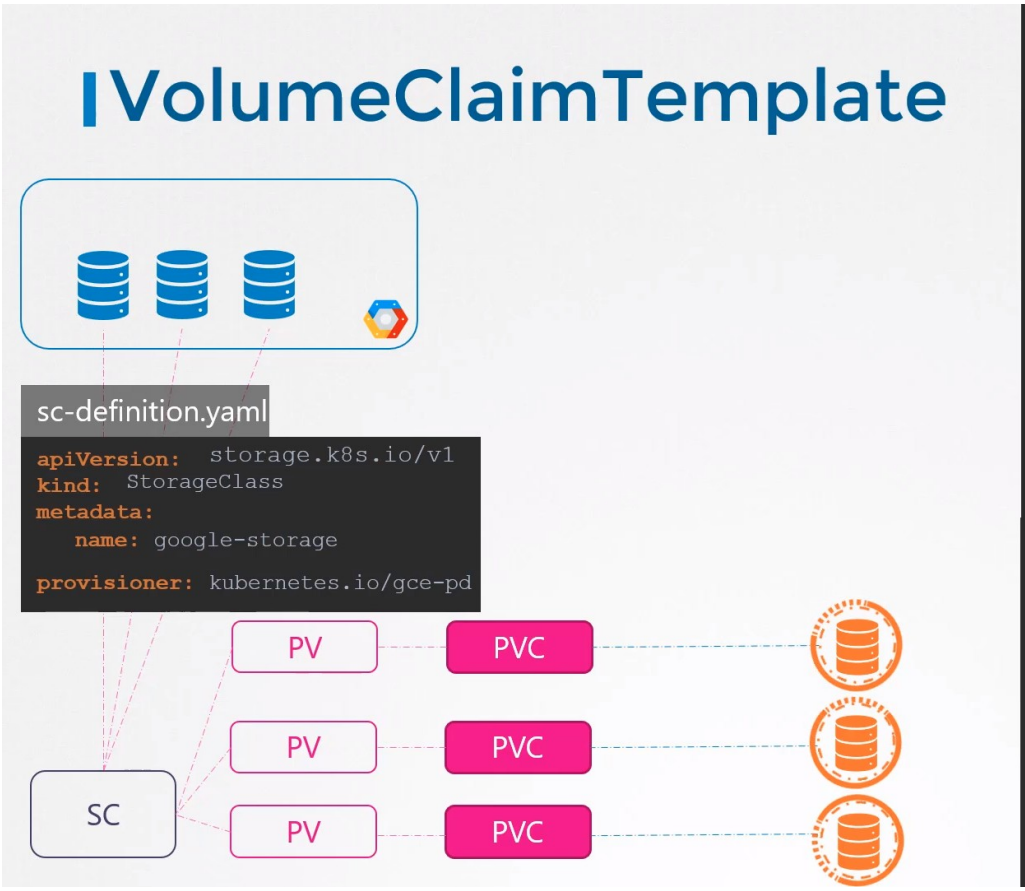
Is an array so you can specify multiple templates.

Topology: So now we have:

1. A stateful set with VolumeClaimTemplates.
2. A StorageClass provisioner for GCE

Process: When the stateful set is created:

1. First it creates the first Pod
2. During the creation of the pod a pvc is created automatically
3. The pvc is associated to the sc
4. The sc creates the volume in the cloud provider (GCE in this example)
5. The sc creates the pv and associates it to the volume in the cloud and binds the pvc to the pv
6. Then the second pod is created
7. The second pod creates the pvc
8. Then the sc provisions a new volume
9. etc for the rest of the pods, 3 replicas in this example.




```yaml
metadata:
  name: mysql
  labels:
    app: mysql
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql
        volumeMounts:
          - mountPath: /var/lib/mysql
            name: data-volume
  volumeClaimTemplates:
  - metadata:
      name: data-volume
    spec:
      accessModes:
        - ReadWriteOnce
      storageClassName: google-storage
      resources:
        requests:
          storage: 500Mi
```

What if one of these pods fails and is recreated?
Stateful Set do NOT automatically delete the pvc or the associated volume to the pod.

Instead it assures that the pod is reattached to the same pvc that was attach to before.

That's how stateful sets ensure stable storage for pods.

Define, Build and Modify Container Images:

Create the image:

Dockerfile

```
FROM Ubuntu

RUN apt-get update
RUN apt-get install python

RUN pip install flask
RUN pip install flask-mysql

COPY . /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask run
```

1. OS - Ubuntu
2. Update apt repo
3. Install dependencies using apt
4. Install Python dependencies using pip
5. Copy source code to /opt folder
6. Run the web server using "flask" command

Build the image:

```
docker build Dockerfile -t mmumshad/my-custom-app
```

Push it to dockerhub:

```
docker push mmumshad/my-custom-app
```

Docker file:

- **Instructions:** Everything on the left in capitals are instructions. Each of this instructs docker to perform a specific action while creating the image.

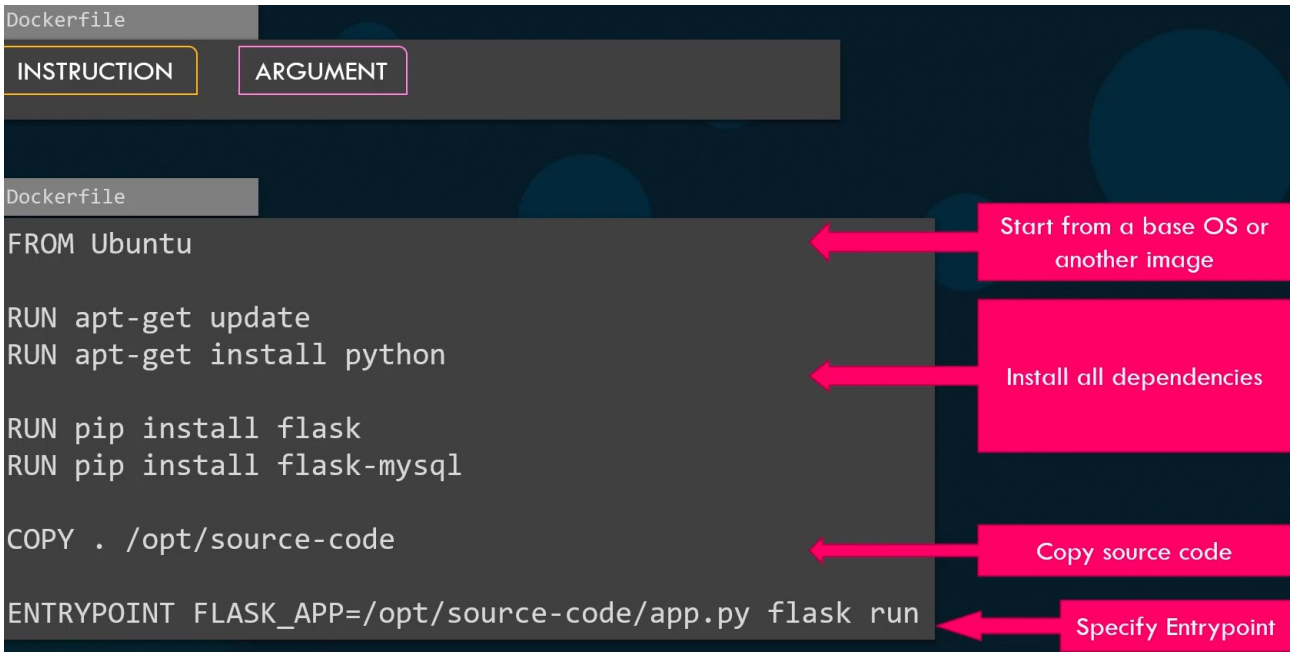
- **Arguments:** Everything on the right are arguments to perform those instructions.

FROM Ubuntu → Defines what the base OS should be for this container.
Every docker image must be based of another docker image. Either an OS or another image that was created before based on a OS.
All docker files MUST start with the FROM instruction.

RUN → Instructs docker to run a particular command on those based images. In this example docker runs apt-get command to fetch the updated packages and installs required dependencies on image.

COPY → Copies files from the local system onto the docker image. In this case the source code of the application is in the current folder (.) and will copy it to /opt/source-code inside the docker image

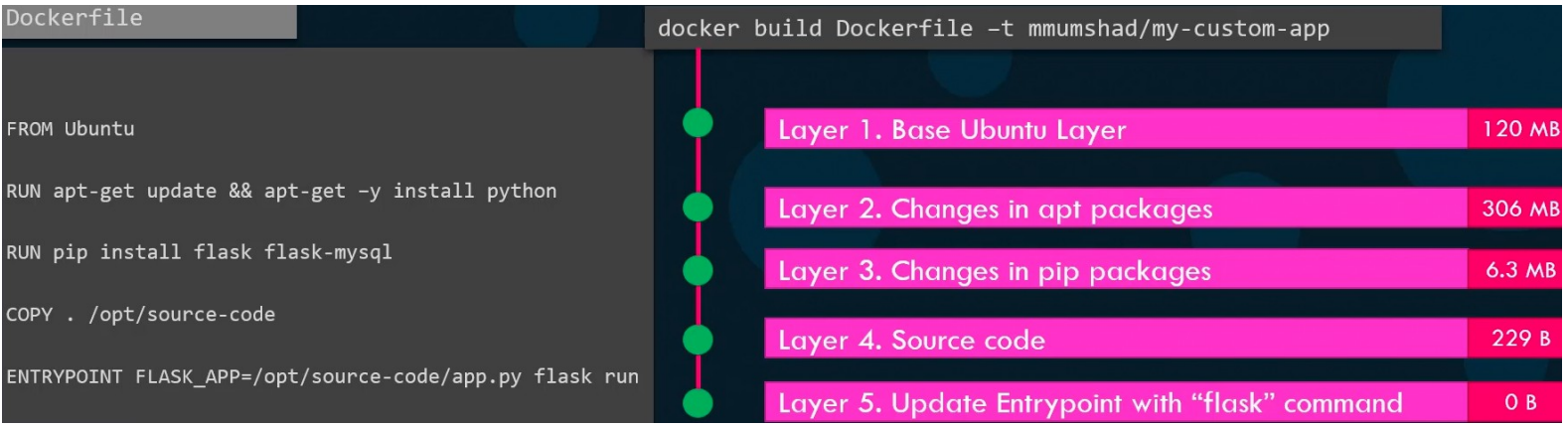
ENTRYPOINT → Allows us to specify a command that will be run when the image is run as a container.



Layered architecture: When docker builds the images it build this in a layered architecture.

Each line of instruction creates a new layer in the docker image with just the changes of the previous layer.

For example:



Each layer only stores the changes on the previous layer it is reflected on the size as well.

You can see this information if you run the docker history command followed by the image name.

When you run the docker build command you can see the various steps involved and the result of each task.

All the layers build are cached so the layer architecture helps you to restart docker build from that particular step in case it fails of if you want to add new steps in the build process it would not have to start over again.

This way rebuild an image is faster as you don't need to wait to rebuild the entire image each time.

Helpfull when you update source code in your application as it may change more frequently.

Only the layers above the updated layer needs to be rebuild.

What can you containerize: You can containerize everything.
Databases, development tools, OS, browsers, utilities like curl, applications like spotify, skype, etc.

Commands:
docker build -t webapp-color .

docker history webapp-color/

Run the image: and publish port 8080 on the container to 8282 on the host.
docker run -p 8282:8080 webapp-color

This will leave the container running, to stop it → ctrl + c

See the OS of a running image: python with tag 3.6
docker run python:3.6 cat /etc/*release*

Run the container in the background: → -d
docker run -p 8383:8080 webapp-color:lite -d

docker ps -a

docker kill my_container

Security Primitives:

Secure the hosts:

- Password based authentication disabled
- SSH key based authentication

Secure K8s:
kube-apiserver is at the center of all operations within k8s by interacting with kubectl or by accessing the API directly. You can perform any operation on the cluster.

This is the first line of defense, control the access to the kube-apiserver itself.

Authentication Mechanisms:

Who can access the cluster:

- Files -Usernames and Passwords
- Files - Usernames and Tokens
- Certificates
- External Authentication providers - LDAP
- Service Accounts

Authorization Mechanisms:

What can they do:

- **RBAC Authorization**

- **ABAC Authorization** → you have to reload the node in every new change.
- **Node Authorization** → used for example with kubelet to talk with kube-apiserver
- **Webhook Mode** → To outsource all the authorization mechanisms. Open Policy Agent, 3rd party tool that helps with admision control authorization.
- **AlwaysAllow:** Allows all request without performing authorization checks.
- **AlwaysDeny:** Denies all request without performing authorization checks.

Where to configure the modes: On the kube-apiserver config file.
--authorization-mode=RBAC

Default option is AlwaysAllowed

You can specify multiple modes: It works in the order specified.
--authorization-mode=Node,RBAC,Webhook

TLS Certificates:
All communications within the cluster between the various components, etcd, scheduler, api-server, kubelet, kube-proxy, etc, is secured using tls encryption.

Communication between applications within the cluster:
By default all pods can access all other pods within the cluster.
You can restrict it using network policies.

Admission Controllers:

Function of Admission Controllers:

- Validate configuration
- Perform additional operations before the pod gets created
- Help us to implement better security measures

Get enabled by default Admission Controllers: In a kubeadm cluster
kubectl exec -it kube-apiserver-controlplane -n kube-system -- kube-apiserver -h | grep 'enable-admission-plugins'

There you can see all the Admission Controllers that are enabled by default

Add Admission Controllers: That are not as default.
Into a kubeadm cluster

Edit /etc/kubernetes/manifests/kube-apiserver.yaml

- --enable-admission-plugins=NodeRestriction,NamespaceAutoProvision

Deny Admission Controllers: That are already allowed as default
Edit /etc/kubernetes/manifests/kube-apiserver.yaml

- --disable-admission-plugins=DefaultStorageClass

Implementing order:

1. **Mutating** → first the system checks if the request needs to be modified by adding something, like the namespace, sc, etc.

2. **Validating** → then it checks if have to be validated.

Commands:

Get default enabled Admission Controllers: In a kubeadm cluster
kubectl exec -it kube-apiserver-controlplane -n kube-system -- kube-apiserver -h | grep 'enable-admission-plugins'

Get manually added Admission Controllers: If kube-apiserver is running as a pod
ps -ef | grep kube-apiserver | grep admission-plugins

Create your own Admission Controllers:
Dynamic Admission Control are Admission Webhooks that we can use to create our own mutating or validating webhooks.

Like the Ingress Controllers, to deploy a MutatingWebhookConfiguration or a ValidatingWebhookConfiguration you need to create:

- A deployment.
- A service
- A MutatingWebhookConfiguration or ValidatingWebhookConfiguration object
- A ns
- A secret

What are Admission Controllers: Help us implement better security measures to enforce how a cluster is used.

Besides of simply validating configuration Admission Controllers can do a lot more, such as change the request itself, or perform additional operations in the backend before the pod gets created.

