

Kubernetes: Advanced Pod Configuration

Course Outline

1. What is Kubernetes?

- Components
- Installation

2. Basics of Docker

- Namespaces
- Building and running Docker images

3. Pods and Deployments

- Running basic workloads in Kubernetes
- Scale, Update, Rollback

4. Advanced Pod configuration

- Args, Envs, ConfigMaps, Secrets
- Init- and sidecar containers
- Scheduling and debugging

5. Networking in Kubernetes

- What are network plugins?
- Service abstraction and ingress

6. Persistent storage

- Basics of storage: block vs. object vs. file system
- StorageClass, PVC, PV

7. Security

- RBAC: Roles, ServiceAccounts, RoleBindings
- Security context and network security policy

8. Advanced topics

- Helm
- Custom resources and operators

Quick Recap: Pod: yaml (minimal)

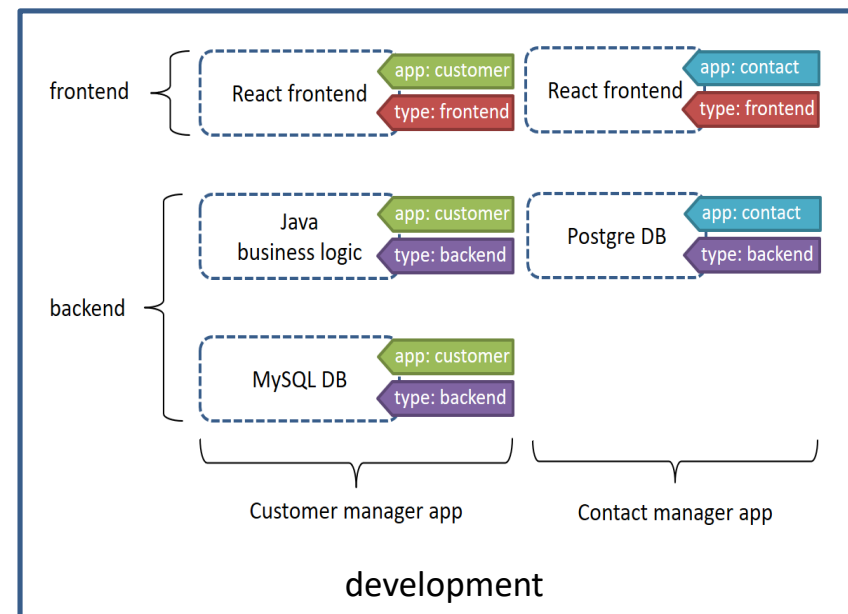
```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: www
  name: www
spec:
  containers:
  - image: nginx:1.17
    name: www
```

Pod: yaml (more realistic)

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: www
  namespace: development
  annotations:
    registry: "https://hub.docker.com/"
  name: www
spec:
  containers:
  - image: nginx:1.17
    name: www
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
    imagePullPolicy: Always
```

arbitrary non-identifying
metadata to objects

two primitives that can help you
define the concept of an application



Pod: yaml (more realistic)

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: www
  namespace: development
  annotations:
    registry: "https://hub.docker.com/"
  name: www
spec:
  containers:
  - image: nginx:1.17
    name: www
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
    imagePullPolicy: Always
```

Labels	Annotations
key/value pairs	key/value pairs
usually for identification and selection (for K8s queries)	non-identifying information, usually for metadata required by the object itself (for tools and libs)
e.g. app: frontend	e.g. nginx.ingress.kubernetes.io/canary: "true"

Pod: yaml (more realistic)

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:
```

```
  labels:
```

```
    run: www
```

```
  namespace: development
```

```
  annotations:
```

```
    registry: "https://hub.docker.com/"
```

```
  name: www
```

```
spec:
```

```
  containers:
```

```
  - image: nginx:1.17
```

```
    name: www
```

```
  resources:
```

```
    requests:
```

```
      memory: "64Mi"
```

```
      cpu: "250m"
```

```
    limits:
```

```
      memory: "128Mi"
```

```
      cpu: "500m"
```

```
  imagePullPolicy: Always
```

arbitrary non-identifying
metadata to objects

each object has it,
unique for that type of resource:

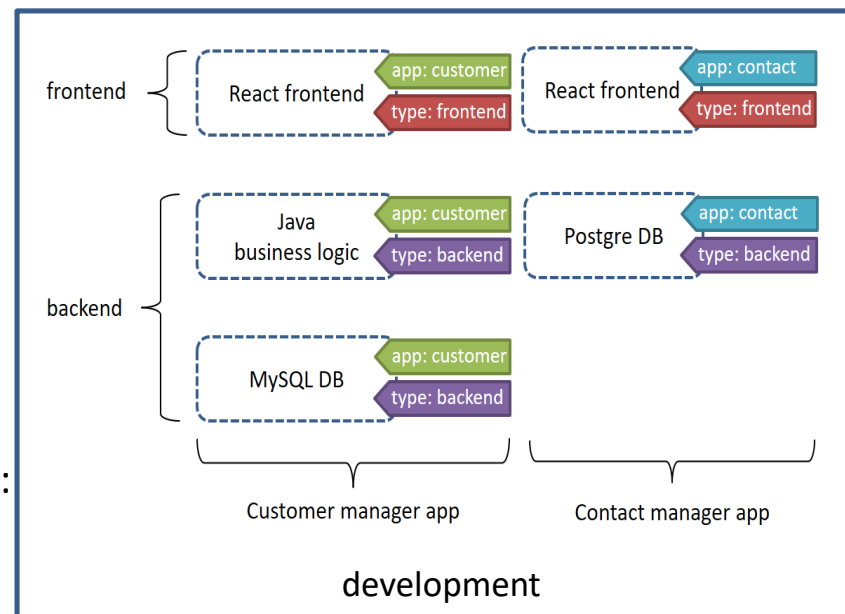
- only one Pod can have "name: www"
- but a Service or container can have "name: www" as well

- only start if at least this amount is available
- Scheduler choose a Node based on this

- will be terminated if starts to use more
- Kubelet enforces the limit

[IfNotPresent, Never]

two primitives that can help you
define the concept of an application



Further config parameters:

- Volumes
- Environment variables
- NodeAffinity
- Probes
- Security config
- ...

- Every pod consumes resources (CPU, Memory, Storage)
- Scheduler decides where to place a Pod, based on resources (and other factors):
 - If there isn't enough resources on a Node, the Scheduler will choose another one
- Pod can have requirements and limits:

```
resources:
```

```
  requests:
```

```
    memory: 64Mi
```

```
    cpu: 250m
```

```
  limits:
```

```
    memory: 128Mi
```

```
    cpu: 500m
```

- cpu unit:

- one cpu = 1000 millicores
- one cpu = 1 vCPU/Core for cloud providers and 1 hyperthread on bare-metal Intel processors

- memory:

- more straightforward: K8s accepts both SI notation (K,M,G,T,P,E) and Binary notation (Ki,Mi,Gi,Ti,Pi,Ei)
- 256MB, you can assign 268.4M (SI notation) or 256Mi (Binary notation).

Pod: Resources

- Request: allowed for a container to use more resource
- Limit: not allowed to use more resource
 - If uses more memory: will be killed by OOMKilling message
 - If uses more cpu: it won't be killed! It will be throttled.
 - In reality, CPU is measured as a function of time.
 - 1 CPU limit: the app runs up to 1 CPU second, every second.
 - With one thread 1 CPU will be consumed / sec
 - With two threads 1 CPU will be consumed / 0.5 sec
 - What happens for the other 0.5 sec? The process has to wait for the next slot available.
- Also important to note for CPU usage:
 - When containers compete for CPU then they compare their CPU shares and increase their usage accordingly
 - E.g.: let's have a 2 core Node with three containers that request: 1vCPU, 2vCPU, 3vCPU
the CPU time is divided as follows:
 - 1000 share (or 33.33% CPU) to the first container.
 - 2000 shares (or 66.66% CPU) to the second container.
 - 3000 shares (or 99.99% CPU) to the third container.

- How K8s knows about the Pod status?
- Kubelet sends probes to containers
 - All succeeded? --> Pod status = Succeeded
 - Any failed? --> Pod status = Failed
 - Any running? --> Pod status = Running
- Additional probes that user can define (since she knows her application):
 - Liveness:
 - Failed? Kubelet assumes container is dead
 - Restart policy will kick in --> [Always, OnFailure, Never]
 - Use-case: Kill and restart if probe fails (1. Add liveness probe, 2. Specify restartPolicy [Always, OnFailure])
 - Readiness:
 - Ready to service requests?
 - Failed? Endpoint object will remove pod from services
 - Use-case: Send traffic only after probe succeeds (1. Pod goes live, 2. Only accept traffic after readiness probe OK)
 - StartupProbe: to cover a very long initialization period (then let liveness do more frequent queries)

Types: Exec script, HTTP or TCP

Container lifecycle hooks:

- **PostStart:**
 - Called immediately after container created
 - However, there is no guarantee that the hook will execute before the container ENTRYPOINT
 - No parameters are passed to the handler
- **PreStop:**
 - Called immediately before a container is terminated
 - Blocking, it is synchronous, so it must complete before the call to delete the container can be sent
 - No parameters are passed to the handler

=> Hook handlers:

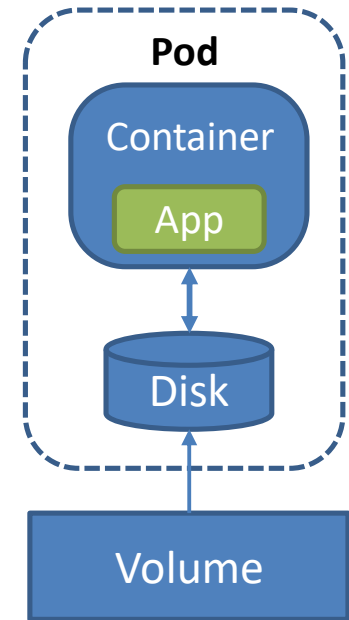
- Execute a script or call a HTTP request against a specific endpoint on the container
- Delivery is at least once: the handler needs to handle this correctly

Volumes:

Storage that lasts longer than the Container(s) in a Pod

Solves two problems:

1. On-disk files in a Container are ephemeral
 - after restart starts with a clean state
2. Often necessary to share files between Containers in a Pod:
 - but every Container needs to mount it independently



volumeMounts:

- name: config-vol
mountPath: /etc/config

volumes:

- name: config-vol
hostPath:
path: /data

Volumes:

Storage that lasts longer than the Container(s) in a Pod

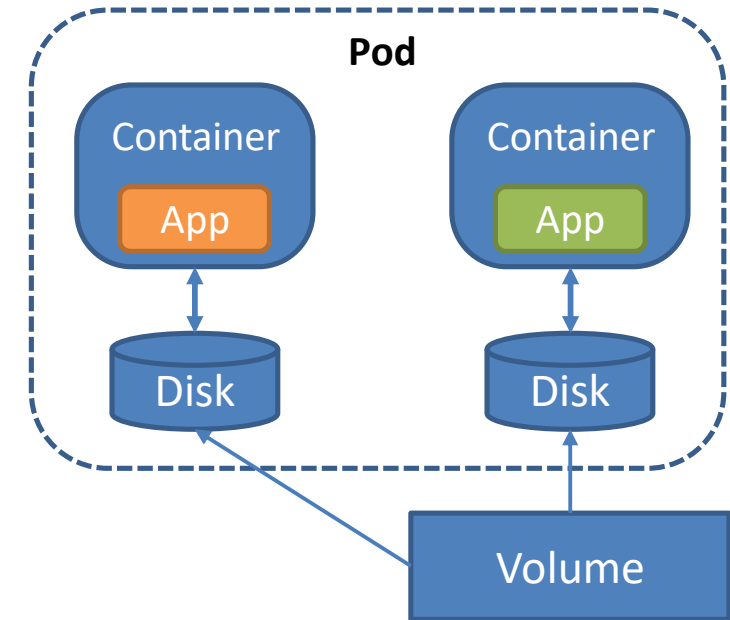
Solves two problems:

1. On-disk files in a Container are ephemeral
 - after restart starts with a clean state
2. Often necessary to share files between Containers in a Pod:
 - but every Container needs to mount it independently

Many types, e.g.:

- emptyDir: exists as long as that Pod is running
 - hostPath: file or directory from the host node's filesystem
 - persistentVolume / persistentVolumeClaim
 - gcePersistentDisk
 - downwardAPI: make downward API data available to applications
 - configMap
 - Secret
- } Used for configuration purposes

(More details on storage in lecture 6th)



volumeMounts:

- name: config-vol
mountPath: /etc/config

volumes:

- name: config-vol

configMap:

- name: log-config
- items:
 - key: log_level
path: log_level

Pod: Volumes and App Configuration

App configuration from:

1. Command and arguments

