Kubernetes - Part 2

Solar Team

Pod Overview

- Pods are the smallest deployable units of computing that you can create and manage in Kubernetes.
- A Pod is a logical group of one or more containers.
- Usually, you don't need to create Pods directly.
- Kubernetes uses workload resources such as Deployment, Job, or StatefulSet to implement application scaling and auto-healing.
- A Pod is the **basic unit of scaling**.
- A Pod is not intended to be treated as a durable, long-lived entity.
- Pods do not "heal" or repair themselves.
- https://kubernetes.io/docs/reference/kubernetes-api/workloads-resources/pod-v1/

Basic Pod Manifest

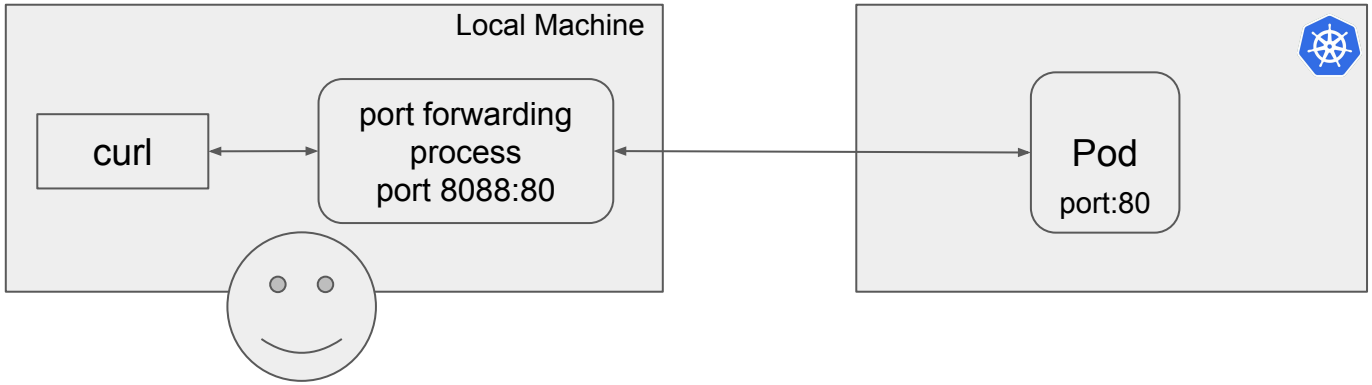
```yaml
apiVersion: v1
kind: Pod
metadata:
 name: web
 labels:
   app: web
spec:
 containers:
 - name: nginx
   image: nginx
```

Managing and Accessing Pods

- Create - Imperative command
  - kubectl run nginx --image=nginx
  - kubectl get all # check resources
  - Option --generator=run-pod/v1 needs for kubectl version below 1.18
  - kubectl run web --image=nginx --replicas=3 --generator=run-pod/v1 # what happen!
  - kubectl get all # check resources
- Create/Update - Imperative object configuration
  - kubectl create -f <YAML file>
  - kubectl apply -f <YAML file>
- Delete
  - kubectl delete pods <pod name>

Access Pods with Port Forwarding

- Port forwarding is useful for testing Pod accessibility.
- More advanced ways to access Pods are by using Service objects
- kubectl run web --image=nginx # check message returned and created resource
- kubectl run web --image=nginx -generator=run-pod/v1 # create only pod
- kubectl port-forward web-xxx 8088:80
- curl localhost:8888



Pods Debuging

- kubectl logs <pod name>
- kubectl logs <pod name> --container=<container name> # multiple container pods
- kubectl exec -it <container name> sh
- kubectl exec -it <pod name> -c <container name> sh
- kubectl run -i --tty --rm=true busybox --image=radial/busyboxplus --restart=Never -- sh # utility pod
  - nslookup <pod name>
  - curl <pod name>

Show Pod Details - kubectl describe pods web

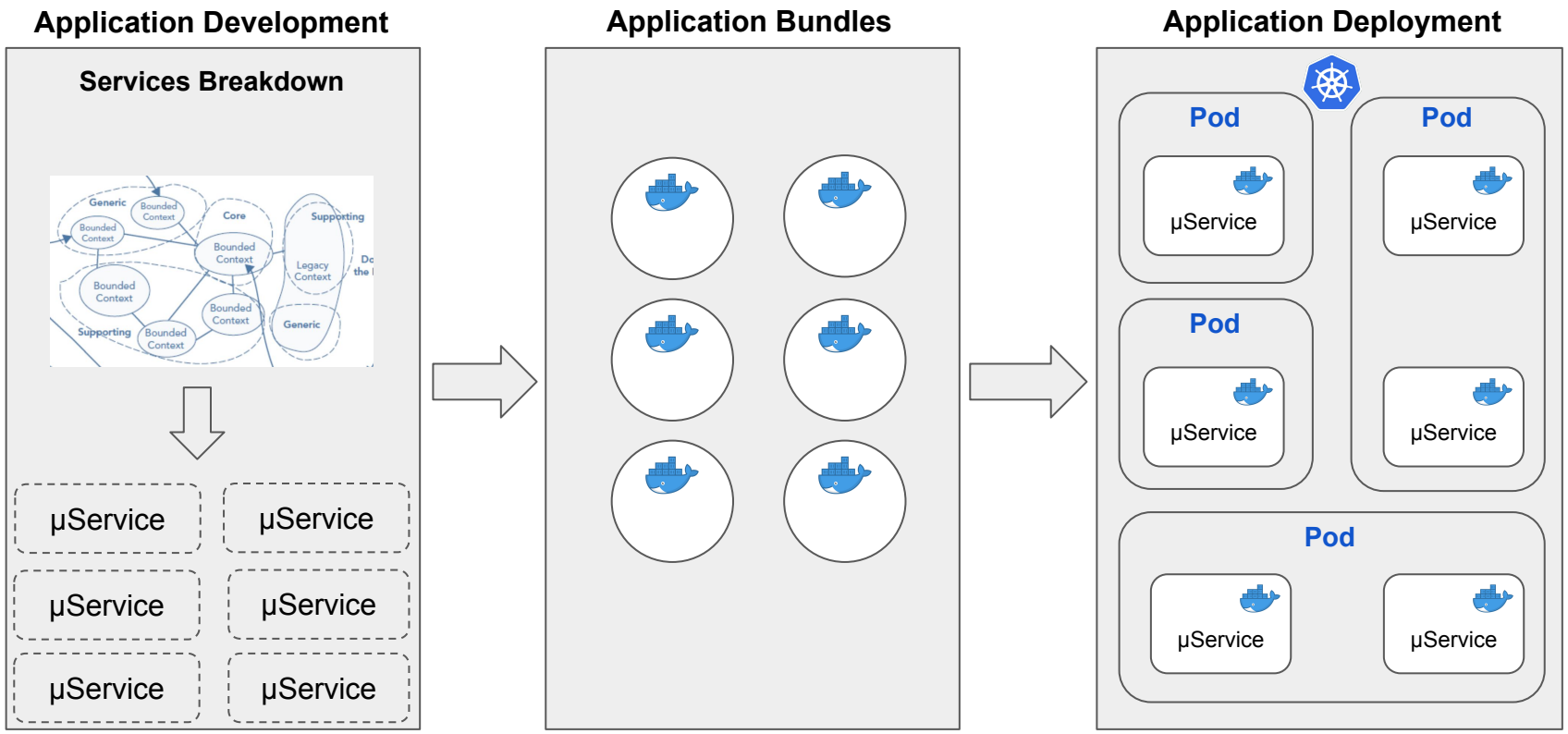
```
Name:         web
Namespace:    default
Priority:     0
Node:         docker-desktop/192.168.65.3
Start Time:   Tue, 16 Mar 2021 17:34:00 +0700
Labels:       app=web
              env=dev
Annotations:  kubectl.kubernetes.io/last-applied-configuration:
                {"apiVersion":"v1","kind":"Pod","metadata":{"annotations":{},"labels":{"app":"web","env":"dev"},"name":"web","namespace":"default"},"spec"...
Status:       Running
IP:           10.1.0.204
IPs:          <none>
Containers:
  nginx:
    Container ID:   docker://bd83e07518e4453ebc045836ac4bb3b302624e0257d8be68878b9f010f2b17e3
    Image:          nginx
    Image ID:       docker-pullable://nginx@sha256:d2925188effb4ddca9f14f162d6fba9b5fab232028aa07ae5c1dab764dca8f9f
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Tue, 16 Mar 2021 17:34:06 +0700
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-w7plz (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-w7plz:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-w7plz
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:          <none>
```

Pod Details - kubectl get pods web -o yaml


```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |

{"apiVersion":"v1","kind":"Pod","metadata":{"annotations":{},"labels":{"app":"web","env":"dev"},"
name":"web","namespace":"default"},"spec":{"containers":[{"image":"nginx","name":"nginx"}]}}
  creationTimestamp: "2021-03-16T10:34:00Z"
  labels:
    app: web
    env: dev
  name: web
  namespace: default
  resourceVersion: "1307783"
  selfLink: /api/v1/namespaces/default/pods/web
  uid: 1eedca06-8643-11eb-bd28-025000000001
spec:
  containers:
  - image: nginx
    imagePullPolicy: Always
    name: nginx
    resources: {}
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: default-token-w7plz
      readOnly: true
  dnsPolicy: ClusterFirst
  enableServiceLinks: true
  nodeName: docker-desktop
  priority: 0
  restartPolicy: Always
  schedulerName: default-scheduler
  securityContext: {}
  serviceAccount: default
  serviceAccountName: default
  terminationGracePeriodSeconds: 30
  tolerations:
  - effect: NoExecute
    key: node.kubernetes.io/not-ready
    operator: Exists
    tolerationSeconds: 300
  - effect: NoExecute
    key: node.kubernetes.io/unreachable
    operator: Exists
    tolerationSeconds: 300
  volumes:
  - name: default-token-w7plz
    secret:
      defaultMode: 420
      secretName: default-token-w7plz
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: "2021-03-16T10:34:00Z"
    status: "True"
    type: Initialized
  - lastProbeTime: null
    lastTransitionTime: "2021-03-16T10:34:08Z"
    status: "True"
    type: Ready
  - lastProbeTime: null
    lastTransitionTime: "2021-03-16T10:34:08Z"
    status: "True"
    type: ContainersReady
  - lastProbeTime: null
    lastTransitionTime: "2021-03-16T10:34:00Z"
    status: "True"
    type: PodScheduled
  containerStatuses:
  - containerID:
docker://bd83e07518e4453ebc045836ac4bb3b302624e0257d8be68878b9f
010f2b17e3
    image: nginx:latest
    imageID:
docker-pullable://nginx@sha256:d2925188effb4ddca9f14f162d6fba9b5fab2
32028aa07ae5c1dab764dca8f9f
    lastState: {}
    name: nginx
    ready: true
    restartCount: 0
    state:
      running:
        startedAt: "2021-03-16T10:34:06Z"
  hostIP: 192.168.65.3
  phase: Running
  podIP: 10.1.0.204
  qosClass: BestEffort
  startTime: "2021-03-16T10:34:00Z"
```