Steps when running kubectl commands:

1. Type the command in kubectl → k create pod

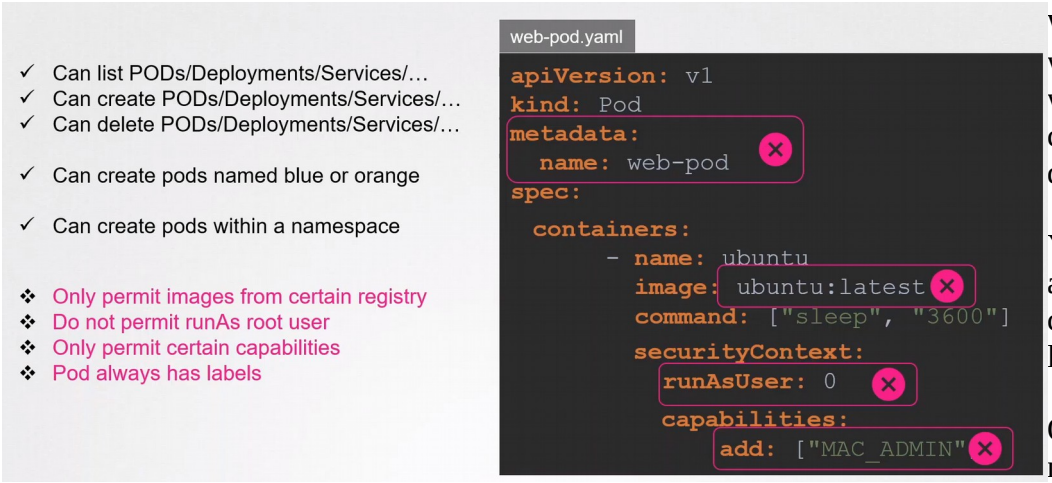
2. The request goes to the kube-apiserver. When the request hits the api-server it goes to:

    1. **Authentication process:** Usually done through certificates, which are in ~/.kube/config.

       The Authentication process is responsible for identifying the user who sends the request and making sure the user is valid.

    2. **Authorization process:** We check if the user have permissions to perform that operation.

       This is achieved by RBAC. If the request that came in match one of the conditions of the role in which the user is, it is allowed to go though, otherwise is rejected.



With RBAC you can say which actions over which resource the user can do, create, list, delete, etc.

You can specify to only apply this action over a certain pods with certain labels.

Or over a certain namespace.

But you can not permit images from a certain registry or not allow to use images with the “latest” tag or not permitting to run the pod as root user, or permit certain capabilities or to enforce the metadata field to always contain labels.

You can do all this with Admission Controllers.

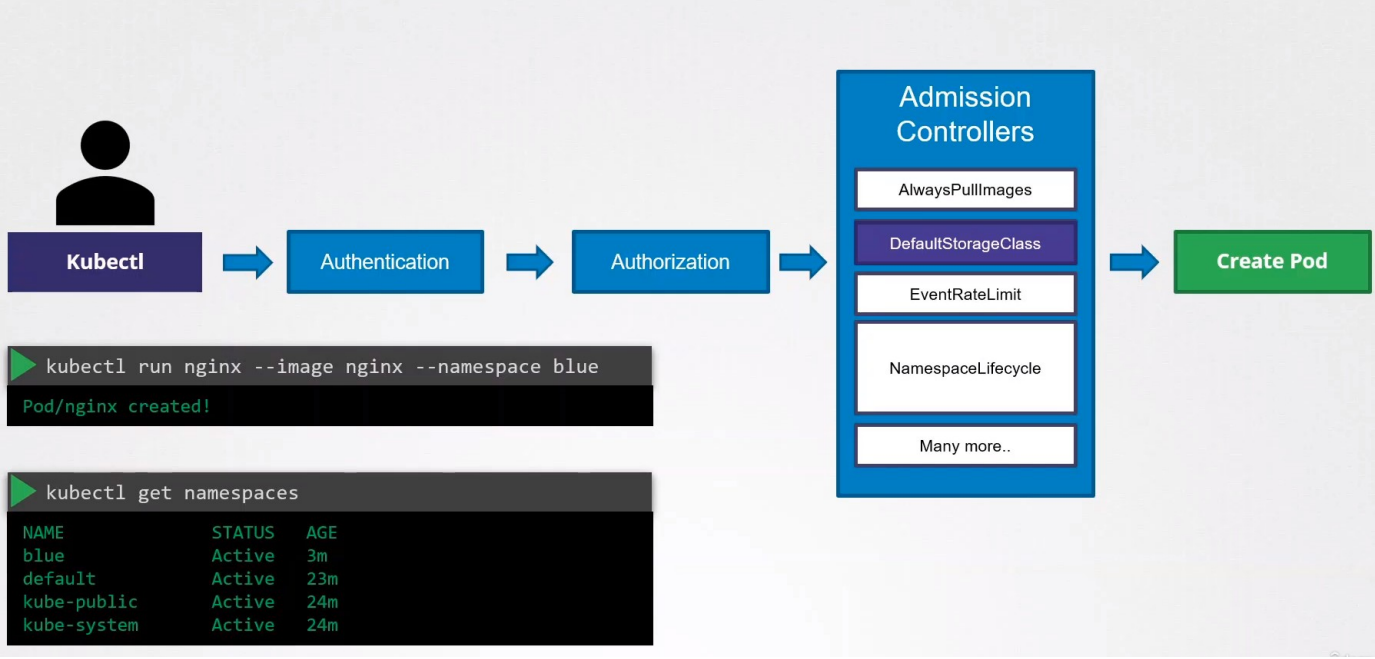
    3. **Admission Controllers:** Allows you to specify actions like the above mentioned, there are many more.

3. The pod is created

4. The information is persisted in the etcd database

Admission Controllers:
There are many of them that come pre-build with k8s. Such as:

- **AlwaysPullImages:** Ensures that every time a pod is created the image is always pulled
- **DefaultStorageClass:** Observes the creation of pvc and automatically adds a default sc to them if none is specified. Not enabled by default
- **EventRateLimit:** Prevent api server from flooding with requests
- **NamespaceLifecycle:** If you want to create a pod in a particular namespace that does not exists, this ac will create it automatically. Is not enabled by default
- Many more...



See list of Enabled by default AC: (highlighted in green)

```
kube-apiserver -h | grep enable-admission-plugins
--enable-admission-plugins strings       admission plugins that should be enabled in addition to default enabled ones
(NamespaceLifecycle, LimitRanger, ServiceAccount, TaintNodesByCondition, Priority, DefaultTolerationSeconds,
DefaultStorageClass, StorageObjectInUseProtection, PersistentVolumeClaimResize, RuntimeClass, CertificateApproval,
CertificateSigning, CertificateSubjectRestriction, DefaultIngressClass, MutatingAdmissionWebhook,
ValidatingAdmissionWebhook, ResourceQuota). Comma-delimited list of admission plugins: AlwaysAdmit, AlwaysDeny,
AlwaysPullImages, CertificateApproval, CertificateSigning, CertificateSubjectRestriction, DefaultIngressClass,
DefaultStorageClass, DefaultTolerationSeconds, DenyEscalatingExec, DenyExecOnPrivileged, EventRateLimit,
ExtendedResourceToleration, ImagePolicyWebhook, LimitPodHardAntiAffinityTopology, LimitRanger, MutatingAdmissionWebhook,
NamespaceAutoProvision, NamespaceExists, NamespaceLifecycle, NodeRestriction, …. TaintNodesByCondition,
ValidatingAdmissionWebhook. The order of plugins in this flag does not matter.
```

Run this command to see the AC in a kubeadm based k8s cluster

```
kubectl exec kube-apiserver-controlplane -n kube-system -- kube-apiserver -h | grep enable-admission-plugins
```

ps -ef | grep kube-apiserver | grep admission-plugins
To add an AC: For kube-apiserver.service or kubeadm clusters in manifests folder.

kube-apiserver.service

```
ExecStart=/usr/local/bin/kube-apiserver \\
  --advertise-address=${INTERNAL_IP} \\
  --allow-privileged=true \\
  --apiserver-count=3 \\
  --authorization-mode=Node,RBAC \\
  --bind-address=0.0.0.0 \\
  --enable-swagger-ui=true \\
  --etcd-servers=https://127.0.0.1:2379 \\
  --event-ttl=1h \\
  --runtime-config=api/all \\
  --service-cluster-ip-range=10.32.0.0/24 \\
  --service-node-port-range=30000-32767 \\
  --v=2
  --enable-admission-plugins=NodeRestriction,NamespaceAutoProvision
```

/etc/kubernetes/manifests/kube-apiserver.yaml

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
    - --authorization-mode=Node,RBAC
    - --advertise-address=172.17.0.107
    - --allow-privileged=true
    - --enable-bootstrap-token-auth=true
    - --enable-admission-plugins=NodeRestriction,NamespaceAutoPro
    image: k8s.gcr.io/kube-apiserver-amd64:v1.11.3
    name: kube-apiserver
```

To disable AC add: --disable-admission-plugins=DefaultStorageClass

Example: Create the pod nginx in the blue namespace. Blue ns does not exist, don't create it. Just enable NamespaceAutoProvision in the kubeapi-server

vi /etc/kubernetes/manifests/kube-apiserver.yaml

- --enable-admission-plugins=NodeRestriction,**NamespaceAutoProvision**

k run nginx --image=nginx -n blue

The pod should be created and the blue namespace too.

NamespaceAutoProvision and NamespaceAutoProvision are deprecated and now replaced by: NamespaceLifecycle

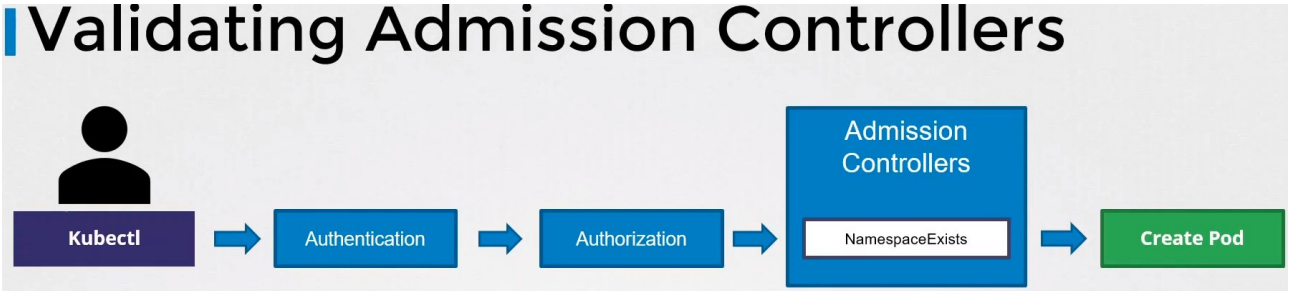
NamespaceLifecycle ac will reject any request to unexisting NS and will make sure that default NS such as default, kube-system and kube-public cannot be deleted.

Disable default storage class AC:
This Admission Controller assign a PVC to the default storage class if the creator of the pvc does not specify any sc.

Add it into the kube-apiserver manifest:
- --disable-admission-plugins=DefaultStorageClass

Check admission plugins:
ps -ef | grep kube-apiserver | grep admission-plugins

Validating and Mutating Admission Controllers:



(NamespaceExists is deprecated, now is called NamespaceLifecycle)

Validating Admission Controller: Are those that can validate the request.

Mutating Admission Controller:

Another example with DefaultStorageClass:
We submit a request to create a pvc. The request goes through Authentication and authorization and finally to Admission Controller.

The pvc DefaultStorageClass admission controller will watch or request to create a pvc and check if it has a storage class mentioned in it.

If not, which is true in this example, will modify the request to add the storage class to the request.

This sc will be the one that the cluster have configured as default.

This is called mutating ac. It can change or mutate the object itself before the object is created.

Both: There are AC that can do both, validating and mutating.

Order: Mutating is involved first, followed by validating admission controllers. Because any mutating object can be validated during the validation process.

Configure your own Admission Controller:
To support external Admission Controllers there are two special admission controllers available:

- MutatingAdmissionWebhook
- ValidatingAdmissionWebhook

Point the external Admission Controllers:
We can configure the webhooks to point to a server that's hosted either in the k8s cluster or outside it.

In our server we have our own admission controller service running with our own code and our own logic.

After a request goes through all the build in admission controllers it hits the webhook that's configured.

Once it hits the webhook it makes a call to the admission webhook server by passing in an admission review object in json format.

This object has all the details about the request such as the user that makes the request and the type of operation that the user is trying to perform and on what object and details about the object itself.

On receiving the request the admission webhook server response with an admission review object with a result of whether the request is allowed or not.

If the allowed field is true then is allowed and if false then is denied.

How to set this up:

1. Deploy an admission webhook server.
2. Configure the webhook in k8s by creating a webhook configuration object.

Deploy our own admission webhook server:

Could be an API server build in any platform. Code example of a webserver written in GO

```go
package main

import (
        "encoding/json"
        "flag"
        "fmt"
        "io/ioutil"
        "net/http"

        "k8s.io/api/admission/v1beta1"
        metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
        "k8s.io/klog"
        // TODO: try this library to see if it generates correct json patch
        // https://github.com/mattbaird/jsonpatch
)

// toAdmissionResponse is a helper function to create an AdmissionResponse
// with an embedded error
func toAdmissionResponse(err error) *v1beta1.AdmissionResponse {
        return &v1beta1.AdmissionResponse{
                Result: &metav1.Status{
                        Message: err.Error(),
                },
        }
}

// admitFunc is the type we use for all of our validators and mutators
type admitFunc func(v1beta1.AdmissionReview) *v1beta1.AdmissionResponse

// serve handles the http portion of a request prior to handing to an admit
// function
func serve(w http.ResponseWriter, r *http.Request, admit admitFunc) {
        var body []byte
        if r.Body != nil {
                if data, err := ioutil.ReadAll(r.Body); err == nil {
                        body = data
                }
        }
```

You can build your own code in different languages.

The only requirement is that must accept to mutate and validate API and must respond with the json object that the webserver expects.

Another example written in python.

```python
@app.route("/validate", methods=["POST"])
def validate():
    object_name = request.json["request"]["object"]["metadata"]["name"]
    user_name = request.json["request"]["userInfo"]["name"]
    status = True
    if object_name == user_name:
        message = "You can't create objects with your own name"
        status = False
    return jsonify(
        {
            "response": {
                "allowed": status,
                "uid": request.json["request"]["uid"],
                "status": {"message": message},
            }
        }
    )


@app.route("/mutate", methods=["POST"])
def mutate():
    user_name = request.json["request"]["userInfo"]["name"]
    patch = [{"op": "add", "path": "/metadata/labels/users", "value": user_name}]
    return jsonify(
        {
            "response": {
                "allowed": True,
```

There are two calls, a validate call and a mutate call.

Validate call receives the validation webhook request and compares the name of the object and the name of the user who sent the request and rejects the request if its the same name.

Mutate call gets the user name and responds with a json patch operation of adding the user name as a label to any request that was raised by anyone.

Close look to the mutate call:

A patch object is a list of patch operations each operation being “add, remove,replace,move,copy or test”

And then specify the path within the json object that needs to be targeted for change.

Then the value that needs to be added, user_name, that’s gonna be the value for that particular label.