```
spec:
  containers:
  - name: hello
    image: busybox
    command: ['sh', '-c', 'echo "Hello!""']
```

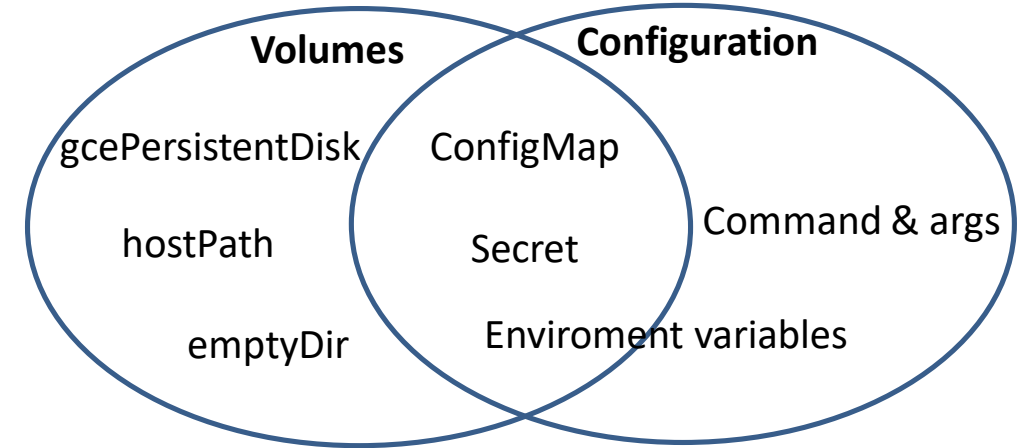
2. Environment variables

```
spec:
  containers:
  - name: envvar-demo-container
    image: gcr.io/google-samples/node-hello:1.0
    env:
    - name: DEMO_GREETING
      value: "Hello from the environment"
```

3. Volumes

```
volumes:
- name: host-data
  hostPath:
    path: /tmp/html
```

```
spec:
  volumeMounts:
  - name: host-data
    mountPath: /usr/share/nginx/html
```



per container

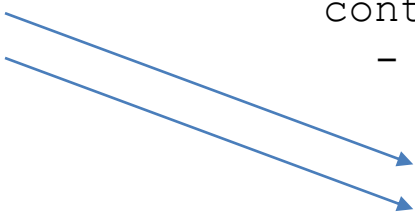
per pod

- `command` and `arguments` fields
- The `command` and `arguments` cannot be changed after the Pod is created.
- Override the default `command` and `arguments` provided by the container image
- Dockerfile vs. YAML configuration:

```
FROM ubuntu
ENTRYPOINT ["sleep"]
CMD ["10"]
```

spec:

```
  containers:
  - name: ubuntu-sleeper
    image: ubuntu-sleeper
    command: ["sleep-deeper"]
    args: ["500"]
```



The diagram shows two blue arrows. The first arrow points from the `ENTRYPOINT ["sleep"]` line of the Dockerfile to the `command: ["sleep-deeper"]` line of the Kubernetes spec. The second arrow points from the `CMD ["10"]` line of the Dockerfile to the `args: ["500"]` line of the Kubernetes spec.

- Rules of overriding the default Entrypoint and Cmd:

Image Entrypoint	Image Cmd	Container command	Container args	Command run
[/ep-1]	[foo bar]	<not set>	<not set>	[ep-1 foo bar]
[/ep-1]	[foo bar]	[/ep-2]	<not set>	[ep-2]
[/ep-1]	[foo bar]	<not set>	[zoo boo]	[ep-1 zoo boo]
[/ep-1]	[foo bar]	[/ep-2]	[zoo boo]	[ep-2 zoo boo]

- User defined variables that are available for the Container inside the Pod
- Usually related to the application logic inside the Container
- Need to define individually
- Multiple options:

```
env:  
- name: DEMO_GREETING  
  value: "Hello from the environment"
```

```
env:  
- name: SECRET_USERNAME  
  valueFrom:  
    secretKeyRef:  
      name: mysecret  
      key: username
```

```
env:  
- name: LOG_LEVEL  
  valueFrom:  
    configMapKeyRef:  
      name: env-config  
      key: log_level
```

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: mysql  
  labels:  
    db: mysql  
spec:  
  containers:  
  - image: mysql:5.6  
    name: mysql  
    env:  
    - name: MYSQL_ROOT_PASSWORD  
      valueFrom:  
        secretKeyRef:  
          name: mysql-pass  
          key: password  
    ports:  
    - containerPort: 3306  
      name: mysql
```

Pod: Configuration – Secrets and ConfigMaps

- The most intuitive ways to define configuration: `Secret` or `ConfigMap`
- Instead of hardcoding config into the app lets have it from `Secret` or from `ConfigMap`
- Can be used in multiple ways:
 - Mount key-value pairs as env variables
 - Mount some key-value pairs as volumes
- Difference:
 - `ConfigMap` is for non-secret configuration data
 - `Secret` is for things which are actually secret like API keys, credentials, etc.
 - Only encoded **not encrypted**
 - base 64 encoding
 - secret is only sent to a node if a Pod on that node requires it
 - Kubelet stores the it in a tmpfs so that the secret is not written to disk storage
 - Once Pod is deleted Kubelet will delete its local copy of the secret data as well
- Similarity:
 - attached as env variables cannot be refreshed into the pod
 - attached as volume can be refreshed into the pod (after change applied)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  log_level: DEBUG
  environment: DEV
```

```
apiVersion: v1
kind: Secret
metadata:
  name: app-config
data:
  DB_HOST: mysql
  DB_USER: root
  DB_PASSWORD: password
```


- The most intuitive ways to define configuration: `Secret` or `ConfigMap`
- Instead of hardcoding config into the app lets have it from `Secret` or from `ConfigMap`
- Can be used in multiple ways:
 - Mount key-value pairs as env variables
 - Mount some key-value pairs as volumes
- Difference:
 - `ConfigMap` is for non-secret configuration data
 - `Secret` is for things which are actually secret like API keys, credentials, etc.
 - Only encoded **not encrypted**
 - base 64 encoding
 - secret is only sent to a node if a Pod on that node requires it
 - Kubelet stores the it in a tmpfs so that the secret is not written to disk storage
 - Once Pod is deleted Kubelet will delete its local copy of the secret data as well
- Similarity:
 - attached as env variables cannot be refreshed into the pod
 - attached as volume can be refreshed into the pod (after change applied)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  log_level: DEBUG
  environment: DEV
```

```
apiVersion: v1
kind: Secret
metadata:
  name: app-config
data:
  DB_HOST: MTIzNDU2Cg==
  DB_USER: cm9vdAo=
  DB_PASSWORD: cGFzc3dvcmQK
```

Pod: Presets

- Objects for injecting certain information into pods at creation time
- Can include `secrets`, `volumes`, `volume mounts`, and `environment variables`
- An admission controller needs to be enabled in the API server config for PodPresets
- Selectors can be used for defining the related pods
- When a Pod is created:
 - Checks the existing PodPresets
 - Try to inject/merge them with Pod description
 - On error throws error and the Pod *will be created without* the presets
 - Annotates the resulting modified Pod spec:

```
podpreset.admission.kubernetes.io/podpreset-<preset name>: "<resource version>"
```

- For `env`, `envFrom`, and `volumeMounts` modifies all container spec field
- For `volumes` modifies only the pod spec field
- Can be disabled for certain pods with an annotation:
 - `podpreset.admission.kubernetes.io/exclude: "true"`

```
apiVersion:
  settings.k8s.io/v1alpha1
kind: PodPreset
metadata:
  name: allow-database
spec:
  selector:
    matchLabels:
      role: frontend
  env:
    - name: DB_PORT
      value: "6379"
  volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
```

- A security context defines privilege and access control settings for a Pod or Container
- Several aspects, but maybe the two most used are:

- Running as privileged or unprivileged:

```
securityContext:  
  runAsUser: 1000  
  runAsGroup: 3000
```

- Linux Capabilities:

- Give a process some privileges, but not all the privileges of the root user
- By default a container run with root user but with not all root capabilities:
 - Usually this is a subset of the host root user's capabilities:
 - » Directory access, usage of specific commands (e.g. kill), setting groupID, performing network related operations
 - » Can be extended or narrowed

```
securityContext:  
  capabilities:  
    add: ["NET_ADMIN", "SYS_TIME"]  
    drop: ["KILL"]
```

- These can be set at the Pod or Container level as well

Policies: Indirect Pod configuration

- Enable indirect Pod configuration and management
- Rules and configuration settings that is applied to a Pod but not part of its original YAML
- Three types:
 - ResourceQuota: restrict resource consumption and creation on a namespace basis
 - LimitRange: constrain resource allocations to Pods or Containers in a namespace
 - SecurityPolicies: controls security sensitive aspects of the pod specification