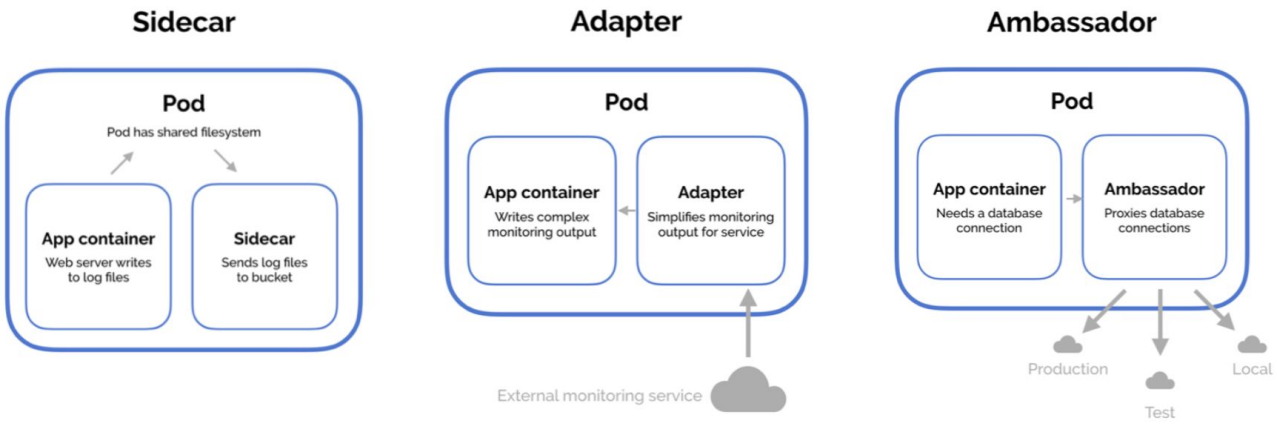
Microservices, Containers and Pods



Containers In Pod

- Pods in a Kubernetes cluster are used in two main ways:
  - **Pods that run a single container**. The "one-container-per-Pod" model is the most common Kubernetes use case.
  - **Pods that run multiple containers that need to work together.** A Pod can encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources.
- Containers are designed to run only a single process.
- Most recommended is one container for a Pod.
- Thinks about functionality, lifecycle dependency, scaling, resources limitation.
- Containers within the same Pod will **share storage and network resources**.
  - Pod containers share the same network namespace, including IP address and network ports.
  - Containers in a Pod communicate with each other inside the Pod on localhost.
  - Pods can specify a set of shared storage volumes that can be shared among the containers.

Multi-container Pod Patterns

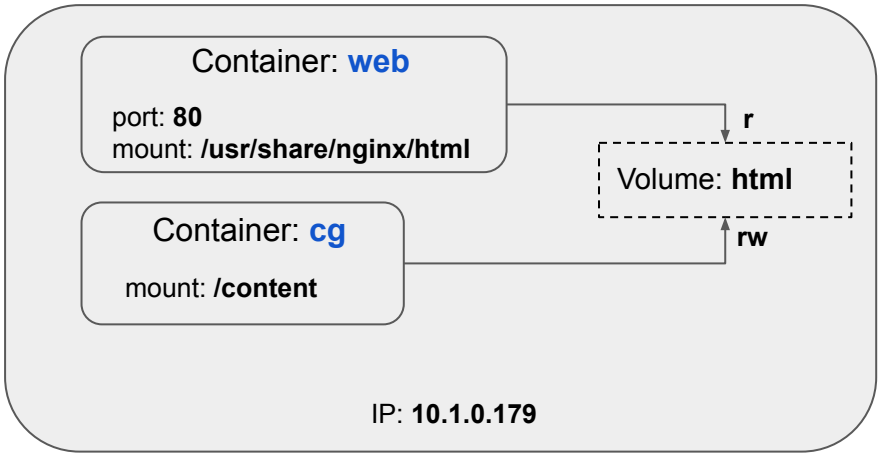


- Three common design patterns and use-cases for combining multiple containers into a single pod.
  - **Sidecar pattern**: A pod spec which runs the main container and a helper container that does some utility work.
  - **Adapter pattern**: Used to standardize and normalize application output or monitoring data for aggregation. It does some kind of restructuring and reformat it, and write the correctly formatted output to the location.
  - **Ambassador pattern**: It connects containers with the outside world. It is a proxy that allows other containers to connect to a port on localhost.