```
@app.route("/mutate", methods=["POST"])
def mutate():
    user_name = request.json["request"]["userInfo"]["name"]
    patch = [{"op": "add", "path": "/metadata/labels/users", "value": user_name}]
    return jsonify(
        {
            "response": {
                "allowed": True,
                "uid": request.json["request"]["uid"],
                "patch": base64.b64encode(patch),
                "patchtype": "JSONPatch",
            }
        }
    )
```

This is then sent as a base64 encoded object as part of the response.

Exam note: You won’t be asked to create any code like this.

All you need to know from this is that:
The Admission webhook server is a server that you deploy that contains the logic or the code to permit or reject a request and must be able to receive and respond with the apropiate responses that the webhook spects.

Host the webhook server:
We can run it as a server somewhere or containerized and deployed within k8s cluster itself as a deployment.

If deployed as a deployment in k8s you will need a service to be accessed.

Configuring Admission Webhook:
Configure our cluster to reach out to the service and validate or mutate the request.

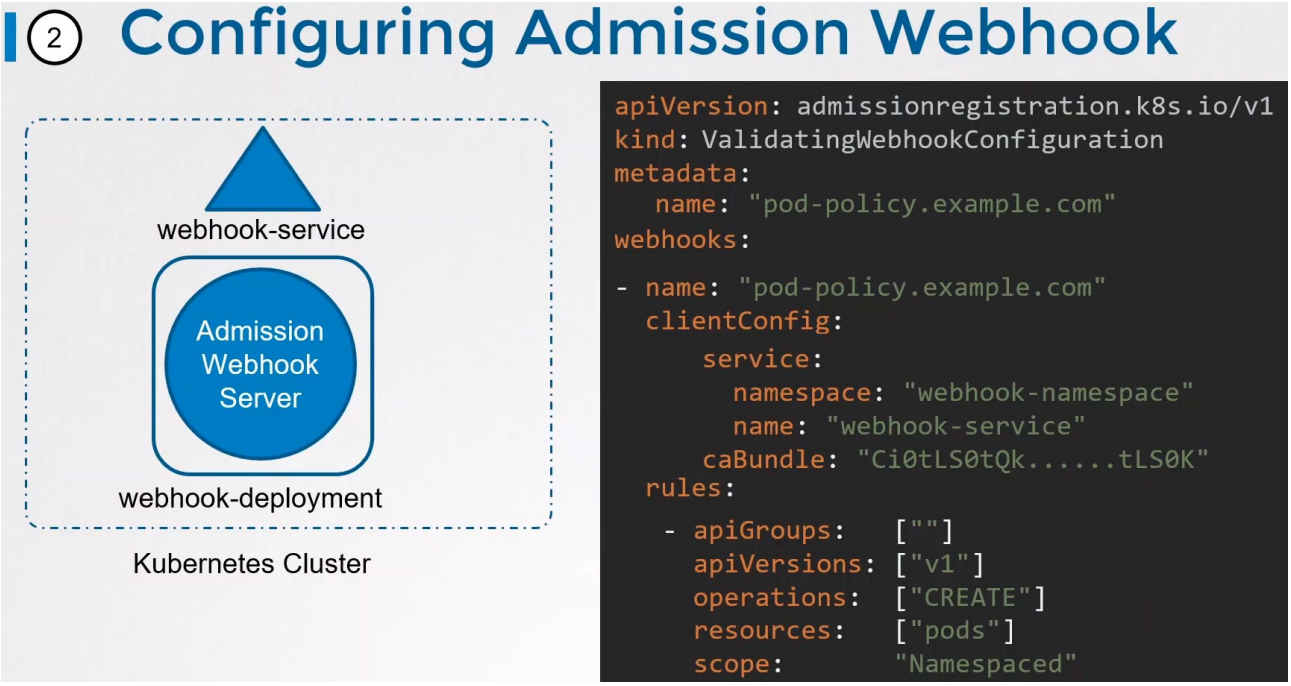
For this we create a validating webhook configuration object.

ClientConfig: Where we configure the location of our webhook server.
If we deploy our server outside the k8s cluster we just provide an url to our server

If we deploy it inside the k8s cluster we have to provide the name of the service and the namespace.
We have to configure a certificate bundle because the communication must be over TLS.

Rules: We must specify when to call our API server. We have to specify the rules to configure exactly when we want our webhook server to be called for validation.

We may not want to do that for all the calls, just while creating pods or deleting pods for example.

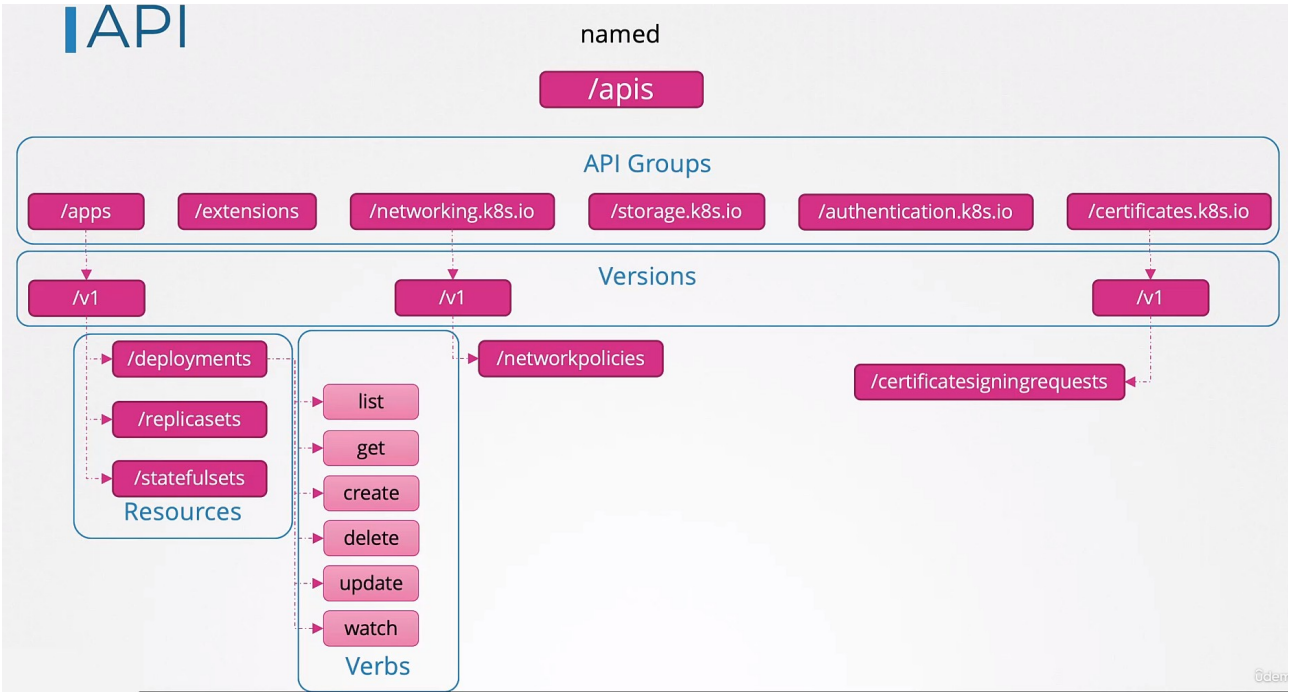


In this example we are only going to call this webhook configuration when calls are made to create pods.

API Versions:

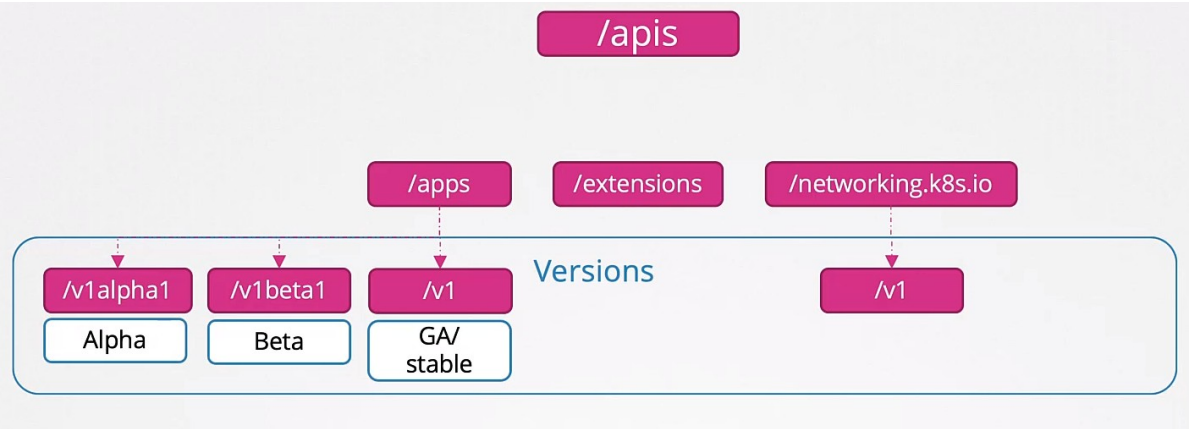
Everything under API are the API Groups like apps, extensions, networking, etc, see image below.

Each API Group have API Versions. [API Groups v1.23]



GA (Generally Available**) Stable version:** Generally Available Stable Version: In the image we see the v1 Version, when an API Group is at v1 that means that is a GA Stable version.d

The API Group may have other versions like Alpha, Beta, as v1alpha1, v1beta1, etc.

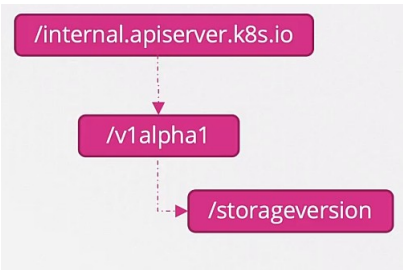


Versions meaning:

Alpha Version: Is when an API is first develop and merged to the k8s codebase and becomes part of the k8s release for the very first time.

At this stage the API Version has Alpha in its name and could be v1alpha1, v1alpha2, etc.

This API Group is **not enabled by default**. For example this resource, called storageVersion is in v1alpha1 only, so you have to call it in the apiVersion of the template.




```
apiVersion: internal.apiserver.k8s.io/v1alpha1
kind: StorageVersion
metadata:
  name: sv-1
spec:
```

However this alpha version is not enabled by default, so if you try to create the object StorageVersion you won't be able to do it, you will need to enable this particular API Version first.

Since this is an alpha version it may lack into end tests and may have bugs and as such is not very reliable.

Also there is no warranty that this API will be available in future releases. It may be dropped without any notice in future releases.

As such this is only really for expert users interested in testing and giving only feedback for the API Group.

Beta Version: After some time the alpha version and once all the major bugs are fixed and it has end to end test the beta version is released.

The version is named v1beta1, v1beta2, etc.

The API Groups in Beta stage are **enabled by default.**

They have e2e tests. It may still have minor bugs. There is commitment from the project on moving it to GA in the future.

Users interested in beta testing and providing feedback.

After being in the beta stage for a few months and a few releases the API Groups moves to the GA or stable version.

GA Stable Version: Generally Available Stable Version

Here there is no longer beta or alpha in the name, just v1, or v2, etc.

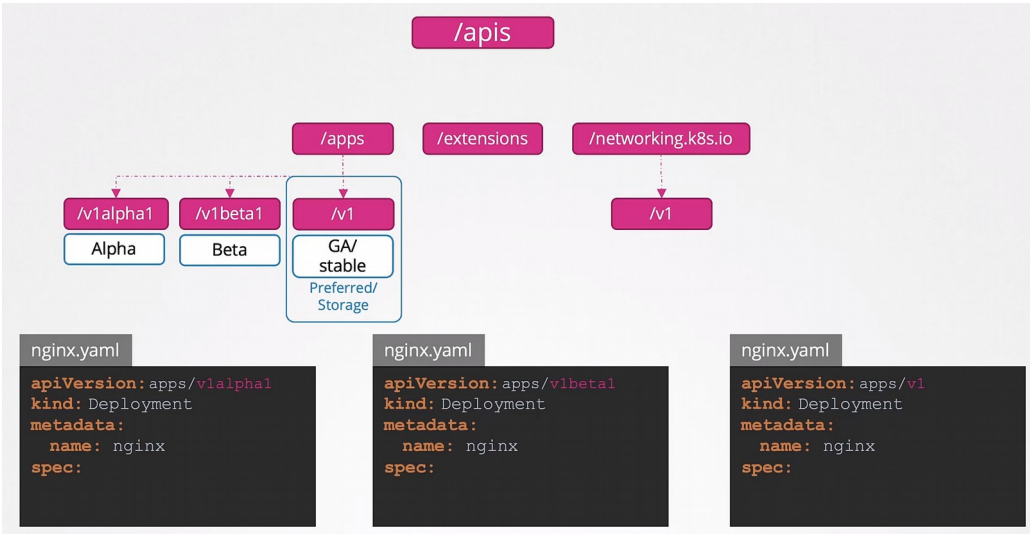
Enabled by default.

All the bugs and tests where done in the alpha and beta stages.

	Alpha	Beta	GA (Stable)
Version Name	vX**alpha**Y (eg: v1**alpha**1)	vX**beta**Y (eg: v1**beta**1)	vX (eg: v1)
Enabled	No. Can enable via flags	Yes by default	Yes by default
Tests	May lack e2e tests	Has e2e tests	Has conformance tests
Reliability	May have bugs	May have minor bugs	Highly reliable
Support	No Commitment. May be dropped later	Commits to complete the feature and move to GA	Will be present in many future releases
Audience	Expert Users interested in giving early feedback	Users interested in beta testing and providing feedback	All users

Most of the API Groups are GA Stable now. However if you take a look to API Groups v1.23 you will see that some of them, like autoscaling api group have v1, v2, v2beta2, v2beta1.

Multiple versions: An API Group can support multiple versions at the same time.
Example: apps API Group have several versions.



This means you would be able to create the same object using any of these versions in your yaml file.

Preferred or Storage version: If you have several versions of an API Group only one can be the preferred one. It can be called Storage version too.

When you have multiple versions enabled and you run k get deployment, you will query the defined one as Preferred/Storage version.

In the above picture v1 is the preferred one.

If you have multiple versions and you use other than the preferred in your yaml manifest, the version will be converted to the preferred one, example:

You have v1alpha1, v1beta1, v1 API Group versions of deployments, and you use v1beta1 in your yaml manifest. K8s will convert v1beta1 to v1 before storing it into the etcd db, because only the preferred/storage version can be used.

Preferred vs Storage versions:

Preferred: Is the default version, used when you retrieve information through the API, like kubectl get command.

Storage: Is the version in which an object is stored in ETCD.

Both are usually the same but can be different.

Get the Preferred version: kubectl explain deployment → It will return the Preferred version.

Or via html:

Get the Storage Version: Query etcd. In this example we can see that is apps/v1

```
ETCDCTL_API=3 etcdctl
  --endpoints=https://[127.0.0.1]:2379
  --cacert=/etc/kubernetes/pki/etcd/ca.crt
  --cert=/etc/kubernetes/pki/etcd/server.crt
  --key=/etc/kubernetes/pki/etcd/server.key
  get "/registry/deployments/default/blue" --print-value-only
```

```
k8s

apps/v1
Deployment


bluedefault"*$cf8dcd55-8819-4be2-85e7-bb71665c2ddf2ZB
successfully progresse8"2
```

Enabling/Disabling API groups: Do it into the kube-apiserver configuration file.