- A tool for administrators to make sure of fair resource sharing in the cluster
- Provides constraints that limit aggregate resource consumption per namespace
 - quantity of objects that can be created in a namespace by type (e.g. Pod, Service, etc.)
 - total amount of compute resources (CPU, memory) that may be consumed by resources
- When resources created quota system tracks usage to ensure it does not exceed the limits
- E.g.
 - In a 16 core + 32 GiB cluster:
 - let team A use 20 GiB and 10 cores, let B use 10GiB and 4 cores
 - hold 2GiB and 2 cores in reserve for future allocation
 - Limit the "testing" namespace to 1 core and 1GiB RAM. Let the "production" namespace use any amount
- Changes to ResourceQuota will not affect already created resources.

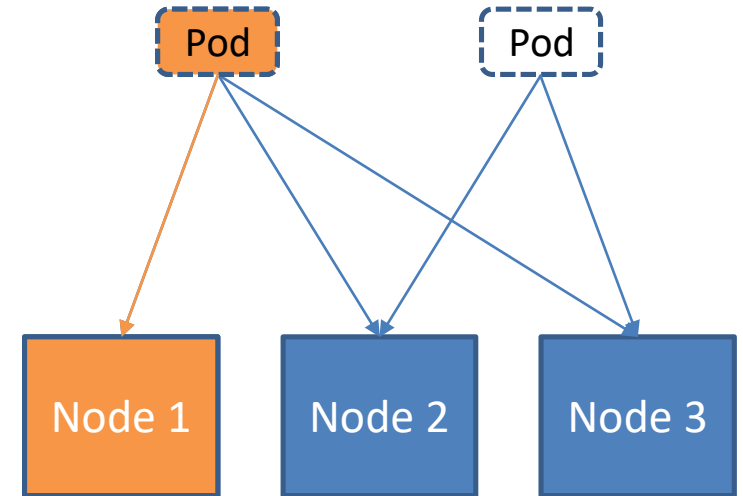
- Even with ResourceQuota on a namespace:
 - a Pod or Container can consume as much CPU and memory as defined by the namespace's resource quota
 - so one Pod or Container could monopolize all available resources
 - LimitRange is a policy to constrain resource allocations (to Pods or Containers) in a namespace.
- Provides constraints for the namespaces that:
 - Enforce minimum and maximum compute resources usage
 - Enforce minimum and maximum storage request per PersistentVolumeClaim
 - Enforce a ratio between request and limit for a resource
 - Set default request/limit for compute resources and automatically inject them to Containers at runtime
- LimitRange validations occurs only at Pod Admission stage, not on Running Pods.
- E.g.
 - In a 2 node, 8 GiB RAM, 16 cores cluster
 - constrain Pods in a namespace to request $100\text{m} < \text{CPU} < 500\text{m}$ and request $200\text{Mi} < \text{Memory} < 600\text{Mi}$
- Changes to LimitRange will not affect already created resources.

Pod: Security Policies

- Cluster-level resource that controls security sensitive aspects of the pod specification
- Define a set of conditions that a pod must run with in order to be accepted into the system
- Some example of control aspects:
 - Running of privileged containers:
 - allows nearly all the same access as processes running on the host
 - Provide a list of allowed volume types:
 - e.g. whether a pod can use ConfigMaps or PersistentVolumeClaims
 - Requiring the use of a read only root file system
 - Users and groups:
 - Controls which user and group ID the containers are run with

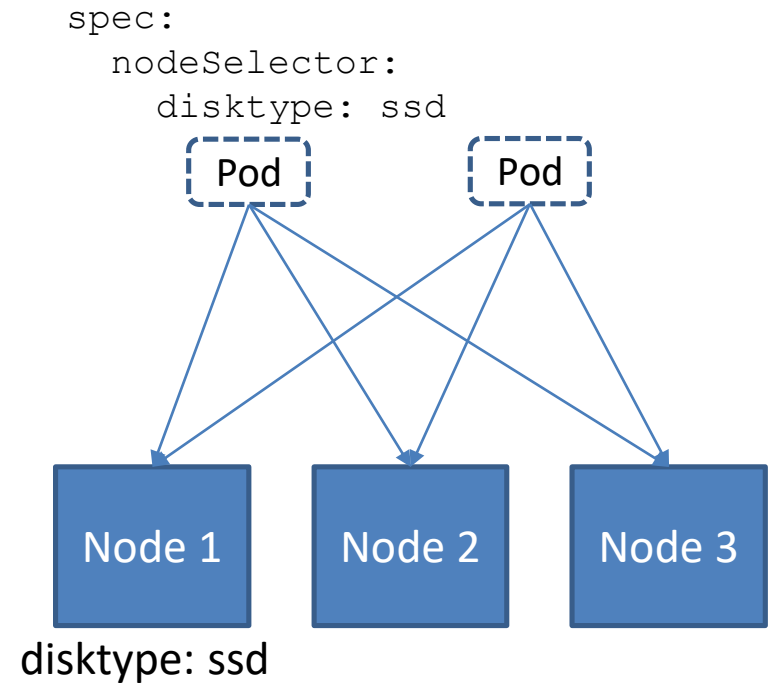
1. Taints and tolerations:

- By default a Pod can be assigned to any Node
- Taints:
 - can be assigned to Nodes (just like labels)
 - if a Node is tainted no pod can be scheduled there
- Tolerations:
 - a Pod with toleration to the taint can be scheduled to the Node
 - Important: it does not bound the Pod to the tainted Node!
- A taint + toleration:
 - Good practice to reserve Node(s) for certain Pod(s) by keeping the rest of the Pods out of the given Node(s)



2. Node selectors:

- constrain a Pod to *only* be able to run on particular Node(s)
- works based on Node labels
- usually a two-step process:
 - Assign label to one or more Nodes, e.g. “size=large”
 - Add nodeSelector part to the Pod YAML file
- Does not keep out other Pods from the Node!
- Other Pods may or may not be placed on that Node
- Good practice for simple cases (e.g. place Pod on certain Nodes)
 - but doesn't have enough expression power for advanced cases (e.g. OR or NOT selectors)

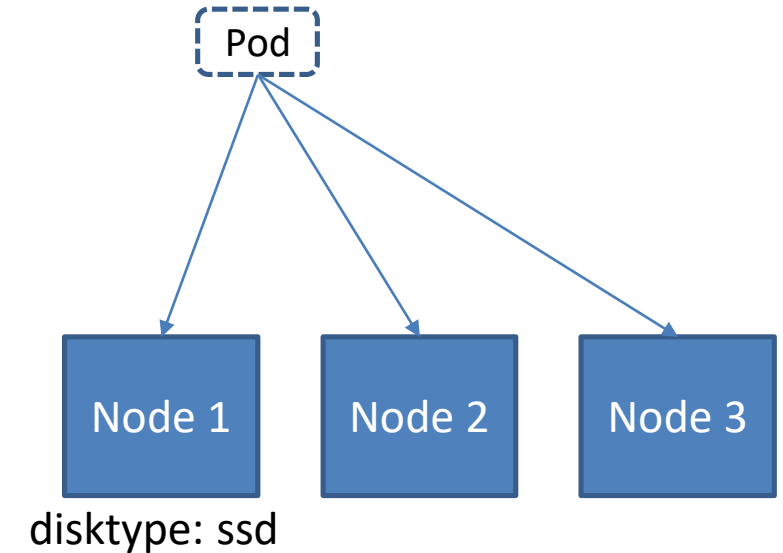


3. Affinity or AntiAffinity:

- Where nodeSelector is not enough
- Enables to write more complex cases
- However, writing it is more complex as well
- The previous NodeSelector example would look like this:

```
affinity:
  podAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: disktype
          operator: In
          values:
          - ssd
```

- However, now we can describe OR or NOT operators:
 - OR: values: [ssd, hdd]
 - NOT: operator: NotIn
- What happens when no such label is present during scheduling or when it is removed during execution?
 - requiredDuringSchedulingIgnoredDuringExecution
 - preferredDuringSchedulingIgnoredDuringExecution



Pod: Influencing Pod placement

An example:

- Let's have 4 Nodes and 4 Pods

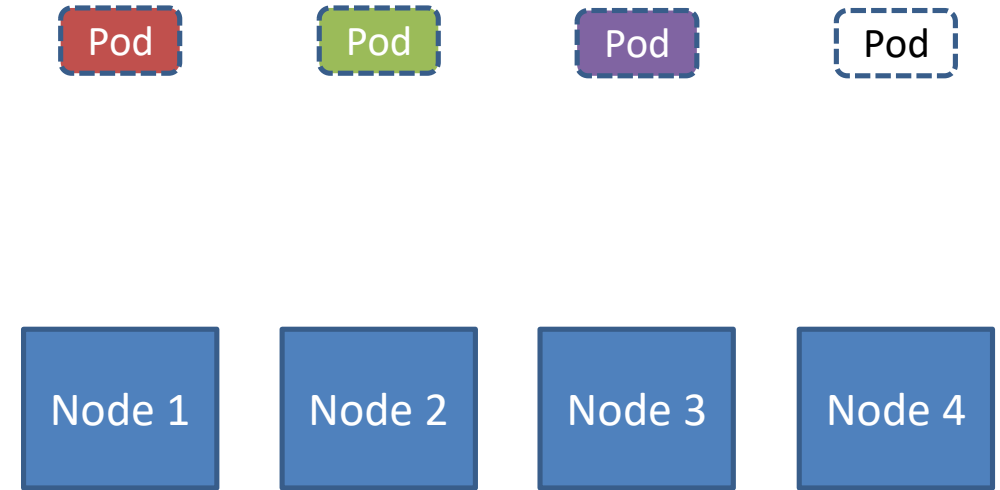
- Desired topology:

- red -> Node 1

- green -> Node 2

- purple -> Node 3

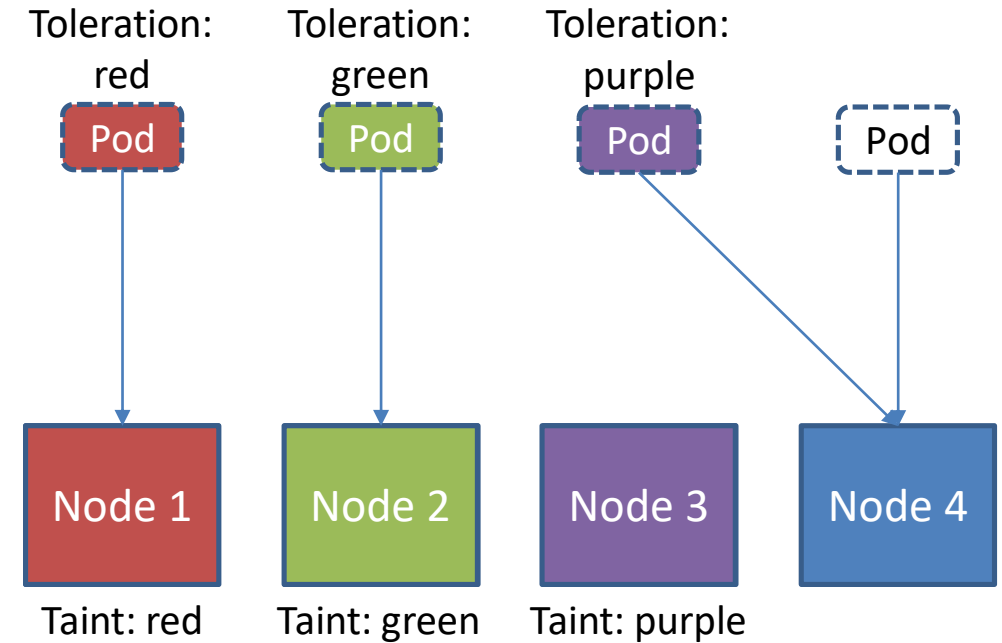
- Node 1,2 and 3 should be reserved for only these pods, respectively



Pod: Influencing Pod placement

An example:

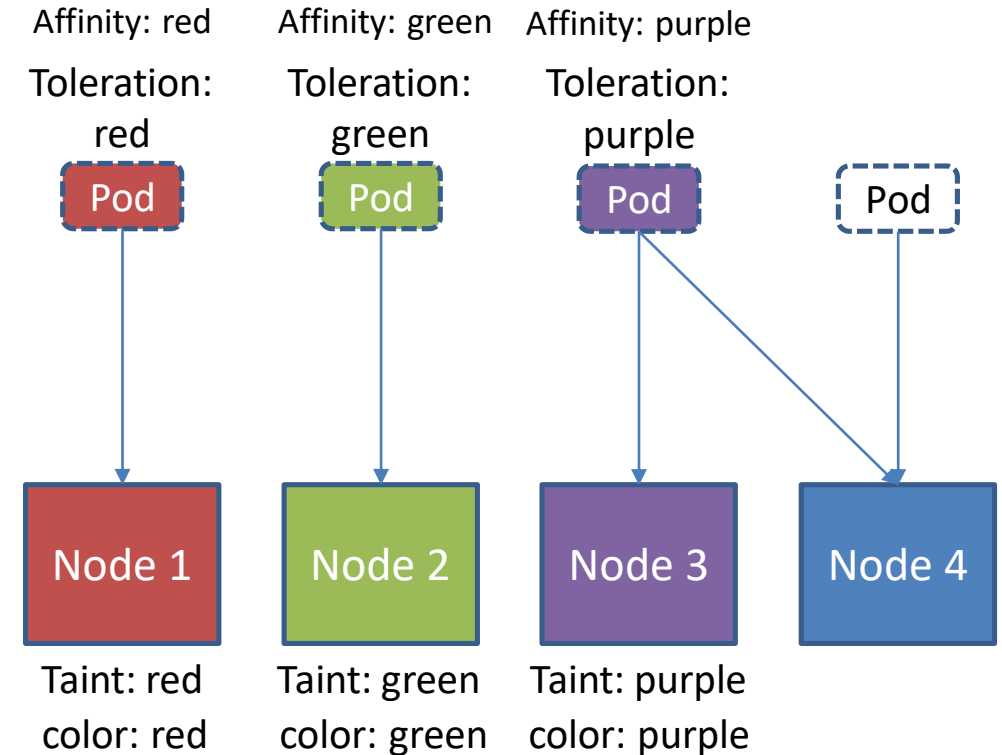
- Let' use taints and tolerations:
 - red and green end up on the proper nodes
 - however, purple ends up on Node 4
 - because this way we can only keep out other pods for sure



Pod: Influencing Pod placement

An example:

- Let' use taints and tolerations:
 - red and green end up on the proper nodes
 - however, purple ends up on Node 4
 - because this way we can only keep out other pods for sure
- Let' add NodeAffinities:
 - Labeling nodes
 - Add affinity definitions in the YAML
- Every colored Pod end up on the intended Node



Troubleshooting

Basically, there are two types:

- Troubleshooting the cluster
- Troubleshooting the application

Cluster:

- Checking the topology of the cluster (Master and Worker nodes)
- Checking the control plane components (API Server, Controller Manager, Scheduler, Etcd)
- Checking the data plane components (Kubelet, Kube-proxy, Container runtime)

Application:

- Debugging Pods
- Debugging Replication Controllers
- Debugging Services

Troubleshooting: Cluster

1. Checking whether all nodes are registered correctly:

```
kubectl get nodes -> are all in Ready state?
```

2. Checking nodes one-by-one:

```
kubectl cluster-info dump
```

- dumps everything to stdout by default
- dumps things in the 'kube-system' namespace but can be specified
- dumps the logs of all of the pods in the cluster

3. Logging into the nodes and check logs:

```
cat /var/log/pods/* (proxy, coreDNS, etcd, scheduler, etc. depending on Master/Worker)
```

```
systemctl status kubelet.service
```

General overview of cluster failure modes:

- Root causes:
 - VM(s) shutdown
 - Network partition within cluster, or between cluster and users
 - Crashes in Kubernetes software
 - Data loss or unavailability of persistent storage (e.g. GCE PD or AWS EBS volume)
 - Operator error, for example misconfigured Kubernetes software or application software

General overview of cluster failure modes:

- Specific scenarios:
 1. Apiserver VM shutdown or apiserver crashing:
 - unable to stop, update, or start new pods, services, replication controllers
 - existing pods and services should continue to work normally, unless they depend on the Kubernetes API
 2. Apiserver backing storage lost:
 - apiserver should fail to come up
 - kubelets will not be able to reach it but will continue to run the same pods and provide the same service proxying
 - manual recovery or recreation of apiserver state necessary before apiserver is restarted
 3. Supporting services (scheduler, etc) VM shutdown or crashes:
 - currently those are colocated with the apiserver, and their unavailability has similar consequences as apiserver
 - they do not have their own persistent state

Troubleshooting: Cluster

General overview of cluster failure modes:

- Specific scenarios:
 4. Individual node (VM or physical machine) shuts down:
 - pods on that Node stop running
 5. Network partition (assuming the master VM ends up in partition A):
 - partition A thinks the nodes in partition B are down
 - partition B thinks the apiserver is down
 6. Kubelet software fault:
 - crashing kubelet cannot start new pods on the node
 - kubelet might delete the pods or not
 - node marked unhealthy
 - replication controllers start new pods elsewhere
 7. Cluster operator error
 - loss of pods, services, etc
 - loss of apiserver backing store
 - users unable to read API

General overview of cluster failure modes:

- Mitigations:
 1. Action: Use IaaS provider's automatic VM restarting feature for IaaS VMs
 - Mitigates: Apiserver VM shutdown or apiserver crashing
 - Mitigates: Supporting services VM shutdown or crashes
 2. Action: Use IaaS providers reliable storage (e.g. GCE PD or AWS EBS volume) for VMs with apiserver+etcd
 - Mitigates: Apiserver backing storage lost
 3. Action: Use high-availability configuration
 - Mitigates: Control plane node shutdown or control plane components (scheduler, API server, etc.) crashing
 - Will tolerate one or more simultaneous node or component failures
 - Mitigates: API server backing storage (i.e., etcd's data directory) lost
 - Assumes HA (highly-available) etcd configuration
 4. Action: Snapshot apiserver PDs/EBS-volumes periodically
 - Mitigates: Apiserver backing storage lost
 - Mitigates: Some cases of operator error
 - Mitigates: Some cases of Kubernetes software fault

Troubleshooting: Application

1. Debugging Pods

```
kubectl get pods -o wide --show-labels
```

```
kubectl describe pods <POD-NAME>
```

- Look at the state of the containers in the pod. Are they all Running? Have there been recent restarts?
- If a Pod is stuck in
 - Pending
 - it means that it can not be scheduled onto a node:
 - usually resource allocation problem, e.g. CPU, memory, Secret, ConfigMap, Volume (describe it)
 - Waiting
 - scheduled to a node, but it can't run on that machine
 - usually image pulling failure (describe it)
 - Crashing
 - Inspect logs
 - » `kubectl logs <POD-NAME> <CONTAINER-NAME>`
 - » `kubectl logs --previous <POD-NAME> <CONTAINER-NAME>`
 - Debugging with container exec (if Pod contains such utilities):
 - » `kubectl exec <POD-NAME> -c <CONTAINER-NAME> -- <CMD> <ARG1> <ARG2> ... <ARGN>`
 - Debugging with ephemeral debug container (1.18 alpha):
 - » `kubectl alpha debug -it ephemeral-demo --image=busybox --target=ephemeral-demo`
 - » Target the namespace of another container

Troubleshooting: Application

1. Debugging Pods

– My pod is running but not doing what is expected:

- Check YAML: `kubectl get pod <POD-NAME> -o yaml`
- Compare the intended and the actual one that is run by the API server
 - there can be differences, e.g. by using LimitRange objects

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-min-max
  namespace: restricted-pods
spec:
  limits:
    - max:
        cpu: "800m"
      min:
        cpu: "200m"
      type: Container
```



```
apiVersion: v1
kind: Pod
metadata:
  name: restricted-pod
spec:
  containers:
    - name: nginx
      image: nginx
```



```
apiVersion: v1
kind: Pod
metadata:
  name: restricted-pod
spec:
  containers:
    - name: nginx
      image: nginx
      resources:
        limits:
          cpu: "800m"
        requests:
          cpu: "500m"
```

2. Debugging ReplicaSets

- Fairly straightforward:
 - Either can create the pods or not (if not, debug pods)
 - `kubectl describe rs <REPLICASET-NAME>`

3. Debugging Services:

- Verify that there are endpoints:
 - `kubectl get endpoints <SERVICE-NAME>`
 - for every Service the apiserver makes an `endpoints` object
 - make sure that the expected replicas match the number of IP addresses of the endpoint
- If there are missing endpoints, try listing pods with the service labels:
 - `kubectl get pods --selector=name=nginx,type=frontend`
 - If list of pods matches expectations:
 - Verify that the pod's `containerPort` matches up with the Service's `targetPort`
- Network traffic is not forwarded, connection is immediately dropped:
 - There are endpoints (if not, debug endpoints):
 - Verify direct connection to pods with Pod IP (since its likely a proxy error)
 - Make sure that application serves on the correct port (`containerPort`), since K8s doesn't do port remapping