Multiple Containers - Example

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: mc
spec:
 volumes:
 - name: html
   emptyDir: {}
 containers:
 - name: web
   image: nginx
   volumeMounts:
   - name: html
     mountPath: /usr/share/nginx/html
     readOnly: true
 - name: cg # Content Generator
   image: debian
   volumeMounts:
   - name: html
     mountPath: /content
   command: ["/bin/sh", "-c"]
   args:
     - while true; do
         date >> /content/index.html;
         sleep 5;
       done
```

apply



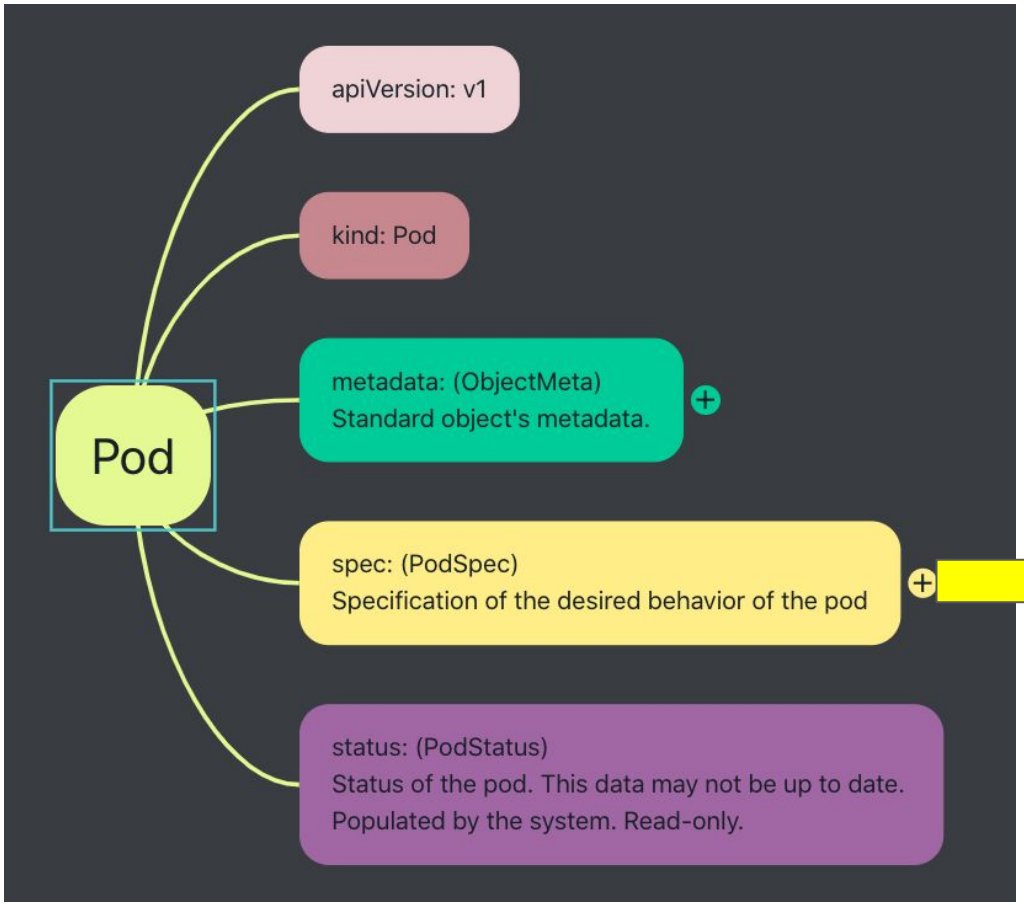
multiple-containers.yml

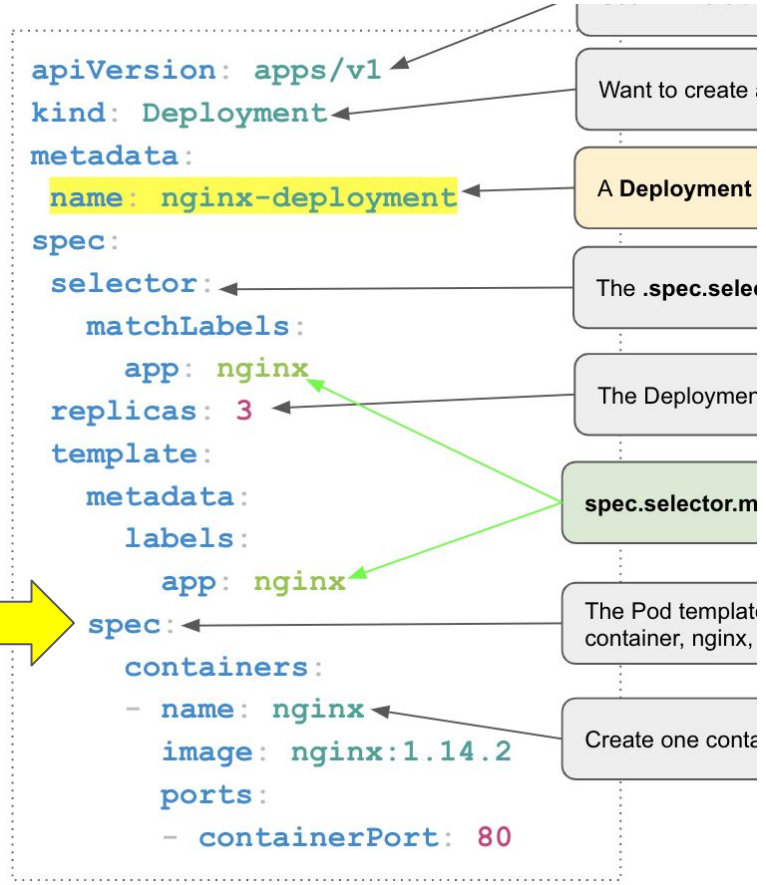
Multiple Containers - Examples (con’t)

- k apply -f multiple-containers.yml
- k get po mc -w
- k describe po mc
- k port-forward pod/mc 8889:80
- k logs mc -c web
- k exec -it mc -c web sh
- curl localhost:8889
- k delete -f multiple-containers.yml

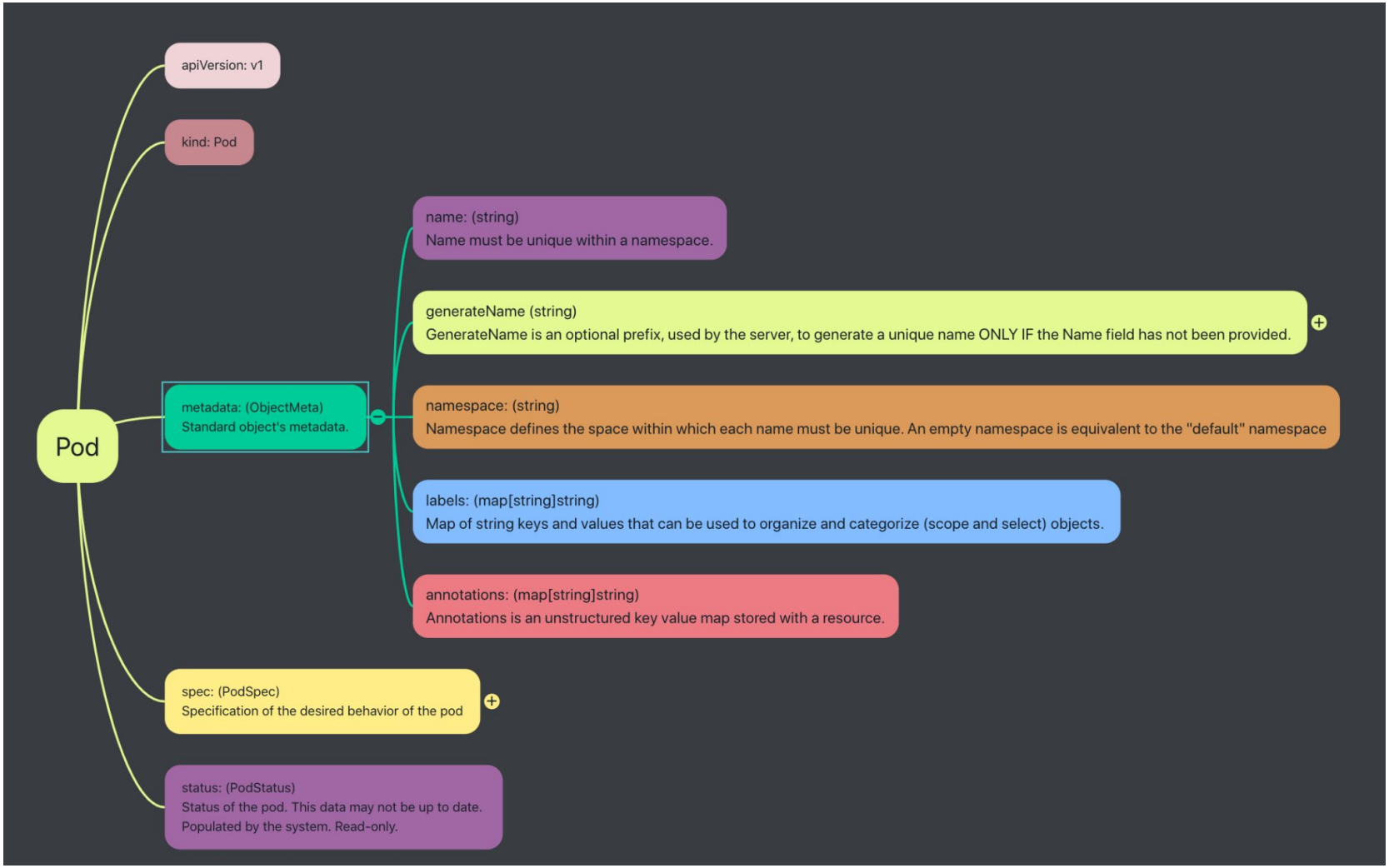
Pod Manifest

- PodSpec is a description of a pod.

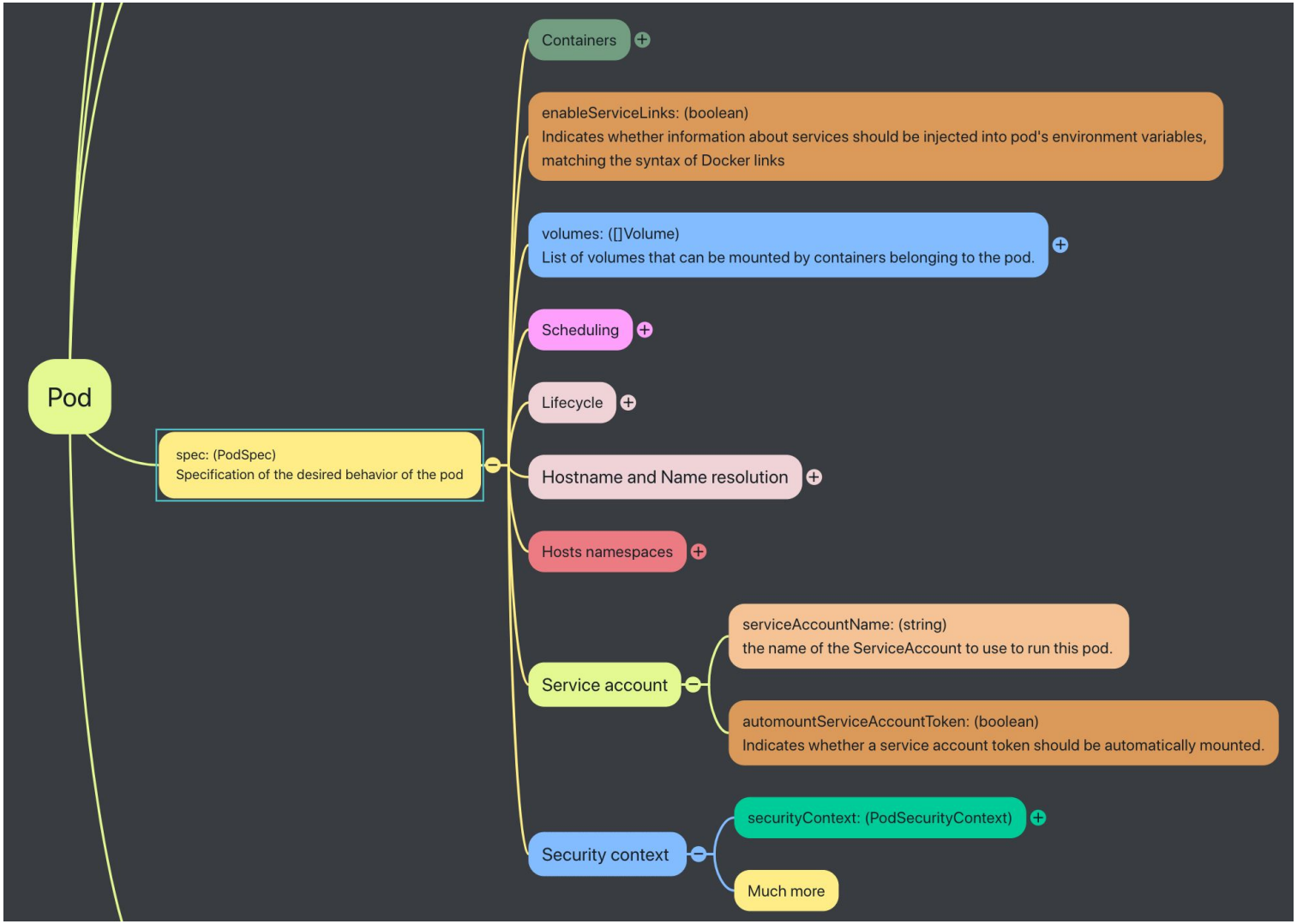




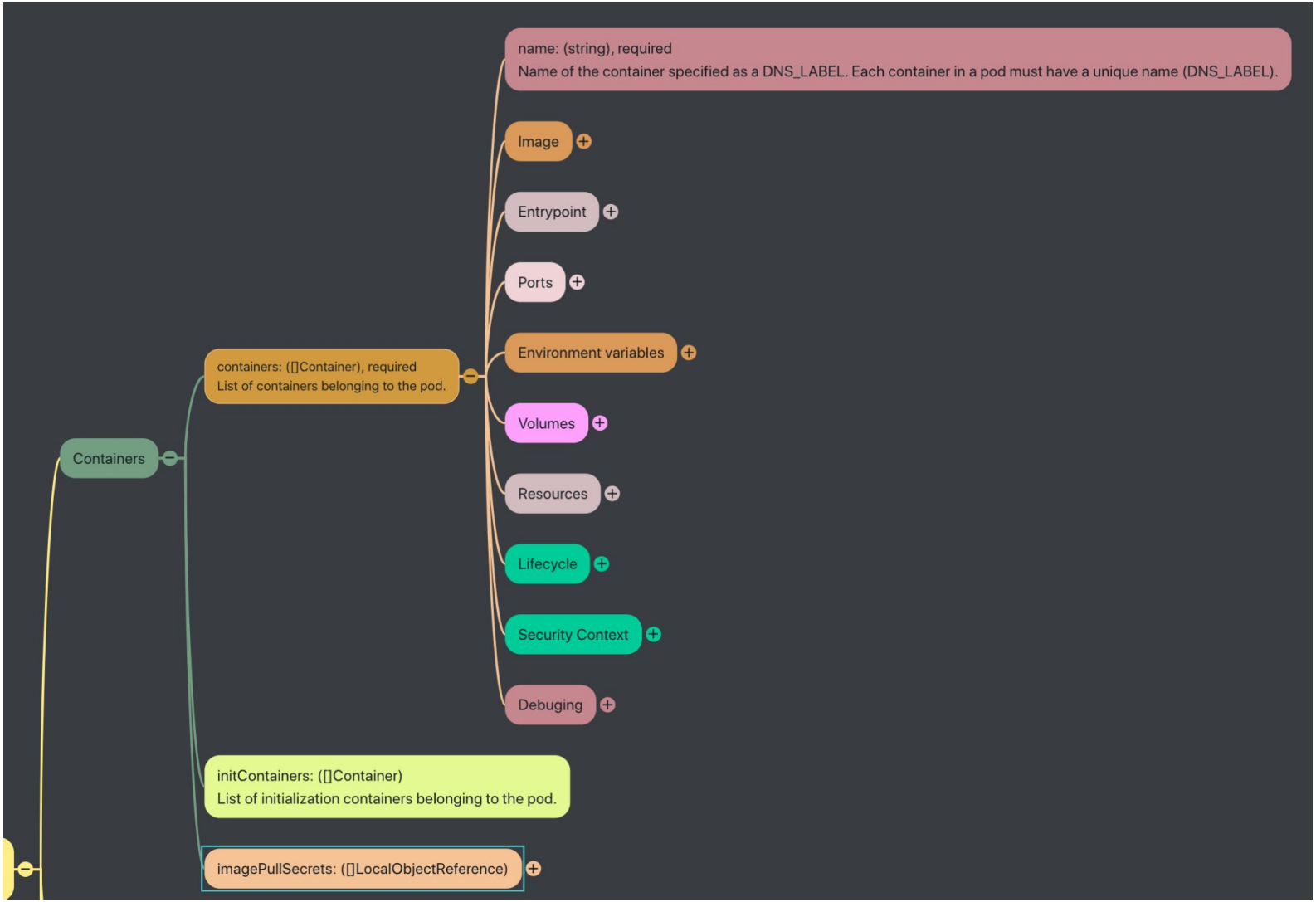
Pod Manifest - Metadata



Pod Manifest - PodSpec

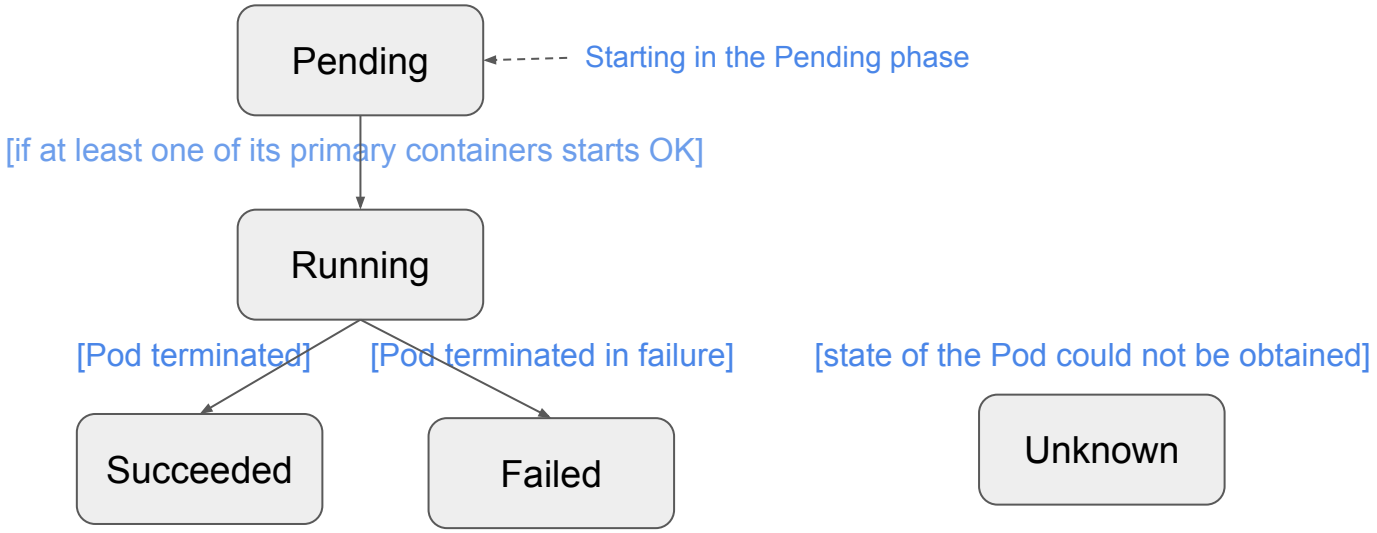


Pod Manifest - Containers



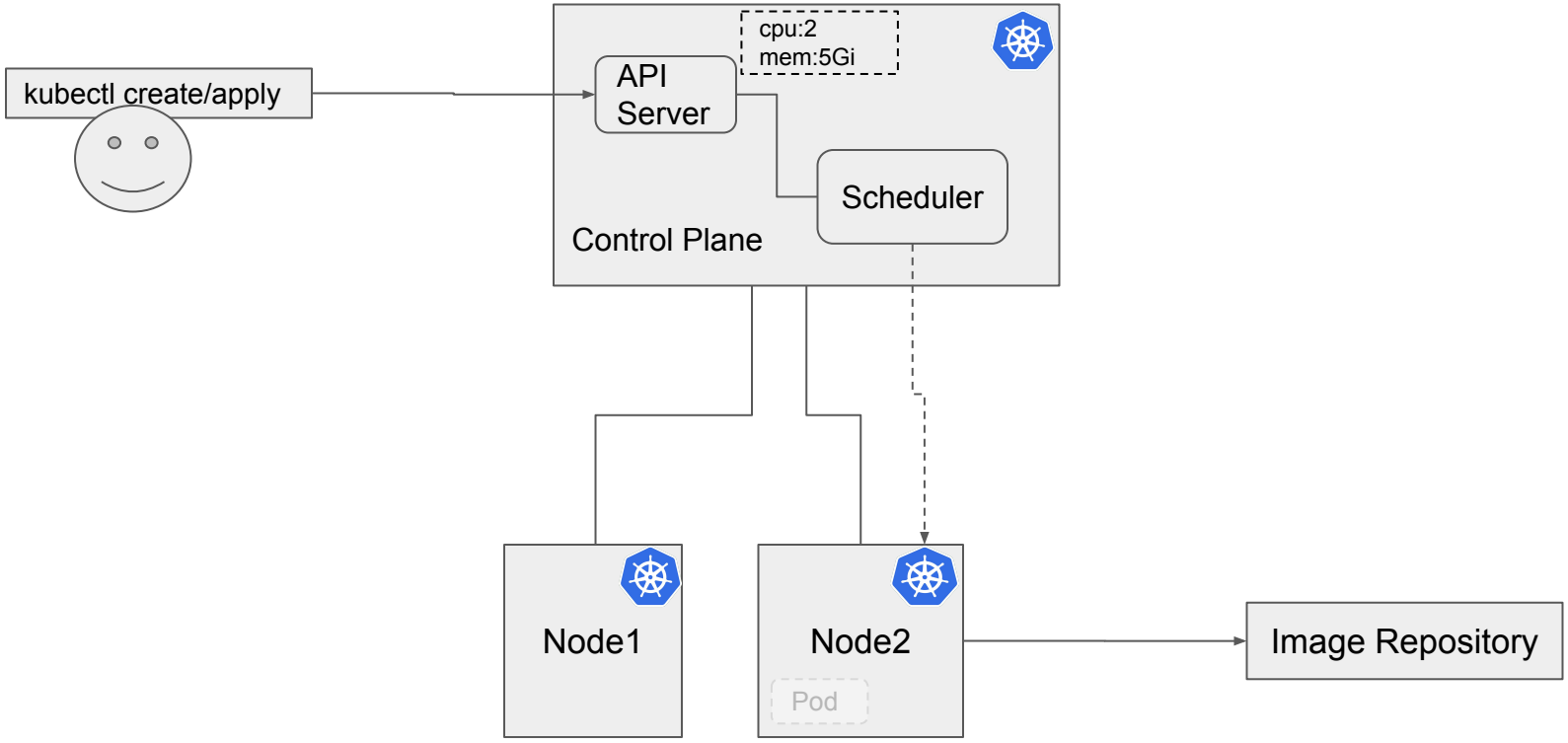
Pod Life-cycle

- Pods are only scheduled once in their lifetime.
- The Pod has a unique ID (UID).
- Once a Pod is scheduled (assigned) to a Node, the Pod runs on that Node until it stops or is terminated.
- A Pod's status field is a PodStatus object, which has a phase field.
- Pods were defined a lifecycle as 5 phases.



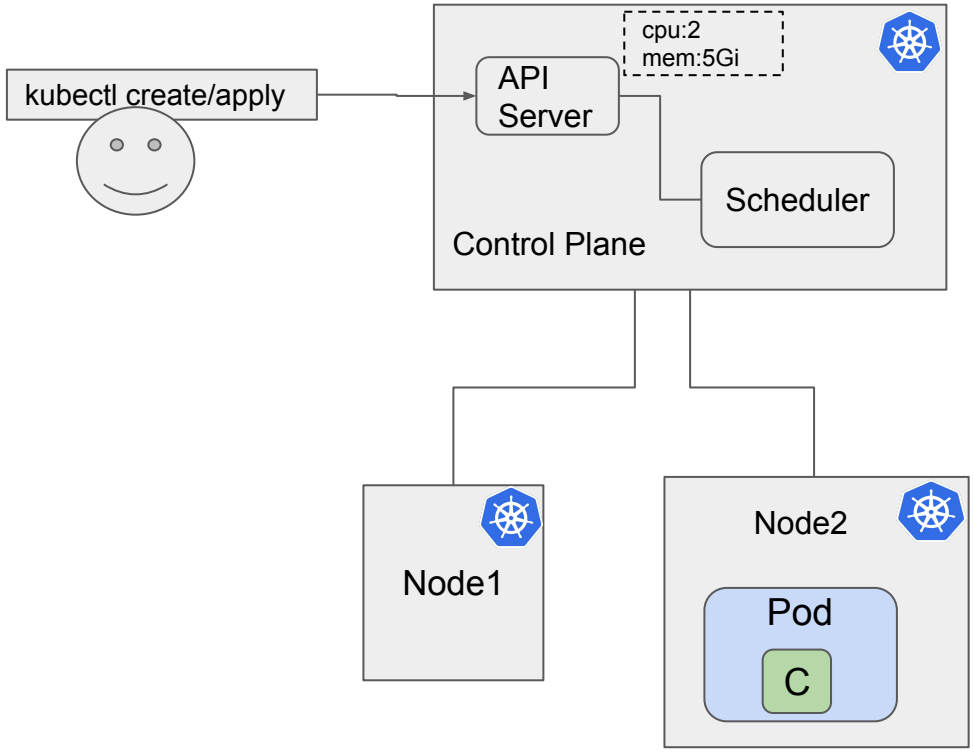
Pod Life-cycle - Pending

- **Pending** - The Pod has been accepted by the Kubernetes cluster, but one or more of the containers has not been set up and made ready to run.
- This includes time a Pod spends waiting to be scheduled as well as the time spent downloading container images over the network.



Pod Life-cycle - Running

- **Running** - The Pod has been bound to a node, and all of the containers have been created. At least one container is still running, or is in the process of starting or restarting.

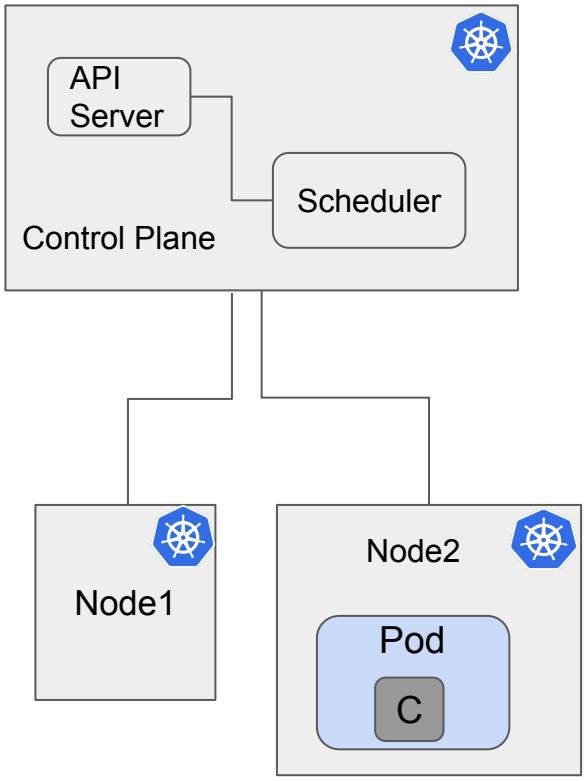


lc1.yml