```
ExecStart=/usr/local/bin/kube-apiserver \\
  --advertise-address=${INTERNAL_IP} \\
  --allow-privileged=true \\
  --apiserver-count=3 \\
  --authorization-mode=Node,RBAC \\
  --bind-address=0.0.0.0 \\
  --enable-swagger-ui=true \\
  --etcd-cafile=/var/lib/kubernetes/ca.pem \\
  --etcd-certfile=/var/lib/kubernetes/apiserver-etcd-client.crt \\
  --etcd-keyfile=/var/lib/kubernetes/apiserver-etcd-client.key \\
  --etcd-servers=https://127.0.0.1:2379 \\
  --event-ttl=1h \\
  --kubelet-certificate-authority=/var/lib/kubernetes/ca.pem \\
  --kubelet-client-certificate=/var/lib/kubernetes/apiserver-etcd-client.crt \\
  --kubelet-client-key=/var/lib/kubernetes/apiserver-etcd-client.key \\
  --kubelet-https=true \\
  --runtime-config=batch/v2alpha1\\
  --service-account-key-file=/var/lib/kubernetes/service-account.pem \\
  --service-cluster-ip-range=10.32.0.0/24 \\
  --service-node-port-range=30000-32767 \\
  --client-ca-file=/var/lib/kubernetes/ca.pem \\
  --tls-cert-file=/var/lib/kubernetes/apiserver.crt \\
  --tls-private-key-file=/var/lib/kubernetes/apiserver.key \\
  --v=2
```

You can add more by using commas.

Remember to restart the kube-apiserver service if you are not running on a kubeadm cluster.

API Deprecations:

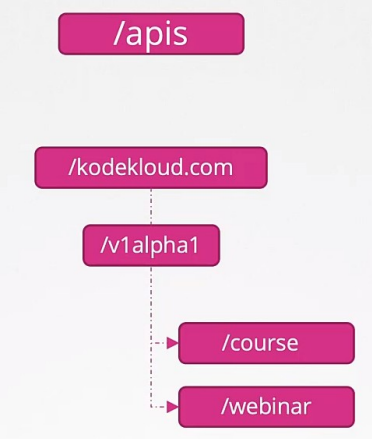
API Deprecations Policy will answer:

- Why do you need to support multiple API Groups versions?
- How many should you support?
- When can you remove an older version that is not longer required.

Example: We plan to contribute to the k8s API project and we create an API Group called kodekloud.com, under which we have two resources called course and webminar.

We develop it in house and be tested and when we are ready to merge it to the k8s project we raise a PR (pull request), then gets accepted and we release an alpha version.

We've called the version: v1alpha1



You can create a course or webminar object by using a yaml file like this one and using kodekloud.com/v1alpha1

ckad-course.yaml