```
containers:
  - name: myapp
    image: busybox
    command: ['sh', '-c', 'echo
The Pod is running && sleep 6']
```

Pod Life-cycle - Succeeded Phase

- **Succeeded** - All containers in the Pod have terminated in success, and will not be restarted.

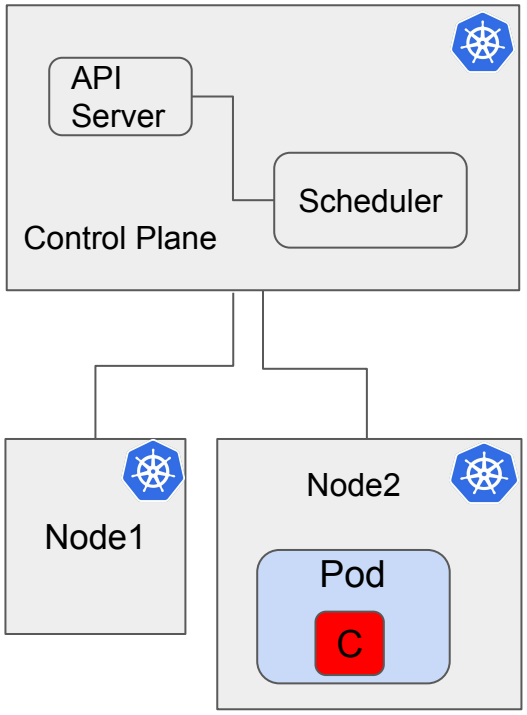


lc2.yml

```
containers:
 - name: myapp
   image: busybox
   command: ['sh', '-c', 'echo The
Pod is running && sleep 6']
 restartPolicy: Never
```

Pod Life-cycle - Failed

- **Failed** - All containers in the Pod have terminated, and at least one container has terminated in failure.
- If a node dies or is disconnected from the rest of the cluster, Kubernetes applies a policy for setting the phase of all Pods on the lost node to Failed.

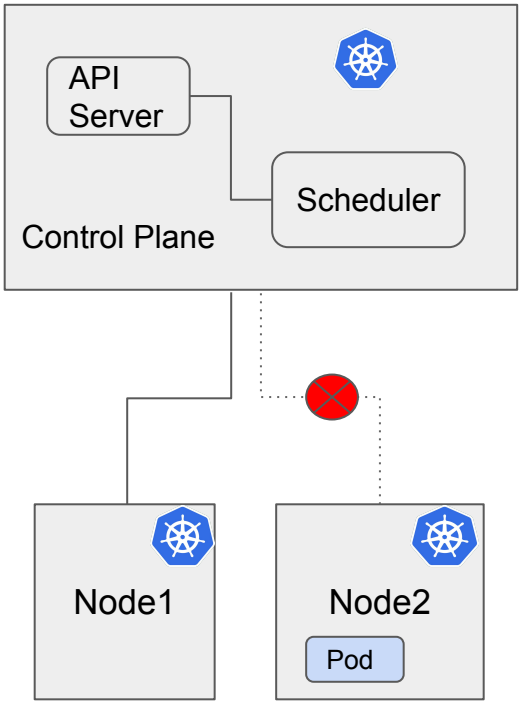


lc3.yml

```
containers:
 - name: myapp
   image: busybox
   imagePullPolicy: IfNotPresent
   command: ['sh', '-c', 'echo The Pod is
running && exit 1']
 restartPolicy: Never
```

Pod Life-cycle - Unknown

- **Unknown** - For some reason the state of the Pod could not be obtained.
- This phase typically occurs due to an error in communicating with the node where the Pod should be running.



Pod Conditions

- A Pod has a PodStatus, which has an array of PodConditions through which the Pod has or has not passed:
  - **PodScheduled**: the Pod has been scheduled to a node.
  - **ContainersReady**: all containers in the Pod are ready.
  - **Initialized**: all init containers have started successfully.
  - **Ready**: the Pod is able to serve requests and should be added to the load balancing pools of all matching Services.

Field name	Description
`type`	Name of this Pod condition.
`status`	Indicates whether that condition is applicable, with possible values " `True` ", " `False` ", or " `Unknown` ".
`lastProbeTime`	Timestamp of when the Pod condition was last probed.
`lastTransitionTime`	Timestamp for when the Pod last transitioned from one status to another.
`reason`	Machine-readable, UpperCamelCase text indicating the reason for the condition's last transition.
`message`	Human-readable message indicating details about the last status transition.

Pod Conditions - Example