```yaml
apiVersion: kodekloud.com/v1alpha1
kind: Course
metadata:
  name: ckad
spec:
```

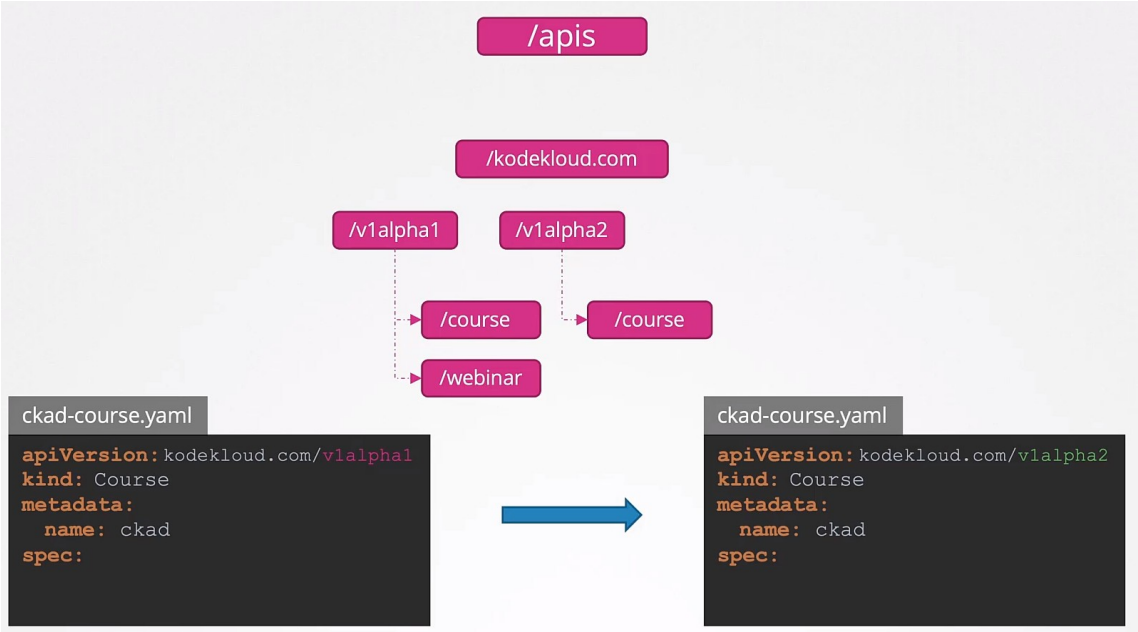
Let's say that the webminar object didn't go well with the users and we decided to removed it.

But in the next k8s relase we can NOT remove it because is breaking the 1st rule of api deprecation policies.

API Deprecation Policy Rules:

1st Rule: API elements may only be removed by incrementing the version of the API group.

Meaning: You can only remove the webminar element from the v1alpha**2** version of the API Group.



It will continue to be present in the v1alpha1 version of the element.

Change of versions:
At this point the resource in etcd db is still at v1alpha1 but our API version have changed to v1alpha2

This is going to be a problem. We will need to go back and change all the API versions in our yaml files from v1alpha1 to v1alpha2.

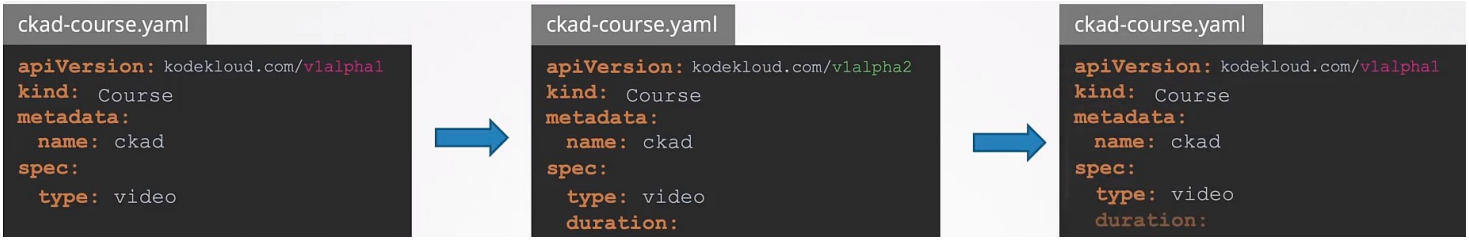
Which is why the release must support both versions.

But the Preferred or Storage versions could be v1alpha2. This means that the users can use the same yaml files to create the resources but internally it will be converted and stored as v1alpha2

2nd Rule: API objects must be able to round-trip between API versions in a given release without information loss, with exception of whole REST resources that do not exist in some versions.

If we create an object as v1alpha1 and it is converted to v1alpha2, and then converted back to v1alpha1 it should be the same as the original v1alpha1 version.

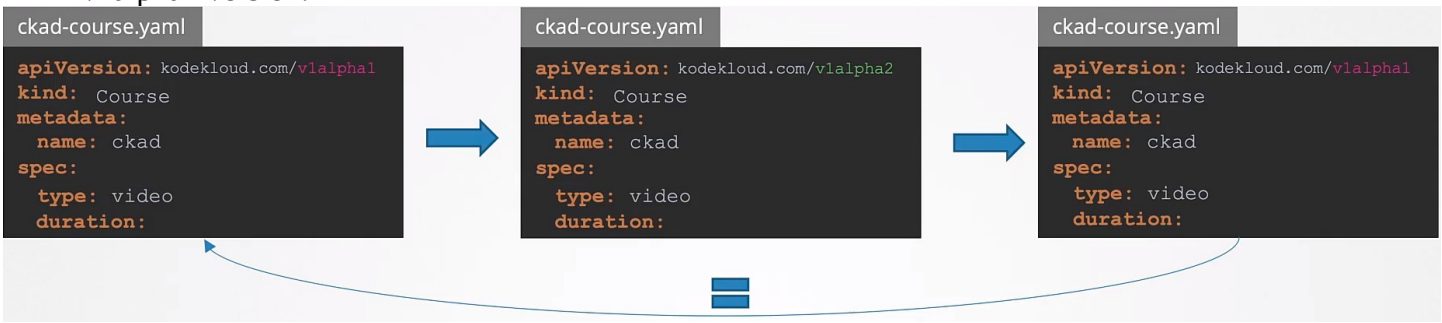
Example:
We have a course object in v1alpha1, it has a spec field called type set to video.



Then we convert it to v1alpha2 and we add a new field called duration, which was not there in the previous version.

When we convert this back to v1alpha1 it will now have the new field, but the original one didn't.

So we must add that equal and field in v1alpha1 so the converted v1alpha1 matches the original v1alpha1 version.



To continue with our story we fixed some more bugs and we are ready for beta.

Our first beta version is now ready, v1beta1. After a few months we released v1beta2 and finally we released a stable (GE) version, v1.

We don't have to have all the older versions. Let's see the rules and best practices around that.

4a Rule: Yes, there is no mistake, rule 3 is still not explained

Other than the most recent API versions in each track, older API versions must be supported after their announced deprecation for a duration of no less than:

- GA: 12 months or 3 releases (whichever is longer)
- Beta: 9 months or 3 releases (whichever is longer)
- Alpha: 0 releases

4b Rule: The Preferred API version and the Storage version for a given group may not advance until after a release has been made that supports both the new version and the previous version.

3rd Rule: An API version in a given track may not be deprecated until a new API version at least as stable is released.

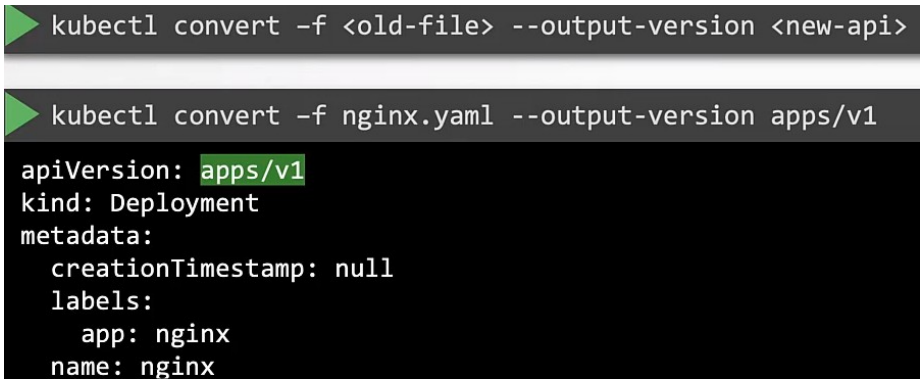
Meaning: GA (Generally Available) versions can deprecate beta or alpha versions but not all the way around, an alpha version can not deprecate a GA version. Only a GA Stable version can deprecate another.

Kubectl Convert:
As we have been seen when k8s clusters are upgraded we have new API been added and old ones being deprecated and removed.

When an old API is removed you have to updated your manifest files to the new version, example:

We may have a lot of yaml files pointing to the old version.

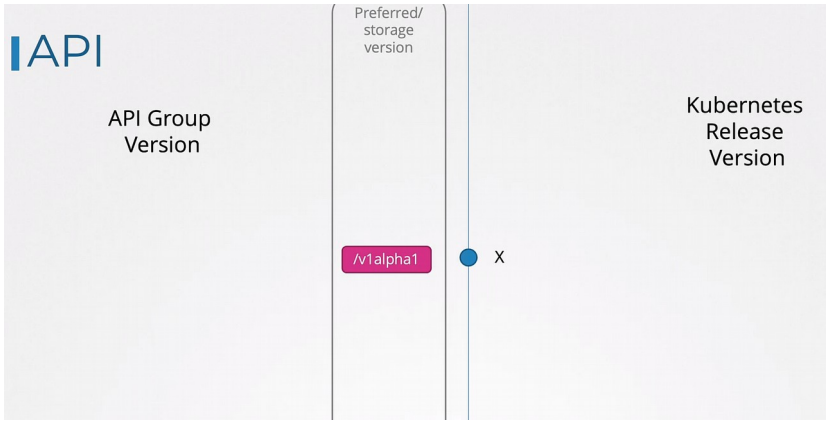
You can use kubectl convert command to do so.



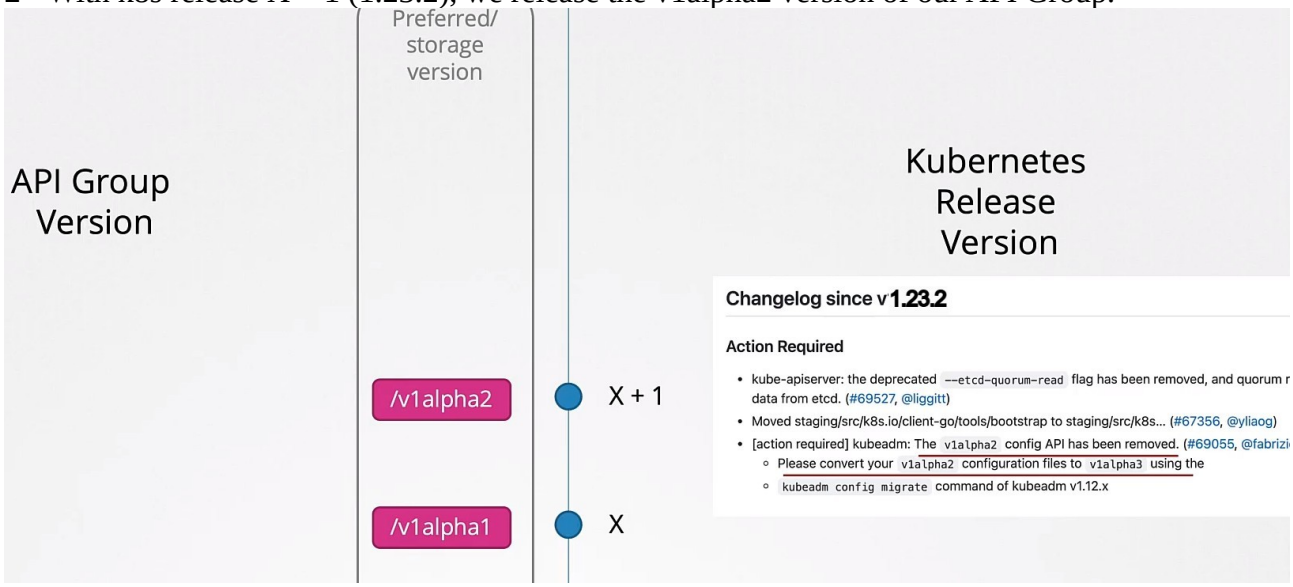
How releases are moved and deprecated:

1 - We start with the API Group version v1alpha1.
Kubernetes Release Version could be 1.23.1

Since this is the only version it is the Preferred/Storage version



2 - With k8s release X + 1 (1.23.2), we release the v1alpha2 version of our API Group.



v1alpha1 will be removed from the k8s release X + 1 following the 4a Rule, which says that old alpha versions will be keep 0 releases once there is a superior version, like in this example with v1alpha2.

And you have to mention about v1alpha1 being removed in the release notes of that k8s version

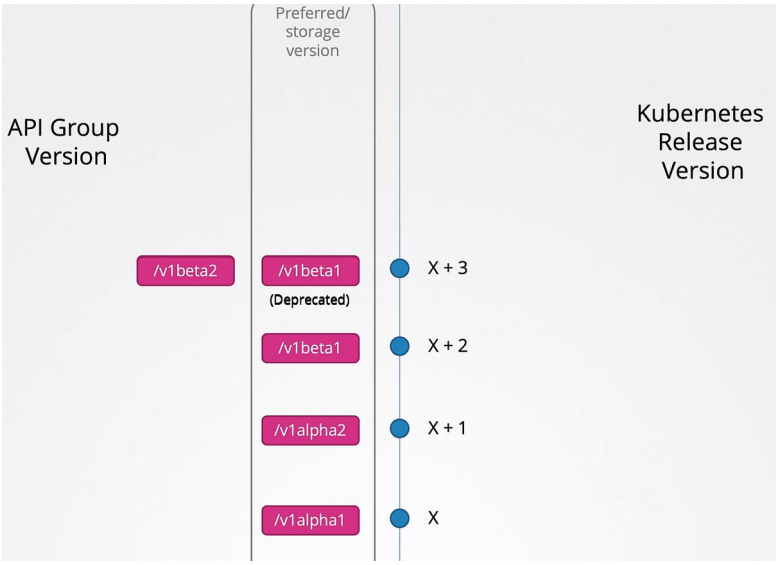
3 - Now the X + 3 k8s version we release v1beta1 API Group version. V1alpha2 won't be required to be in this release.

So you have to inform users in the release notes about removing v1alpha2 and to migrate all the needed yamls to v1beta1

Have in mind that rule 4a asks to keep beta versions for 3 releases or 9 months.

4 - In X + 3 release we have v1beta2 release. As rule 4a says we have to keep the previous beta (v1beta1) for 9 months or 3 releases.

So this X + 3 release will have both, v1beta1 and v1beta2



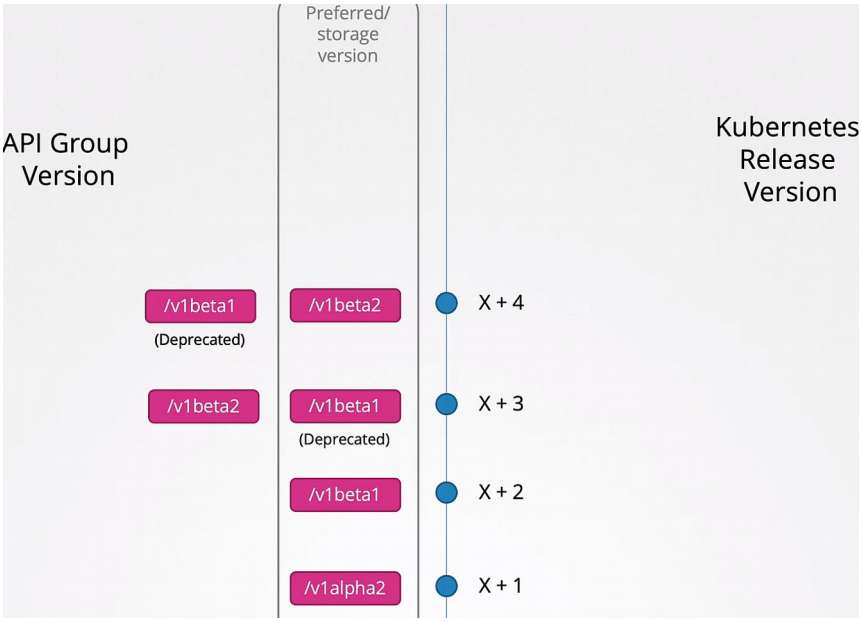
v1beta1 is deprecated but not removed. If you use v1beta1 it will show a deprecation warning.

Also note that v1beta1 is continuing to be the Preferred version.

Why is that, instead of having v1beta2 as preferred we have v1beta1?

Because Rule 4b, asks to have the old beta released and the new one. (?)
Is in the next release that the new beta version will be used as Preferred.

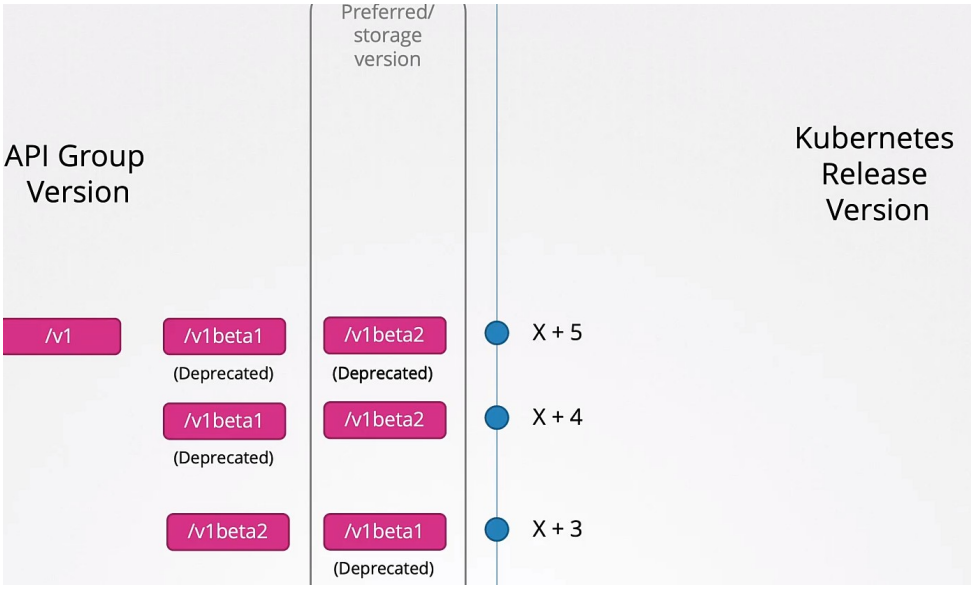
5 - X + 4 have both releases but v1beta2 is now the preferred version.



6 - The next release, X + 5, will have the first GA (Generally Available) Stable v1 version.

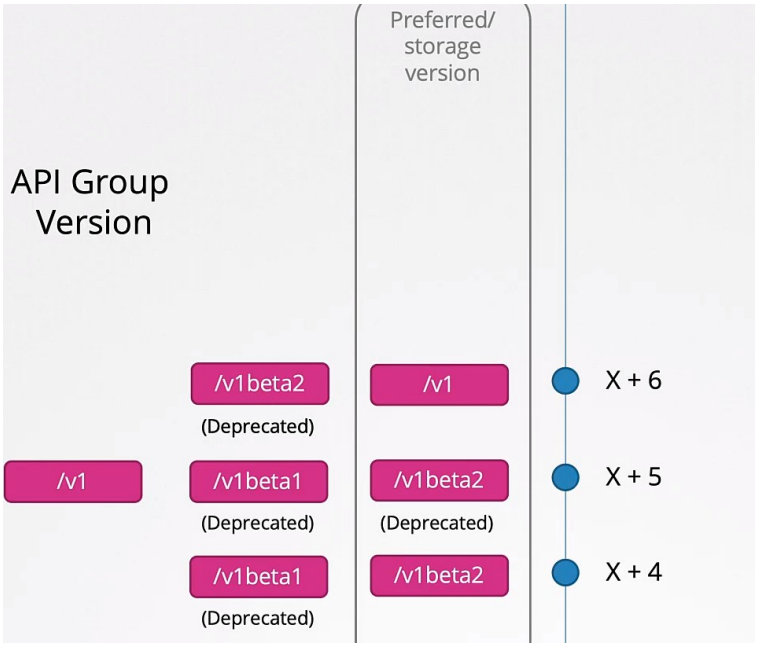
v1beta1 and v1beta2 are still there, now both become deprecated.

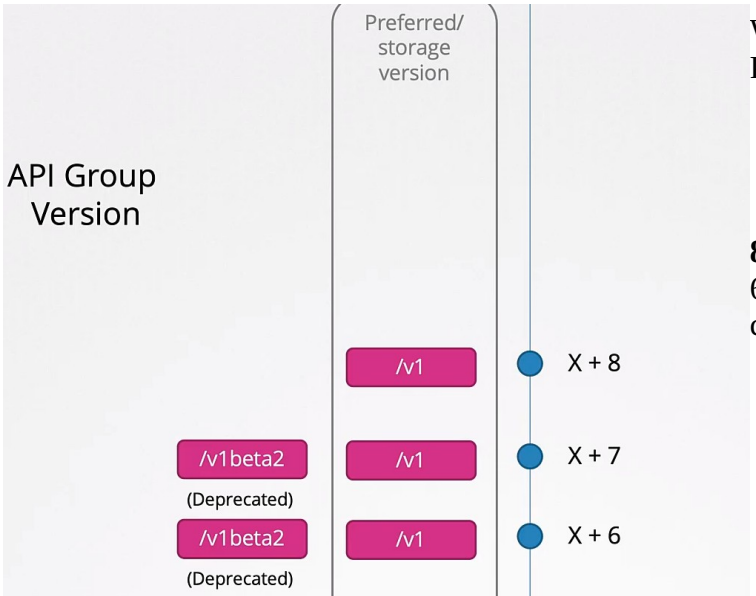
v1beta2 is still the preferred one because there is only one version of the GA Stable one (v1)



7 - In the next release (X + 6) we can remove v1beta1 because as per Rule 4a we have to have it during 3 releases and this have been achieved.

V1beta1 has been there since X + 2, deprecated in X + 3 and its being around in X + 4 and X + 5

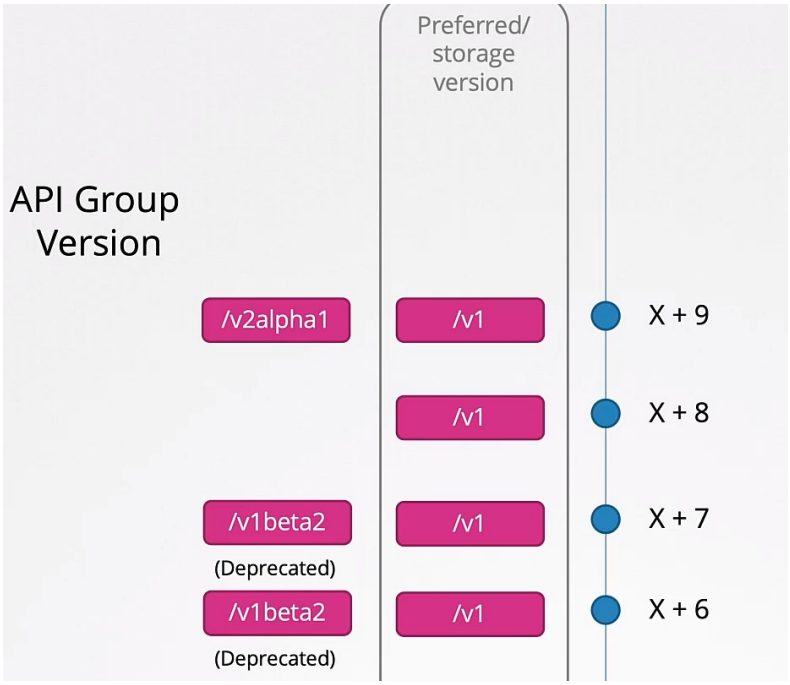




With X + 6 we can move v1 to the Preferred/storage version

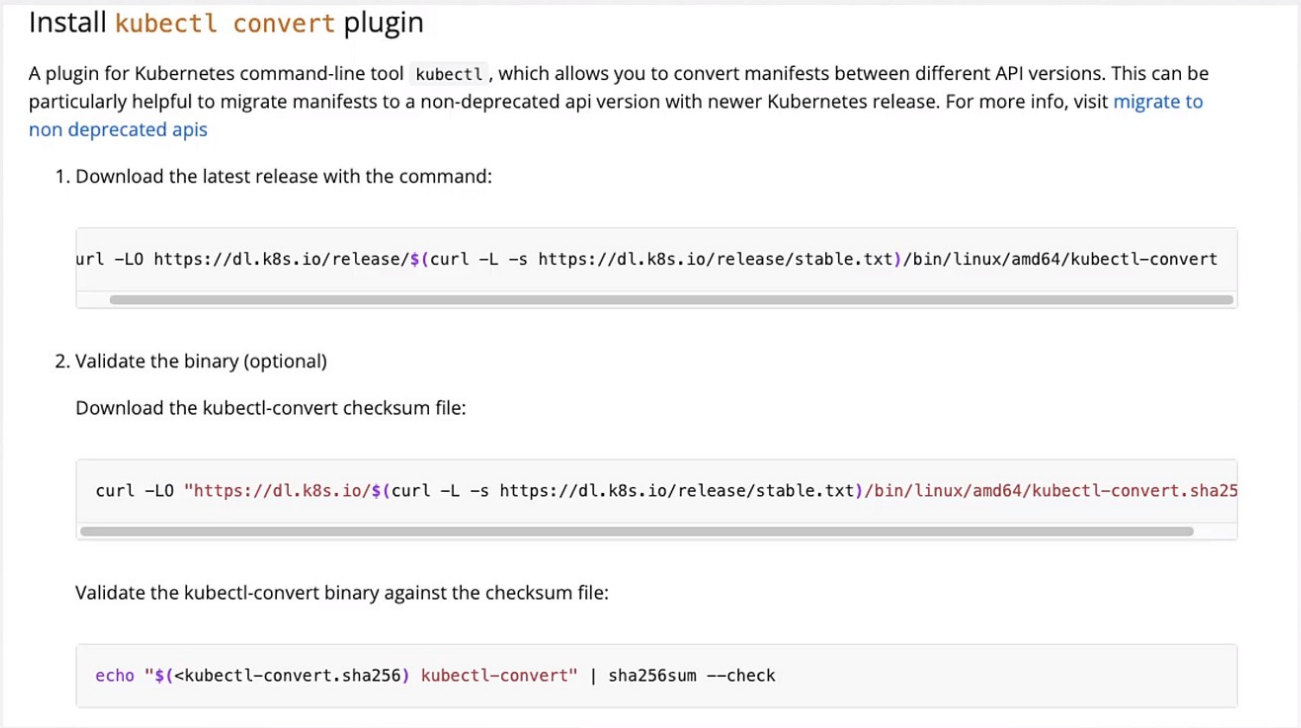
8 - X + 7 and X will be the same than X + 6. X + 8 will remove v1beta2 after completing its 4 releases cycle,

9 - X + 9 have a new alpha version but following rule 3 we can NOT deprecate a GA (Generally Available) Stable version with an alpha or beta, only another GA Stable version can deprecate another GA Stable version.



kubectl convert may not be in your k8s cluster because is a separated plugin, you will need to install it:

Installing Kubectl Convert



Install `kubectl convert` plugin

A plugin for Kubernetes command-line tool `kubectl`, which allows you to convert manifests between different API versions. This can be particularly helpful to migrate manifests to a non-deprecated api version with newer Kubernetes release. For more info, visit migrate to non deprecated apis

1. Download the latest release with the command:

```
url -LO https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl-convert
```

2. Validate the binary (optional)

   Download the kubectl-convert checksum file:

```
curl -LO "https://dl.k8s.io/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl-convert.sha25
```

   Validate the kubectl-convert binary against the checksum file:

```
echo "$(<kubectl-convert.sha256) kubectl-convert" | sha256sum --check
```

https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/#install-kubectl-convert-plugin

Commands:

-- **Identify the short names of the resources:** kubectl api-resources

-- **Get the patch version:** X. Y. Z (1.22.3)
X → Major version (1)
Y → Minor version (22)
Z → Patch version (3)

-- **Get the API Group** of a resource called Job is part of:

kubectl explain job

Look for VERSION → you will see: batch/v1
batch is the API Group
v1 is the version

-- **Get the preferred version** for authorization.k8s.io
Run → kubectl proxy → leave it running in the terminal
kubectl proxy 8001& → this runs it in the background, theoretically

Open another terminal and run:
curl http://127.0.0.1:8001/apis/authorization.k8s.io
curl localhost:8001/apis/authorization.k8s.io → this works the same

-- Enable v1alpha1 version for rbac.authorization.k8s.io
Do a backup of kube-apiserver.yaml first

Then add following line:
- --runtime-config=rbac.authorization.k8s.io/v1alpha1

Add it before the etcd lines.

Once saved check that the kube-apiserver pod is in running state, check if you can list nodes, k get nodes.

Will take a few minutes to have back the cluster.

-- Install and convert one old version to v1:
Install kubectl convert

kubectl-convert -f ingress-old.yaml --output-version networking.k8s.io/v1 > ingress.yaml

kubectl get ing ingress-space -oyaml | grep apiVersion
apiVersion: networking.k8s.io/v1
  - apiVersion: networking.k8s.io/v1

Custom Resource Definitions, CRD:

Resource: Like deployments, ReplicaSet, Job, Cronjob, Statefulset, Namespace, etc.

When you create a deployment, k8s creates a deployment and stores its information in the ETCD datastore.

We can create, list, delete the deployment. All of this will create, list and modify the deployment object, or Resource, in the ETCD datastore.

When we create deployments it create pods equal to the number of replicas defined in the deployment.

Who is responsible for that. Thats the job of a Controller, in this case a Deployment Controller.

Controller: Is any process or code that runs in the background in a loop and its job is to continuously monitor resources that is supposed to manage, in the above example the Deployment.

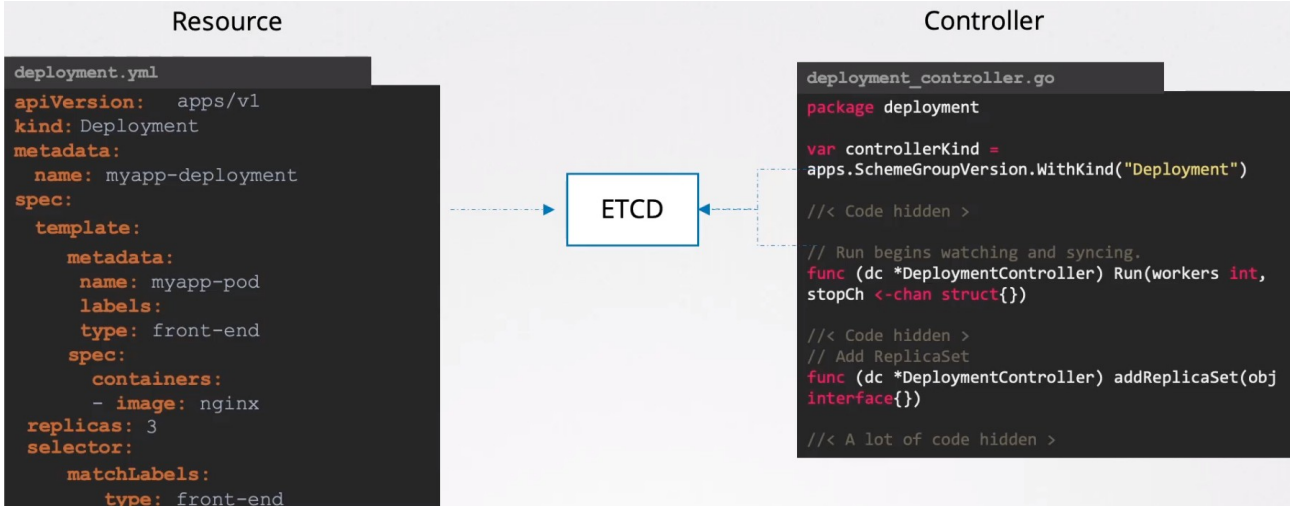
When we create, update or delete the deployment, the Controller makes the necessary changes on the cluster to match what we have done.

In this case when we create a deployment object the controller creates a replica set which in turn creates as many pods as we have specified.

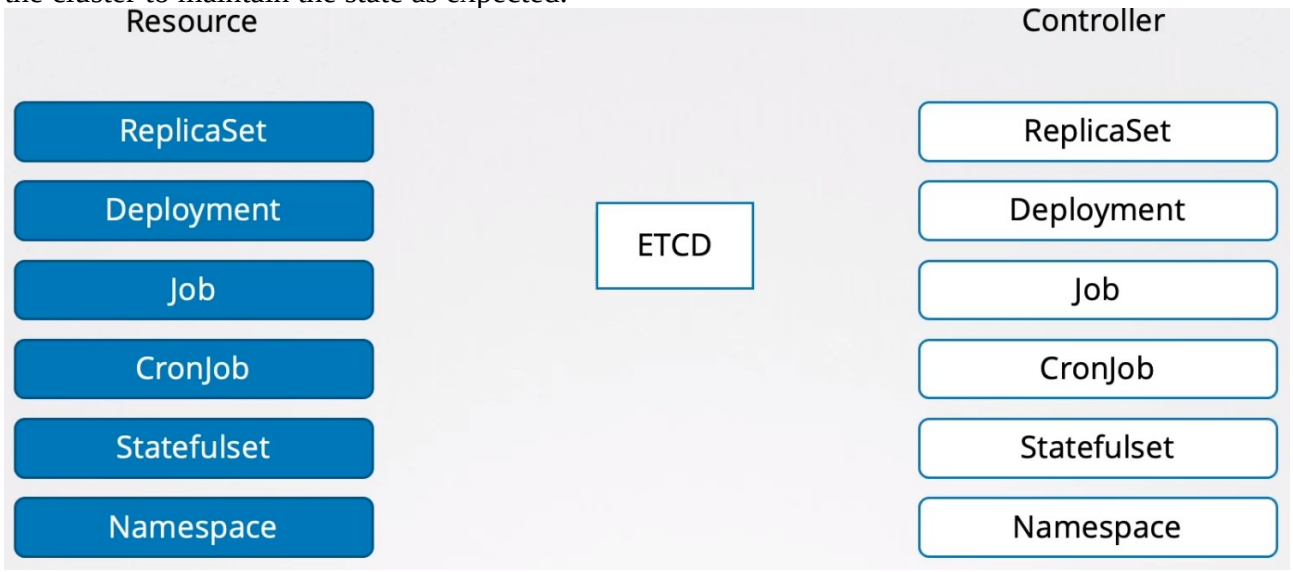
That’s the job of a Controller

You don’t have to create any controller because are already built in into the k8s cluster.

Here is how a Controller looks like. Controllers are written in Go.



The resources below are just an example of the many that k8s have. Each Resource have a Controller responsible of watching the status of the resources and make the necessary changes on the cluster to maintain the state as expected.



Create your own Resource: Let’s create an object to book flight tickets as example.
This example shows you that the object could be any thing.

What do we want: Create an object called FlightTicket in which you specify the origin and destination of the flight, the number of tickets, etc.

We want to be able to create that resource. We want to be able to list it and delete it if necessary.

This is going to create or delete the object into the ETCD datastore.

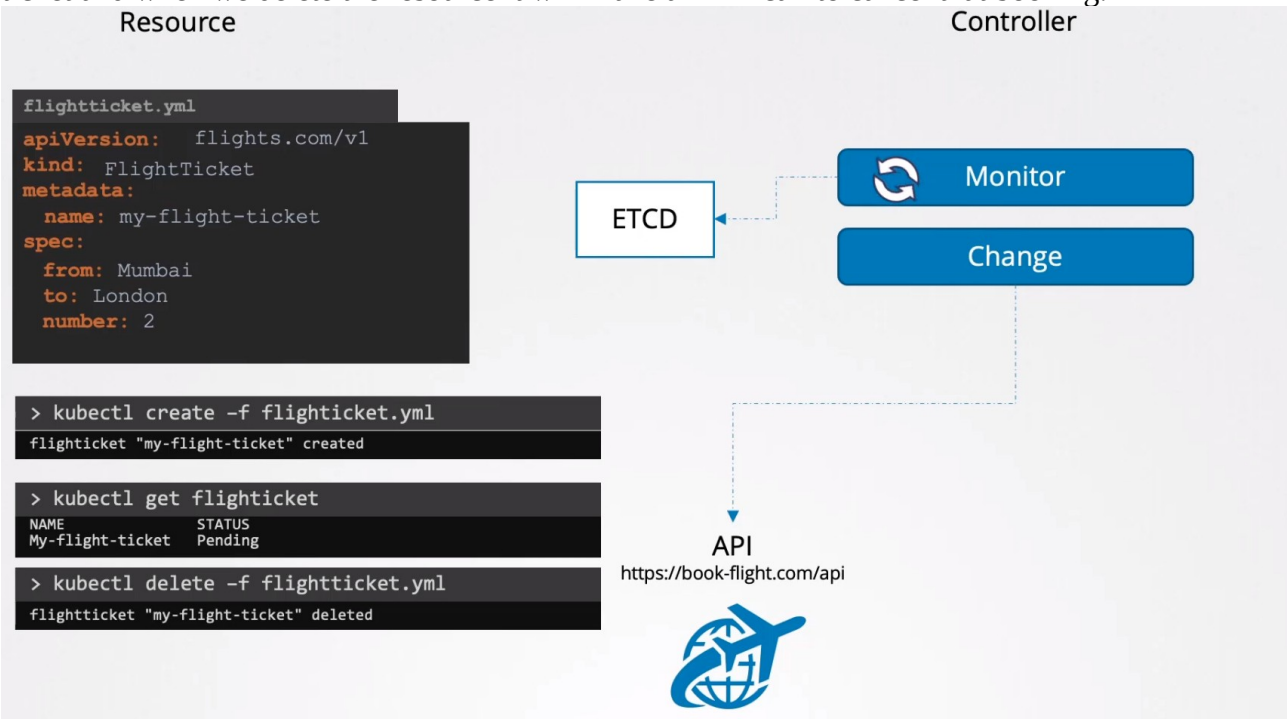
But is not going to actually book a flight ticket.
What we want is to go outside the cluster and book a flight ticket for real.
Say for instance, there is this API https://book-flight.com/api that we can call to book a flight ticket.

How we do call this API whenever we create a FlightTicket object to book a flight ticket?

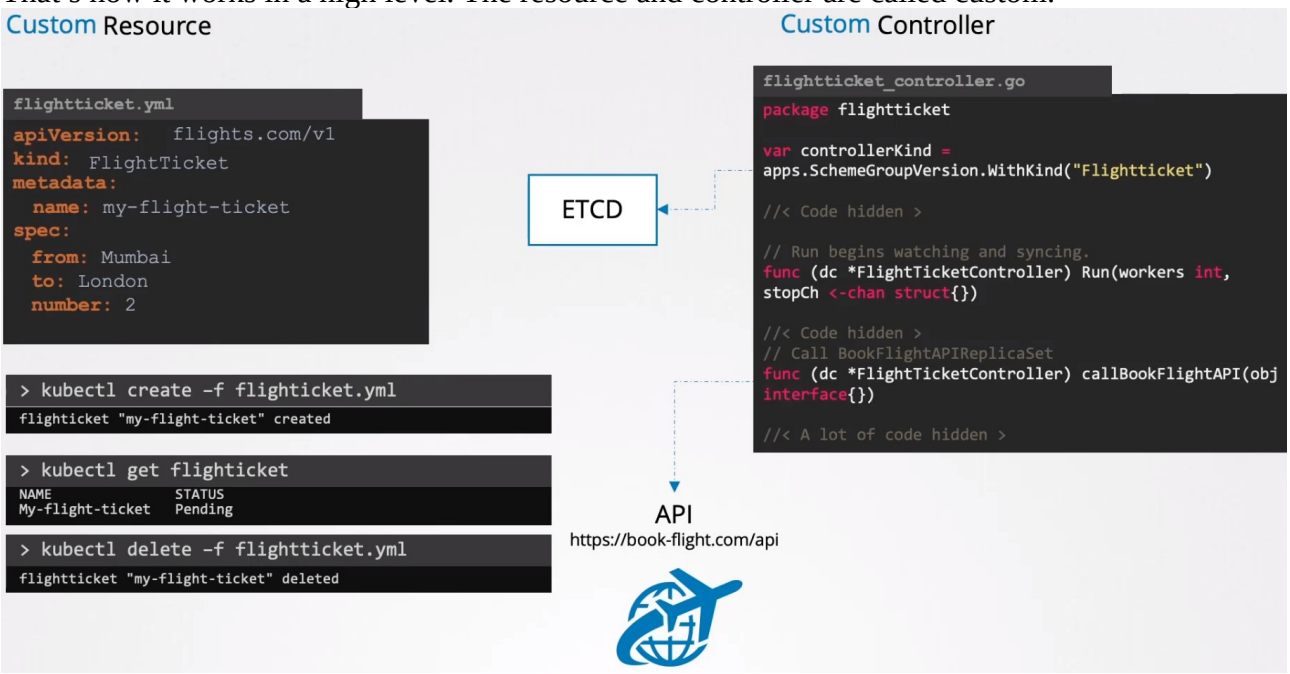
For that we need a controller.

So we will create a flight ticket Controller, written in Go and we will watch for the creation, addition or deletion of the flight ticket resource.

And when we create the resource it will contact with the book-flight ticket API, to book a flight ticket and when we delete the resource it will make an API call to cancel that booking.



That's how it works in a high level. The resource and controller are called custom:



How we are going to do it: If we create the Custom Resource and the Custom Controller is NOT going to work because we need first to configure it in the k8s API.

We do so by creating a **CRD**, **Custom Resource Definition**.

We are going to use CRD to tell k8s that we would like to create objects of kind FlightTicket

Custom Resource:

flightticket.yml