```
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: "2021-03-22T13:22:52Z"
    status: "True"
    type: Initialized
  - lastProbeTime: null
    lastTransitionTime: "2021-03-22T13:22:48Z"
    message: 'containers with unready status:
[myapp-container]'
    reason: ContainersNotReady
    status: "False"
    type: Ready
  - lastProbeTime: null
    lastTransitionTime: "2021-03-22T13:22:48Z"
    message: 'containers with unready status:
[myapp-container]'
    reason: ContainersNotReady
    status: "False"
    type: ContainersReady
  - lastProbeTime: null
    lastTransitionTime: "2021-03-22T13:22:48Z"
    status: "True"
    type: PodScheduled
```

Pod Readiness - A Custom Pod Condition

- Your application can inject extra feedback or signals into PodStatus: Pod readiness.
- To use this, set readinessGates in the Pod's spec to specify a list of additional conditions that the kubelet evaluates for Pod readiness.

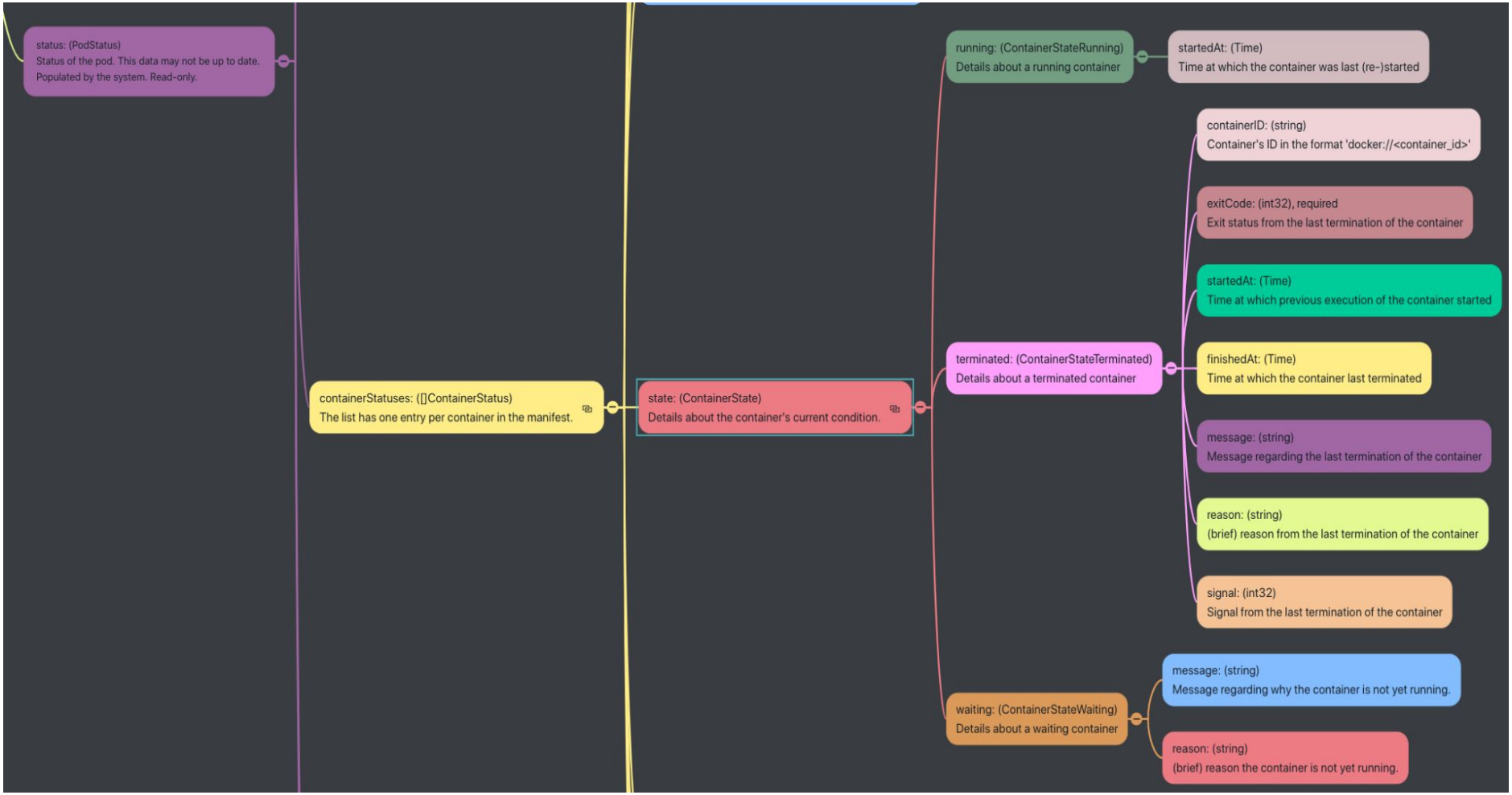
```yaml
kind: Pod
...
spec:
  readinessGates:
    - conditionType: "www.example.com/feature-1"
status:
  conditions:
    - type: Ready                              # a built in PodCondition
      status: "False"
      lastProbeTime: null
      lastTransitionTime: 2018-01-01T00:00:00Z
    - type: "www.example.com/feature-1"        # an extra PodCondition
      status: "False"
      lastProbeTime: null
      lastTransitionTime: 2018-01-01T00:00:00Z
  containerStatuses:
    - containerID: docker://abcd...
      ready: true
...
```

- Pod is evaluated to be ready when, all containers in the Pod are ready and all conditions specified in readinessGates are True

Container Life-cycle

- Kubernetes tracks the state of each container inside a Pod.
- There are three possible container states: **Waiting**, **Running**, and **Terminated**.
- **Waiting state** - a container is still running the operations it requires in order to complete start up.
- **Running state** - a container is executing without issues.
- **Terminated state** - a container began execution and then either ran to completion or failed for some reason.
- There is a **reason field** that summarize why the container is in that state.

Container Life-cycle - Container State



Container Lifecycle Hooks

- The hooks enable Containers to be aware of events in their management lifecycle.
- The developer provides a code to handle when the corresponding lifecycle hook happened.
- There are two hooks that are exposed to Containers:
  - **PostStart** - This hook is executed immediately after a container is created. However, there is no guarantee that the hook will execute before the container ENTRYPOINT.
  - **PreStop** - This hook is called immediately before a container is terminated due to an API request or management event such as a liveness/startup probe failure, preemption, resource contention and others.
- Two types of hook handlers that can be implemented for Containers:
  - **Exec** - Executes a specific command, such as pre-stop.sh
  - **HTTP** - Executes an HTTP request against a specific endpoint on the Container.

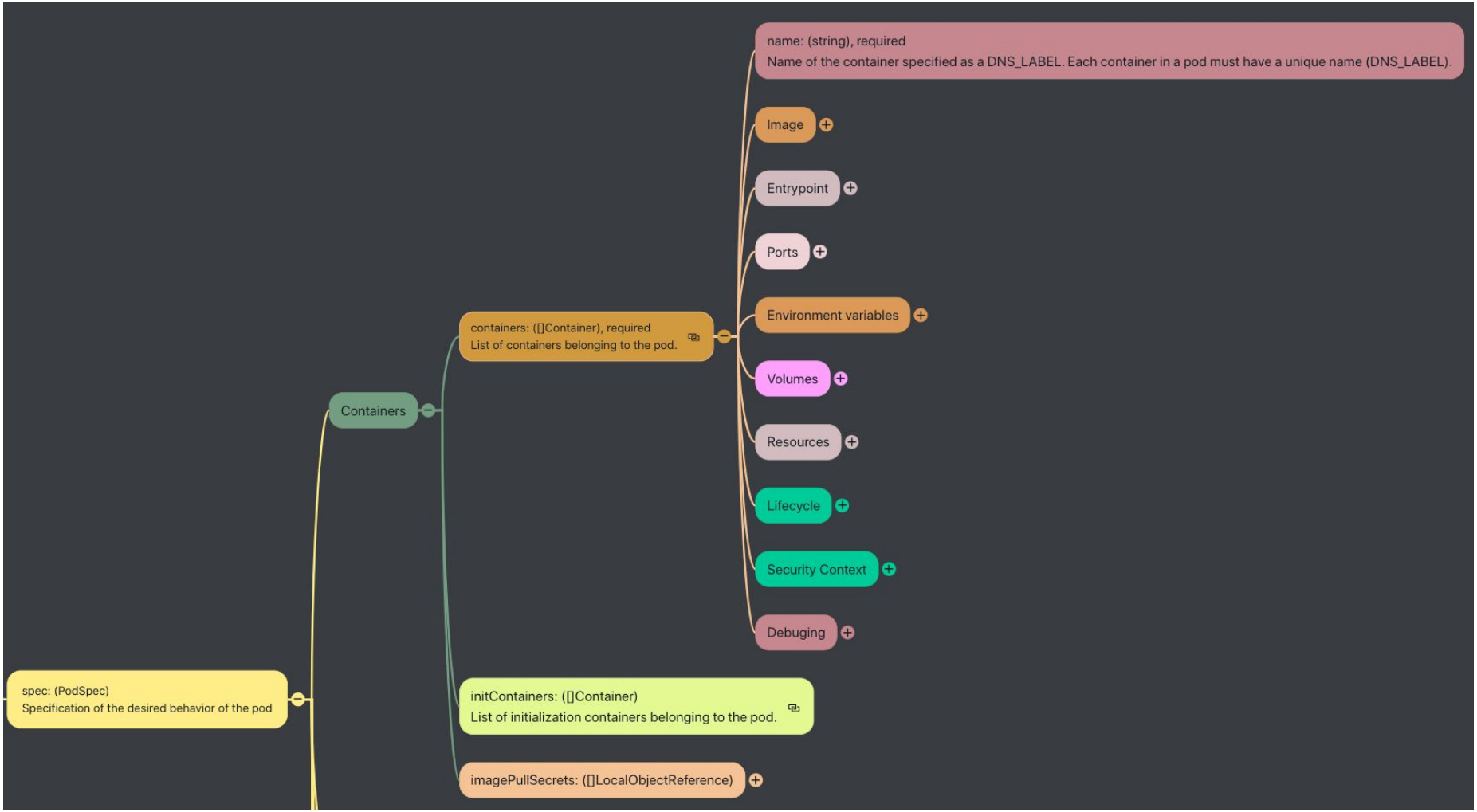
Container Lifecycle Hooks - Example

```
apiVersion: v1
kind: Pod
metadata:
 name: lifecycle-demo
spec:
 containers:
 - name: lifecycle-demo-container
   image: nginx
   lifecycle:
     postStart:
       exec:
         command: ["/bin/sh", "-c", "echo Hello from the postStart handler >
/usr/share/message"]
     preStop:
       exec:
         command: ["/bin/sh","-c","nginx -s quit; while killall -0 nginx; do sleep 1;
done"]
```

Container Restart Policy

- The spec of a Pod has a **restartPolicy** field with possible values Always, OnFailure, and Never.
- Always means that the container will be restarted even if it exited with a zero exit code (i.e. successfully). **This is the default.**
- OnFailure means that the container will only be restarted if it exited with a non-zero exit code (i.e. something went wrong).
- Never means that the container will not be restarted regardless of why it exited.
- The default value is Always.
- The restartPolicy applies to all containers in the Pod.
- Handle by kubelet (restart conatiners and reset backoff timer).
- Using an exponential back-off delay (10s, 20s, 40s, …), that is capped at 5 minutes
- Perform a reset backoff timer after 10 minutes of fully working.

Pod Manifest - initContainers



Init Containers

- Init containers run and complete before the app containers are started.
- Init containers are exactly like regular containers, except:
  - Init containers always run to completion.
  - Each init container must complete successfully before the next one starts.
- If restartPolicy=”Always” - the kubelet repeatedly restarts that init container until it succeeds.
- If restartPolicy=”Never” - Kubernetes treats the overall Pod as failed.
- Init containers do not support lifecycle, livenessProbe, readinessProbe, or startupProbe
- If you specify multiple init containers for a Pod, kubelet runs each init container sequentially.

Init Containers - Example

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: myapp-pod
 labels:
   app: myapp
spec:
 containers:
 - name: myapp-container
   image: busybox:1.28
   command: ['sh', '-c', 'echo The app is running! && sleep 3600']
 initContainers:
 - name: init-myservice
   image: busybox:1.28
   command: ['sh', '-c', "until nslookup myservice.$(cat
/var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluste
r.local; do echo waiting for myservice; sleep 2; done"]
 - name: init-mydb
   image: busybox:1.28
   command: ['sh', '-c', "until nslookup mydb.$(cat
/var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluste
r.local; do echo waiting for mydb; sleep 2; done"]
```

initContainer.yml

```yaml
apiVersion: v1
kind: Service
metadata:
 name: myservice
spec:
 ports:
 - protocol: TCP
   port: 80
   targetPort: 9376
```

initContainer-myservice.yml

```yaml
apiVersion: v1
kind: Service
metadata:
 name: mydb
spec:
 ports:
 - protocol: TCP
   port: 80
   targetPort: 9377
```

initContainer-mydb.yml

Container Probes

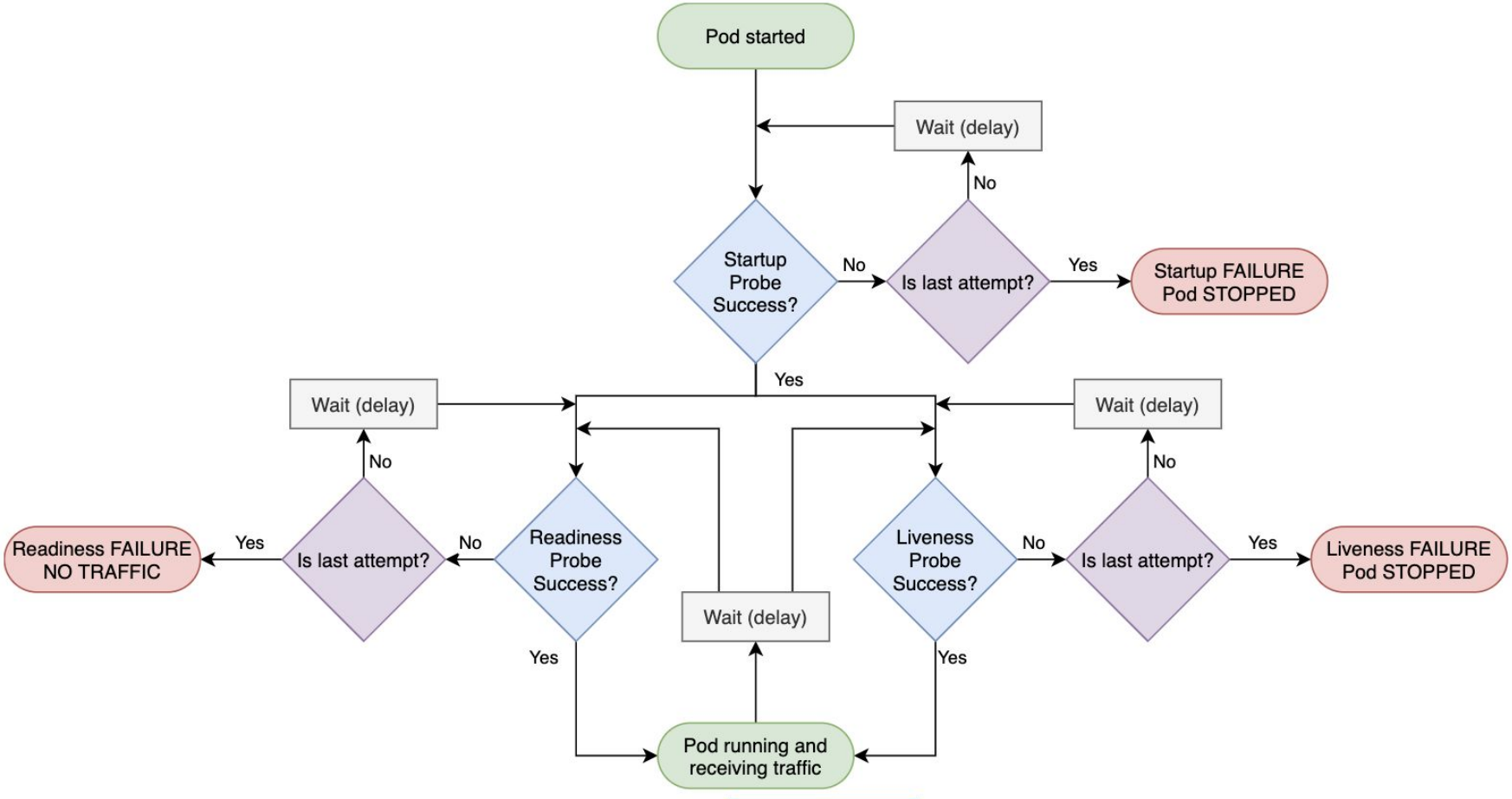
- **Container probes** are **diagnostics** and **health-check** performed by kubelet on the container.
- There are three kinds of probes which kubelet can run on running containers:
  - **livenessProbe** : The kubelet uses liveness probes to know when to restart a container.
  - **readinessProbe** : The kubelet uses readiness probes to know when a container is ready to start accepting traffic.
  - **startupProbe** (v1.20 stable) : The kubelet uses startup probes to know when a container application has started.
- **Init containers** may not have Lifecycle actions, Readiness probes, Liveness probes, or Startup probes.
- There are three ways to implement a probe:
  - **ExecAction** : Executes a command inside the container. The diagnostic is considered successful if the command returns 0.
  - **TCPSocketAction** : Performs a TCP socket check against the container IP and specified port. The diagnostic is considered successful if the port is open.
  - **HTTPGetAction** : Runs an HTTP GET action against the container IP with the specified port and path. The diagnostic is considered successful if the response has a status code between 200 and 400.

Probe Configurations

livenessProbe: (Probe)
Periodic probe of container liveness. Container will be restarted if the probe fails.

- exec: (ExecAction)
  One and only one of the following should be specified. Exec specifies the action to take.
- httpGet: (HTTPGetAction)
  HTTPGet specifies the http request to perform.
- tcpSocket: (TCPSocketAction)
  TCPSocket specifies an action involving a TCP port. TCP hooks not yet supported
- initialDelaySeconds: (int32)
  Number of seconds after the container has started before liveness probes are initiated.
- periodSeconds: (int32)
  How often (in seconds) to perform the probe. Default to 10 seconds. Minimum value is 1.
- timeoutSeconds: (int32)
  Number of seconds after which the probe times out. Defaults to 1 second. Minimum value is 1.
- failureThreshold: (int32)
  Minimum consecutive failures for the probe to be considered failed after having succeeded. Defaults to 3. Minimum value is 1.
- successThreshold: (int32)
  Minimum consecutive successes for the probe to be considered successful after having failed. Defaults to 1.

Probes Flow



Source: https://andrewlock.net/

Probe Configurations - Option Fields

- initialDelaySeconds: Number of seconds after the container has started before liveness or readiness probes are initiated. Defaults to 0 seconds. Minimum value is 0.
- periodSeconds: How often (in seconds) to perform the probe. Default to 10 seconds. Minimum value is 1.
- timeoutSeconds: Number of seconds after which the probe times out. Defaults to 1 second. Minimum value is 1.
- successThreshold: Minimum consecutive successes for the probe to be considered successful after having failed. Defaults to 1. Must be 1 for liveness and startup Probes. Minimum value is 1.
- failureThreshold: When a probe fails, Kubernetes will try failureThreshold times before giving up. Giving up in case of liveness probe means restarting the container. In case of readiness probe the Pod will be marked Unready. Defaults to 3. Minimum value is 1.

Probe Implementations

Agenda

- Pod
- Observability

Probe Implementations - Examples

HttpGetAction

```yaml
ports:
- name: liveness-port
  containerPort: 8080
  hostPort: 8080

livenessProbe:
  httpGet:
    path: /healthz
    port: liveness-port
  failureThreshold: 1
  periodSeconds: 10

startupProbe:
  httpGet:
    path: /healthz
    port: liveness-port
  failureThreshold: 30
  periodSeconds: 10
```

TCPSocketAction

```yaml
spec:
  containers:
  - name: goproxy
    image: k8s.gcr.io/goproxy:0.1
    ports:
    - containerPort: 8080
    readinessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 10
    livenessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 20
```