```yaml
apiVersion: flights.com/v1
kind: FlightTicket
metadata:
  name: my-flight-ticket
spec:
  from: Mumbai
  to: London
  number: 2
```

CRD: Based on the above Custom Resource.

flightticket-custom-definition.yml

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: flighttickets.flights.com
spec:
  scope: Namespaced
  group: flights.com
  names:
    kind: FlightTicket
    singular: flightticket
    plural: flighttickets
    shortnames:
      - ft
  versions:
    - name: v1
      served: true
      storage: true

  schema:
    openAPIV3Schema:
```

spec.scope: Namespaced or NonNamespaced (clusterwide)

spec.group: The apiVersion of the above custom resource, flights.com in this example

spec.names.kind: Names of the resource, from kind in the Custom Resource, FlightTicket in this example.

spec.names.singular: flightticket, used when you type: kubectl get flightticket

spec.names.plural: flighttickets → used by the k8s API resource. If you run *kubectl api-resources* is what is going to be shown. Name to be used in the URL: /apis/<group>/<version>/<plural>

spec.names.shortnames: ft → this way you can refer to it when you call it with kubectl. kubectl get ft

spec.versions: Each resource can be configured with multiple versions as it passes through the various phases of its lifecycle. Example if its a new resource we can provide the v1alpha1 and move it until we get the version v1.

spec.schema: Defines all the parameters that can be specified under the spec section when you create the object. from, to and number in this example.

```
schema:
  openAPIV3Schema:
      type: object
      properties:
        spec:
          type: object
          properties:
            from:
              type: string
            to:
              type: string
            number:
              type: integer
              minimum: 1
              maximum: 10
```

Values minimum and maximum:
We can specify under prototipes.number the range of numbers that the user can introduce, 1-10 in this example. If the user adds less than 1 or more than 10 will give you an error message

We can create the CRD:

```
> kubectl create -f flightticket-custom-definition.yml
customresourcedefinition "FlightTicket" created
```

Now you can create the Custom Resource:

```
> kubectl create -f flighticket.yml
flighticket "my-flight-ticket" created

> kubectl get flighticket
NAME              STATUS
My-flight-ticket  Pending

> kubectl delete -f flightticket.yml
flightticket "my-flight-ticket" deleted
```

All we did until now will create the object and store it in ETCD but is not going to do any outside API call, to do so we have to create the **Custom Controller**

Custom Controllers:

Create the Custom Controller:

What we need to do is to monitor the status of the objects in ETCD and perform actions such as making calls to the book-flight.com/api to book, edit or cancel flights tickets. That's why we need a custom controller.

Controller: Is any process or code that runs in a loop and is continuously monitoring ETCD and listening to events of specific objects being changed, in this case the FlightTicket object, and now we can do that in any way we can.

Write the code: You could write the code in Python but may be challenging as the calls made to the API may be expensive and we will need to create our own queuing and caching mechanisms.

Developing in GO: Provides support for other libraries like caching and queuing mechanisms that can help to build controllers easily and in the right way.

flightticket_controller.go

```
package flightticket

var controllerKind =
apps.SchemeGroupVersion.WithKind("Flightticket")

//< Code hidden >

// Run begins watching and syncing.
func (dc *FlightTicketController) Run(workers int,
stopCh <-chan struct{})

//< Code hidden >
// Call BookFlightAPIReplicaSet
func (dc *FlightTicketController) callBookFlightAPI(obj
interface{})

//< A lot of code hidden >
```

You can start with [sample-controller repo](). Clone it and modify controller.go with your custom code, then build it and run it.

1. Make sure you have installed GO in your machine.
2. Clone the repo, cd sample-controller
3. Customize your code. Somewhere in the code we will make the call to book-flight.com/api
4. Build it
5. We run the code specifying the kubeconfig that the controller can use to authenticate to the k8s API
6. The control process starts locally, watching for the creation of FlightTicket objects and then making the necessary calls

```
> go
Go is a tool for managing Go source code.

go <command> [arguments]
```

```
> git clone https://github.com/kubernetes/sample-controller.git
Cloning into 'sample-controller'...

Resolving deltas: 100% (15787/15787), done.
```

```
> cd sample-controller
```

Customize controller.go with our custom logic

```
> go build -o sample-controller .
go: downloading k8s.io/client-go v0.0.0-20211001003700-dbfa30b9d908

go: downloading golang.org/x/text v0.3.6
```

```
> ./sample-controller -kubeconfig=$HOME/.kube/config
I1013 02:11:07.489479   40117 controller.go:115] Setting up event handlers

I1013 02:11:07.489701   40117 controller.go:156] Starting FlightTicket
controller
```

controller.go

```
package flightticket

var controllerKind =
apps.SchemeGroupVersion.WithKind("Flightticket")

//< Code hidden >

// Run begins watching and syncing.
func (dc *FlightTicketController) Run(workers int,
stopCh <-chan struct{})

//< Code hidden >
// Call BookFlightAPIReplicaSet
func (dc *FlightTicketController) callBookFlightAPI(ob
interface{})

//< A lot of code hidden >
```

Once the controller is ready, you can decide how to distributed.

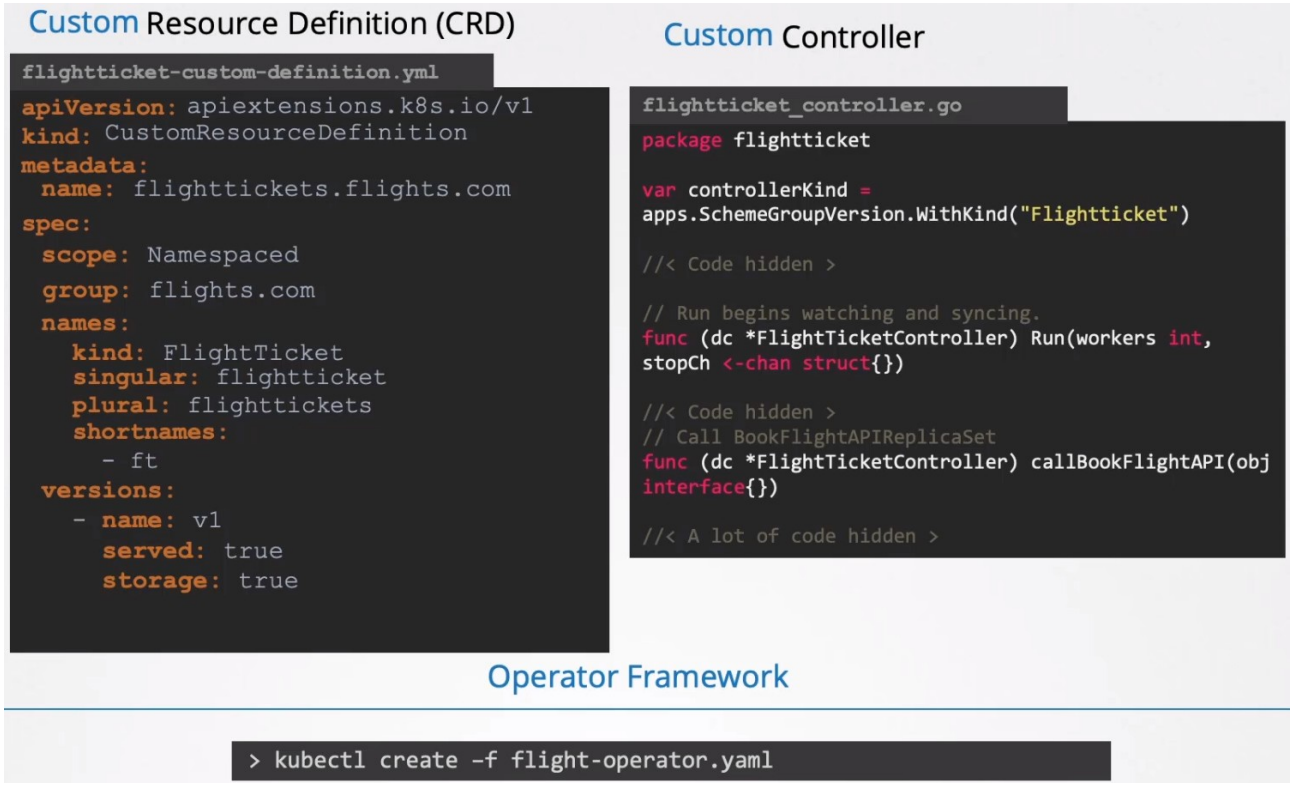
You don’t want to build and run it each time so you can package it into a Docker image and then choose to run it as a pod or deployment in your k8s cluster.

For the exam:

You don’t spect to have to build a Custom Controller. However there may be questions to build Custom Resources Definitions and work with CRD files or to work with existing Controllers.

Operator Framework:

Until now we have to create the CRD and the Custom Controller separately, but with **Operator Framework** we can package them together to be deployed as a single entity.



By deploying the flight-operator.yaml, internally creates the CRD and the resources and also deploys the Custom Controller as a Deployment.

Operator Framework does much more than deploying those two components.

Real life use case: ETCD Operator (one of the most popular)

Used to deploy and manage an ETCD cluster within k8s.

What can you do:

- It has a etcdCluster CRD and an etcd Controller that watches for the etcd cluster resource and deploys etcd within the k8s cluster.

- Take a backup of the etcd cluster.
- Restore a backup to the etcd cluster by simply creating a crd.

K8s Operators do what a human operator typically would do to manage specific operations, such as installing it, maintaining it, take/restore backups, etc.

All operators are available at the operator hub. Where you can find a lot of operators to manage Graphana, Gitlab, Flux, Istio, Jenkins,etcd, and many more.

For the exam: This won't we included in the exam, this is only for learning purposes.

Blue/Green Deployments:

Blue/Green Deployment strategy: The NEW version (green) of the deployment is already deployed alongside with the old version (blue).

100% of the traffic is still routed to the old version.

At this point on time tests are run in the new version.

When all tests are ok we switch traffic to the new version at once.

Istio is good to manage all this.

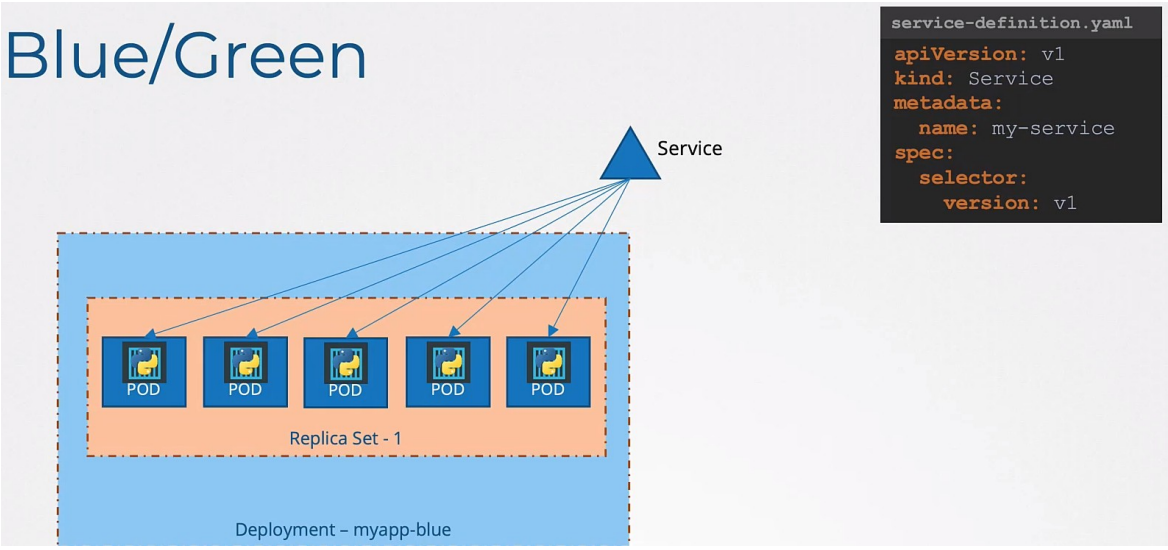
How to implement it: Using label selectors between the service and the deployments.

Blue deployment have the label: version= v1
Green deployment have the label: version=v2

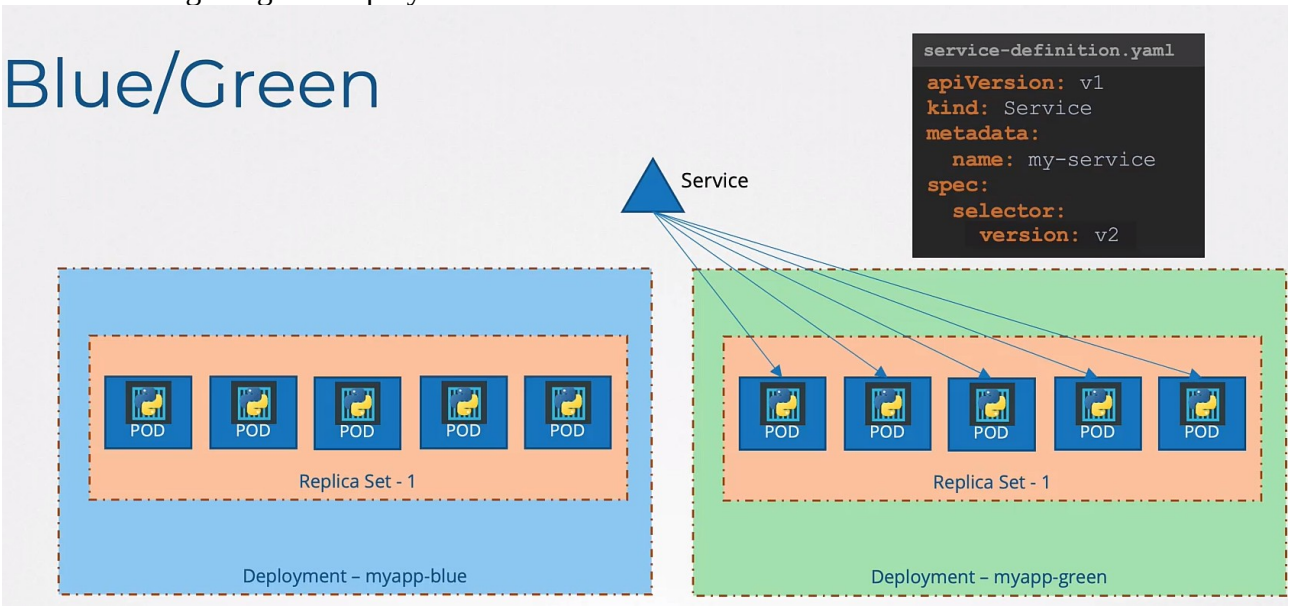
The service exposed to Blue have the label selector: version=1

To switch from blue to green just remove the label selector pointing to blue in the service and add the label of the green deployment.