ExecAction

```yaml
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```

Other Probe Informations

- If readiness probe fails, the endpoints controller removes the container IP from list of endpoints of all services that match the Pod.
- The readiness probe is expected to respond positively only when your application is ready and able to service normal requests.
- Readiness probes runs on the container during its whole lifecycle.
- Liveness probes do not wait for readiness probes to succeed. If you want to wait before executing a liveness probe you should use initialDelaySeconds or a startupProbe.
- If the liveness probe fails, kubelet kills the container and the container is subjected to its Restart Policy.
- All other probes are disabled if a startup probe is provided, until it succeeds.
- If the startup probe fails, the kubelet kills the container, and the container is subjected to its restart policy.
- As soon as the startup probe succeeds once it never runs again for the lifetime of that container.

Probe - Demo

Advanced Scheduling

Assigning Pods to Nodes

- Kubernetes allows you to affect where pods are scheduled.
- Using node selector - a very simple way to constrain pods to nodes with particular labels.
- Using (node) **taints and** (pod) **tolerations** - to keep pods away from certain nodes.
- Using node affinity
- Note: PodAffinity and TopologyKey

Node Selector

- Node Selector is the simplest recommended form of node selection constraint.
- Node label(key=value) = Pod's nodeSelector(key=value
- Example:
  - kubectl label nodes kubernetes-foo-node-1.c.a-robinson.internal disktype=ssd
  - In Pod manifest

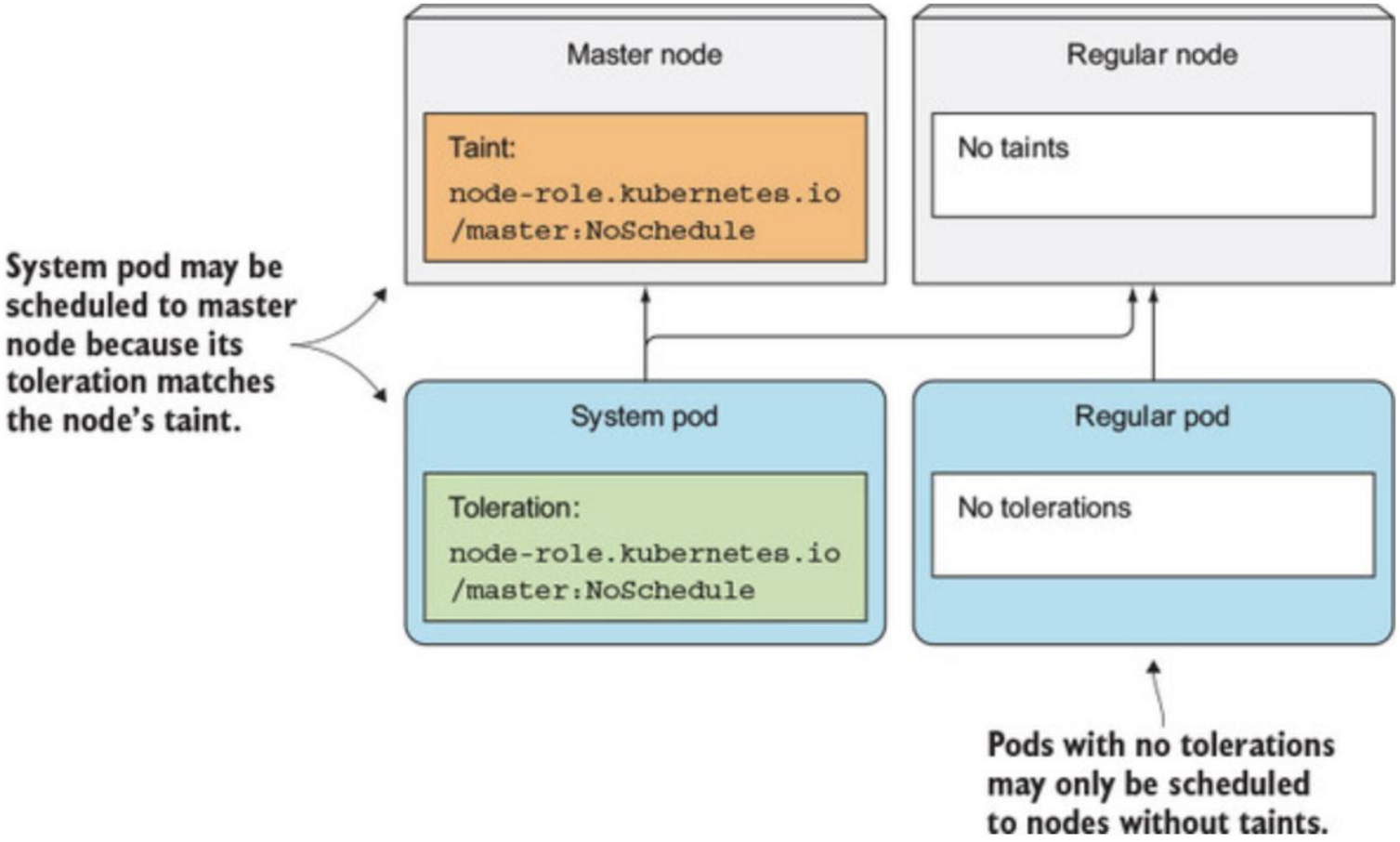
```
apiVersion: v1
kind: Pod
metadata:
 name: nginx
 labels:
   env: test
spec:
 containers:
 - name: nginx
   image: nginx
   imagePullPolicy: IfNotPresent
 nodeSelector:
   disktype: ssd
```

Taints and Tolerations

- Allow a node to repel a set of pods.
- Taints can be used to prevent scheduling of new pods (NoSchedule effect) and to define unpreferred nodes (PreferNoSchedule effect) and even evict existing pods from a node (NoExecute effect).
- A taint consists of a key, value, and effect. As an argument here, it is expressed as key=value:effect.
  - kubectl taint NODE NAME KEY_1=VAL_1:TAINT_EFFECT_1 ... KEY_N=VAL_N:TAINT_EFFECT_N
- Example:
  - kubectl taint node node1.k8s node-type=production:NoSchedule
  - Add toleration to Pod

```
spec:
  replicas: 5
  template:
    spec:
      ...
      tolerations:
      - key: node-type
        operator: Equal
        value: production
        effect: NoSchedule
```

Taints and Tolerations - Example

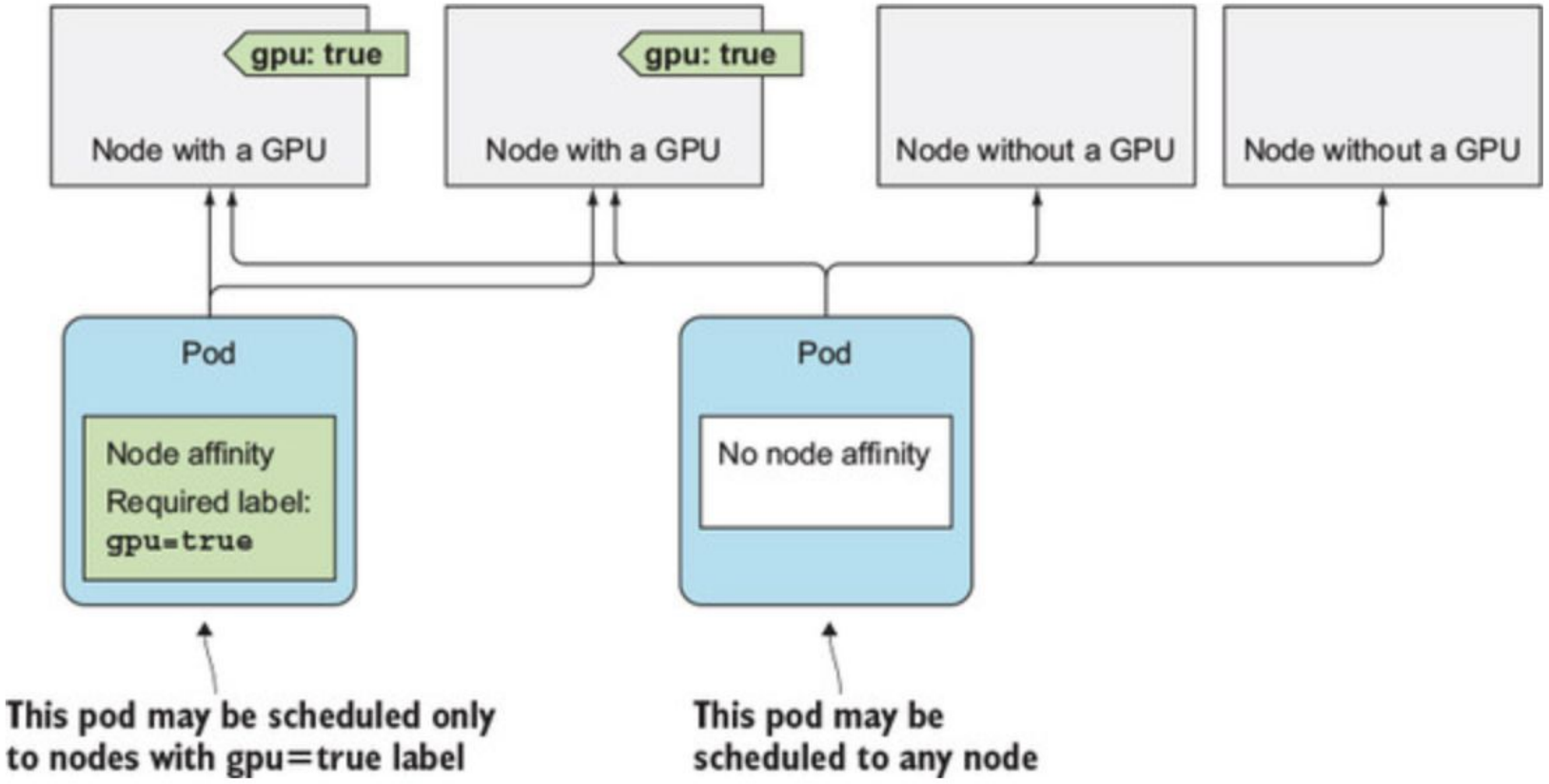


Source: https://livebook.manning.com/

Affinity and Anti-affinity

- The affinity/anti-affinity feature, greatly expands the types of constraints you can express (The language offers more matching rules besides exact matches created with a logical AND operation).
- It is conceptually similar to nodeSelector -- it allows you to constrain which nodes your pod is eligible to be scheduled on, based on labels on the node.
- Tell Kubernetes to schedule pods only to specific subsets of nodes (affinity) and vice versa (anti-affinity).
- Can specify both nodeSelector and nodeAffinity.

Node Affinity - Example



Source: https://livebook.manning.com/

Next...

ConfigMap

Environment Variables

Graceful shutdown

The End.