Before the change in the service



After the change to green deployment in the service:



Canary Updates:

How it works: We deploy the new version and route a small percentage to it, the rest goes to the old version.

If after all tests all looks good we upgrade the old deployment with the new version of the code. Like a rolling update strategy for example.

Then we get rid of the canary deployment.

How to implement it:
We have a deployment with the old code version: v1 and 5 replicas.

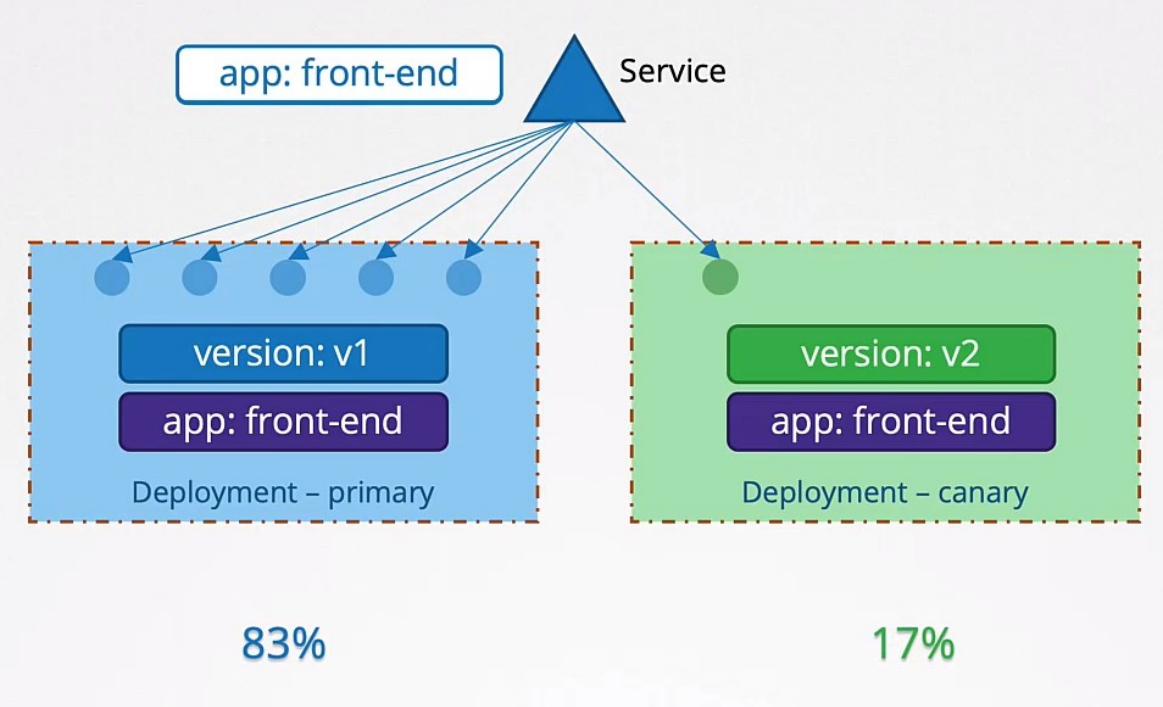
Then we create a service to route traffic to it by associating the label.

We then deploy a second deployment (canary) with the new version of the application version: v2

We want traffic to go both versions at the same time.
However we want only to route a small percentage of traffic to the new version deployment (canary)

How to route traffic to both deployments: From a single service.
We create a common label, called app: front-end, and we apply it in both deployments and the service.

How to reduce the traffic to the canary deployment:
We can do that by reducing the number of pods in the deployment.



Once we see all ok we can upgrade the primary deployment with the new version and then remove the canary deployment.

K8s limitations: With the default tools in k8s we can not say, send to the canary deployment the 1% of the traffic without having to spin up 100 pods.

But with service meshes like Istio you can achieve this and much more things network related.

myapp-primary.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-primary
  labels:
     app: myapp
     type: front-end
spec:
  template:
    metadata:
     name: myapp-pod
     labels:
        version: v1
        app: front-end
    spec:
      containers:
      - name: app-container
        image: myapp-image:1.0
  replicas: 5
  selector:
    matchLabels:
       type: front-end
```

service-definition.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: front-end
```

myapp-canary.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-canary
  labels:
     app: myapp
     type: front-end
spec:
  template:
    metadata:
     name: myapp-pod
     labels:
        version: v2
        app: front-end
    spec:
      containers:
      - name: app-container
        image: myapp-image:2.0
  replicas: 1
  selector:
    matchLabels:
       type: front-end
```

Helm:

Helm: Package manager for k8s.

Makes much more easy to deploy resources. Deploys all the resources at once and already configured to interact between them, you only deploy the big file and that's it.

The same to uninstall the app.

Is like to intall a videogame. You run an installer, you are not installing every part by separate. Parts like the sounds, the textures, the logic of the program, the graphics, etc, etc.

So helm does the same but for k8s resources and apps.

How it works:
helm install wordpress → and helm will install all the parts needed to run wordpress. Parts like services, deployments, pvc, secrets, cm, etc.

Customize the settings: values.yaml

For our app or package by specifying desired values at install time.

But instead of having to edit multiple files in multiple yaml files we have a single location where we can declare every custom setting.

In values.yaml is where we will specify the size of our pv, choose a name for our wordpress website, the admin password, settings to the db, etc.

Commands:

helm install wordpress
helm help
helm upgrade wordpress
helm rollback wordpress
helm uninstall wordpress
helm list
helm search hub wordpress → hub stands for artifact hub, the default repo
helm repo add bitnami https://charts.bitnami.com/bitnami
helm repo list
helm pull --untar bitnami/wordpress → download the chart but don't install it
helm search repo joomla

Install helm:

Prerequisits: k8s cluster, kubectl with proper credentials and login details set up in kubeconfig file to work with k8s cluster.

```
curl https://baltocdn.com/helm/signing.asc | sudo apt-key add -
sudo apt-get install apt-transport-https --yes
echo "deb https://baltocdn.com/helm/stable/debian/ all main" | sudo tee /etc/apt/sources.list.d/helm-stable-debian.list
sudo apt-get update
sudo apt-get install helm
```

Helm Concepts:

Modify Values: values.yaml

Use variables in the yaml objects. Its values are inside values.yaml

Declare the variable by using {{ }}
Fetch the variable value using variable names: {{ .Values.storage}}

templates/deployment.yaml

```
piVersion: apps/v1
ind: Deployment
etadata:
 name: wordpress
 labels:
   app: wordpress
pec:
 selector:
   matchLabels:
     app: wordpress
     tier: frontend
 strategy:
   type: Recreate
 template:
   metadata:
     labels:
       app: wordpress
       tier: frontend
   spec:
     containers:
     - image: {{ .Values.image }}
       name: wordpress
```

templates/pv.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: wordpress-pv
spec:
  capacity:
    storage: {{ .Values.storage }}
  accessModes:
    - ReadWriteOnce
  gcePersistentDisk:
    pdName: wordpress-2
    fsType: ext4
```

templates/service.yaml

```
v1
ce
dpress
dpress
80
dpress
ontend
type: LoadBalancer
```

templates/pvc.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: wp-pv-claim
  labels:
    app: wordpress
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: {{ .Values.storage }}
```

templates/secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: wordpress-admin-password
data:
  key: {{ .Values.passwordEncoded }}
```

Values file:

values.yaml

```
image: wordpress:4.8-apache
storage: 20Gi
passwordEncoded: CajhWVUxSdzI2Qzg
```

Helm Chart: The templates and the values file form a Helm Chart.
There is the chart.yaml included too which contains information about the Chart itself.

Artifact Hub: Create your own chart for your own application or explore existing charts created by other users. Its a repository that stores helm charts.

Search: You can search in the artifact hub for charts or you can use the command:

```
helm search hub wordpress
```

Add repository:
Artifact hub is the community driven chart repository.

There are other repositories as well, like Bitnami.

To search in other repository you must add it first:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
```

The search the repository:

```
helm search repo wordpress
helm repo list
```

Install package:

helm install [release_name] [chart_name]

When this command is run the helm chart package is downloaded from the repository and extracted and installed locally.

Each installation of this chart is called Release.
Each release has a release name.

Example:
Install the same chart several times by running the helm install command

```
helm install release-1 bitnami/wordpress
helm install release-2 bitnami/wordpress
helm install release-3 bitnami/wordpress
```

Each release is completely independent of each other.

Recap:

Helm Chart: Kubernetes YAML template-files combined into a single package, Values allow customisation

Helm Release: Installed instance of a Chart

Helm Values: Allow to customise the YAML template-files in a Chart when creating a Release

Exercises / Commands:

Helm:
List Carts: List installed charts
helm list or helm -n mercury ls

List Repositories: List already installed repositories
helm repo list

Search for helm chart in Artifact Hub: Search for wordpress in the default repository
helm search hub wordpress

Add Bitnami helm chart repository:
helm repo add bitnami https://charts.bitnami.com/bitnami

Search Joomla from the Bitnami repository:
helm search repo joomla

List pending-install: → hidden by default.
helm -n mercury ls -a

Delete Release: → release name: internal-issue-report-apiv1
helm -n mercury uninstall internal-issue-report-apiv1

Upgrade Release:
Gets list of charts of the repo: helm repo list

Update the repo: helm repo update

Upgrade the Chart:
helm upgrade internal-issue-report-apiv2 bitnami/nginx

Install a new release: We want the deployment to have 2 replicas. Do it via helm values during installing.
helm show values bitnami/apache | yq e |grep -i replica
Search for replicaCount

-- And install the release with the deployment having 2 replicas:
helm -n mercury install internal-issue-report-apache bitnami/apache --set replicaCount=2

-- Adding even more values:
helm -n mercury install internal-issue-report-apache bitnami/apache \
--set replicaCount=2 \
--set image.debug=true

List hidden Releases: Hidden because are in pending-install state
helm -n mercury ls -a

Create cron: With two commands on it
k -n neptune create job neb-new-job --image=busybox:1.31.0 $do > /opt/course/3/job.yaml -- sh -c "sleep 2 && echo done"

Create pod: with commands on it
k run pod6 --image=busybox:1.31.0 $do -- sh -c "touch /tmp/ready && sleep 1d" > 6.yaml

Deployments:
See history:
k rollout history deployment name_of_the_deployment

Rollback to previous version:
k rollout undo deployment name_of_the_deployment

Restart all the pods in a deployment:
k rollout restart deployment web-moon

Run temporary pod: for curl testing
k run tmp --restart=Never --rm --image=nginx:alpine -i -- curl http://project-plt-6cc-svc.pluto:3333

This will create a pod, will run curl over that service dns address and will remove it self. project-plt-6cc-svc is the service name and pluto is the ns, if you are in the same ns (pluto in this example) is not necessary to add the ns, just the service and the target port (3333)

To know that works you have to see the welcome to nginx pate, not a failed to connect error

Another example:
k run tmp --restart=Never --rm -i --image=nginx:alpine -- curl 10.44.0.78

ConfigMaps:
Create cm form file and change the data key-name.

k create configmap configmap-web-moon-html --from-file=**index.html**=/opt/course/15/web-moon.html → important to set the index.html key

This is the file: /opt/course/15/web-moon.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Web Moon Webpage</title>
</head>
<body>
```

```
This is some great content.
</body>
</html>
```

This is how it have to be, index.html as data key-name

```
apiVersion: v1
data:
  index.html: |-
    <!DOCTYPE html>
    <html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Web Moon Webpage</title>
    </head>
    <body>
    This is some great content.
    </body>
    </html>
kind: ConfigMap
metadata:
  creationTimestamp: null
  name: configmap-web-moon-html
```

This is how would be if the index.html data key-name is not specified.

k create configmap configmap-web-moon-html --from-file=/opt/course/15/web-moon.html

```
apiVersion: v1
data:
  web-moon.html: |-
    <!DOCTYPE html>
    <html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Web Moon Webpage</title>
    </head>
    <body>
    This is some great content.
    </body>
    </html>
kind: ConfigMap
metadata:
  creationTimestamp: null
  name: configmap-web-moon-html
  namespace: moon
```

Check that the name has been changed in the pods:

k exec web-moon-66c79744fb-24lzf -- find /usr/share/nginx/html

You have to see a few files ending in index.html

```
/usr/share/nginx/html/..2022_03_03_07_39_26.3333887647/index.html
```

If you don’t change the data key-name you will see this

```
/usr/share/nginx/html/..2022_03_03_07_17_59.3753135772/web-moon.html
```

Network Policies: https://editor.cilium.io/

BE AWARE, in the k8s documentation the network policy examples are NOT using the second hyphen, see example:

```
  policyTypes:
  - Egress
  egress:
  - to:
    - podSelector:
```

```
      matchLabels:
        id: api
  ports:
    - protocol: TCP
      port: 53
    - protocol: UDP
      port: 53
```

In the above example there is only one hyphen in *from:* but is NOT in *ports:* which means:
allow outgoing traffic if:
(destination pod has label id:api) **AND** ((port is 53 UDP) OR (port is 53 TCP))

So both rules must match.

You have to use the hyphen in both cases if you want to use OR

```
  policyTypes:
  - Egress
  egress:
  - to:
    - podSelector:
        matchLabels:
          id: api
  - ports:
    - protocol: TCP
      port: 53
    - protocol: UDP
      port: 53
```

Which says:
allow outgoing traffic if
(destination pod has label id:api) **OR** ((port is 53 UDP) OR (port is 53 TCP))

Checking connections before and after:
The exercise asks to create an Egress rule from frontend pod to api pod and allow 53 tcp/udp as well.

So we should be able to access everything before applying the rule, like from frontend to google.com

But after applying the rule we should be able to access only the API pod, nothing else.

Before applying the policy:
exec frontend-556db8f554-zb7pp -- wget -O- www.google.com

```
Connecting to www.google.com (142.250.178.4:80)
-                    100% |********************************| 14104  0:00:00 ETA
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="en"><head><meta content="Search the world's information, including webpages, images, videos and more. Google has many special features to help you find exactly what you're looking for." name="description"><meta content="noodp" name="robots"><meta content="text/html; charset=UTF-8" http-equiv="Content-Type"><meta content="/images/branding/googleg/1x/googleg_standard_color_128dp.png" i
[…]
```

After applying the network policy:
k exec frontend-556db8f554-zb7pp -- wget -O- www.google.de

Connecting to www.google.de (216.58.213.3:80)
^C

Check before and after the connection to api pod:
k -n venus exec frontend-789cbdc677-c9v8h -- wget -O- api:2222

Labels:

Find pods with the label type=runner and add them the label protected=true
k label pod -l type=runner protected=true

Or even better:
k label pod -l "type in (worker,runner)" protected=true

Add an annotation to the pods with label protected=runner

k annotate pod -l protected=true protected="do not delete this pod"