Labels

Labels

- Labels are key/value pairs that can be attached to objects
    - Labels are like **tags** in AWS or other cloud providers, used to tag resources
- You can **label** your **objects**, for instance your pod, following an organizational structure
    - **Key**: environment - **Value**: dev / staging / qa / prod
    - **Key**: department - **Value**: engineering / finance / marketing
- In our previous examples I already have been using labels to tag pods:

```
metadata:
  name: nodehelloworld.example.com
  labels:
    app: helloworld
```

Labels

- Labels are **not unique** and **multiple labels** can be added to one object
- Once labels are attached to an object, you can use filters to narrow down results
    - This is called **Label Selectors**
- Using Label Selectors, you can use **matching expressions** to match labels
    - For instance, a particular pod can only run on a node labeled with “environment” equals “development”
    - More complex matching: “environment” in “development” or “qa”

Node Labels

- You can also use labels to tag **nodes**
- Once nodes are tagged, you can use **label selectors** to let pods only run on **specific nodes**
- There are **2 steps** required to run a pod on a specific set of nodes:
  - First you **tag** the node
  - Then you add a **nodeSelector** to your pod configuration

Node Labels

- First step, add a label or multiple labels to your nodes:

```
$ kubectl label nodes node1 hardware=high-spec
$ kubectl label nodes node2 hardware=low-spec
```

- Secondly, add a pod that uses those labels:

```
apiVersion: v1
kind: Pod
metadata:
  name: nodehelloworld.example.com
  labels:
    app: helloworld
spec:
  containers:
  - name: k8s-demo
    image: wardviaene/k8s-demo
    ports:
     - containerPort: 3000
  nodeSelector:
    hardware: high-spec
```

Demo

Node Selector using labels

Health Checks

Health checks

- If your application **malfunctions**, the pod and container can still be running, but the application might not work anymore
- To **detect** and **resolve** problems with your application, you can run **health checks**
- You can run 2 different type of health checks
    - Running a **command** in the container **periodically**
    - Periodic checks on a **URL** (HTTP)
- The typical production application behind a load balancer should always have **health checks** implemented in some way to ensure **availability** and **resiliency** of the app

Health checks

- This is how a health check looks like on our example container:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nodehelloworld.example.com
  labels:
    app: helloworld
spec:
  containers:
  - name: k8s-demo
    image: wardviaene/k8s-demo
    ports:
     - containerPort: 3000
    livenessProbe:
      httpGet:
        path: /
        port: 3000
      initialDelaySeconds: 15
      timeoutSeconds: 30
```

Demo

Performing health checks

Readiness Probe

Readiness Probe

- Besides livenessProbes, you can also use **readinessProbes** on a container within a Pod
- **livenessProbes**: indicates whether a container is **running**
    - If the check fails, the container will be restarted
- **readinessProbes**: indicates whether the container is **ready to serve** requests
    - If the check fails, the container **will not be restarted**, but **the Pod's IP address will be removed from the Service**, so it'll not serve any requests anymore

Readiness Probe

- The **readiness** test will make sure that **at startup**, the pod will only receive traffic when the test succeeds

- You can use these probes **in conjunction**, and you can configure different tests for them

- If your container always exits when something goes wrong, you don't need a livenessProbe

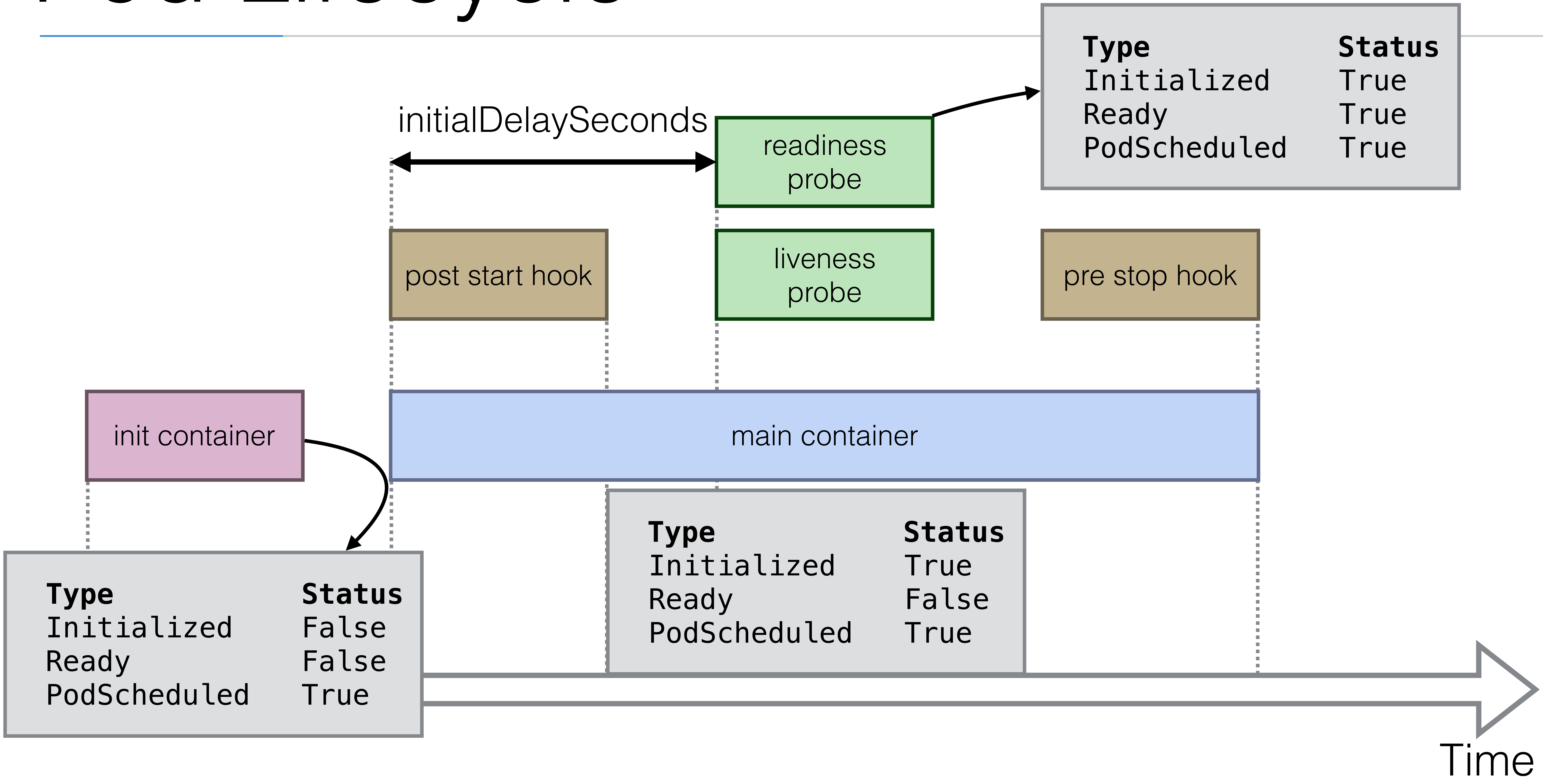
- In general, you configure **both** the livenessProbe and the readinessProbe

Demo

Performing health checks (readinessProbe)

Pod Lifecycle

Pod Lifecycle



Web UI

Web UI

- Kubernetes comes with a **Web UI** you can use instead of the kubectl commands

- You can use it to:

    - Get an **overview** of running applications on your cluster

    - **Creating** and **modifying** individual Kubernetes **resources** and **workloads** (like kubectl create and delete)

    - Retrieve information on the **state** of **resources** (like kubectl describe pod)

Web UI

- In general, you can access the kubernetes Web UI at https://<kubernetes-master>/ui

- If you cannot access it (for instance if it is not enabled on your deploy type), you can install it manually using:

```
$ kubectl create -f https://rawgit.com/kubernetes/dashboard/master/src/deploy/kubernetes-dashboard.yaml
```

- If a password is asked, you can retrieve the password by entering:

```
$ kubectl config view
```

Web UI

- If you are using minikube, you can use the following command to launch the dashboard:

```
$ minikube dashboard
```

- Or if you just want to know the url:

```
$ minikube dashboard --url
```

Demo

Web UI

Advanced topics

Pod Presets

Pod Presets

- Pod presets can **inject information into pods at runtime**
  - Pod Presets are used to **inject Kubernetes Resources** like Secrets, ConfigMaps, Volumes and Environment variables
- Imagine you have 20 applications you want to deploy, and they all need to get a specific credential
  - You can edit the 20 specifications and add the credential, or
  - You can use presets to create 1 Preset object, which will inject an environment variable or config file **to all matching pods**
- When **injecting** Environment variables and VolumeMounts, the Pod Preset will **apply the changes** to **all containers** within the pod

Pod Presets

- This is an example of a Pod Preset

```
apiVersion: settings.k8s.io/v1alpha1   # you might have to change this after PodPresets become stable
kind: PodPreset
metadata:
  name: share-credential
spec:
  selector:
    matchLabels:
      app: myapp
  env:
    - name: MY_SECRET
      value: "123456"
  volumeMounts:
    - mountPath: /share
      name: share-volume
  volumes:
    - name: share-volume
      emptyDir: {}
```

Pod Presets

- You can use **more than one PodPreset**, they'll all be applied to matching Pods
- If there's a **conflict**, the PodPreset will **not be applied** to the pod
- PodPresets can match **zero or more Pods**
  - It's possible that no pods are currently matching, but that matching pods will be launched at a later time

Demo

Pod Presets

StatefulSets

Stateful distributed apps on a Kubernetes cluster

StatefulSets

- Pet Sets was a **new feature** starting from Kubernetes 1.3, and got renamed to StatefulSets which is stable since Kubernetes 1.9
- It is introduced to be able to run **stateful applications**:
  - That need a **stable pod hostname** (instead of podname-randomstring)
    - Your podname will have a sticky identity, using an index, e.g. podname-0 podname-1 and podname-2 (and when a pod gets rescheduled, it'll keep that identity)
  - Statefulsets allow **stateful apps stable storage** with volumes based on their ordinal number (podname-**x**)
    - **Deleting** and/or **scaling** a **StatefulSet down** will not delete the volumes associated with the StatefulSet (preserving data)

StatefulSets

- A StatefulSet will allow your stateful app to use **DNS** to find other **peers**
    - Cassandra clusters, ElasticSearch clusters, use **DNS** to find other members of the cluster
        - for example: **cassandra-0.cassandra** for all pods to reach the first node in the cassandra cluster
    - Using StatefulSet you can run for instance 3 cassandra nodes on Kubernetes named cassandra-0 until cassandra-2
    - If you wouldn’t use StatefulSet, you would get a dynamic hostname, which you wouldn’t be able to use in your configuration files, as the name can always change

StatefulSets

- A StatefulSet will also allow your stateful app to **order the startup and teardown**:
    - Instead of randomly terminating one pod (one instance of your app), you'll know which one that will go
        - When **scaling up** it goes from 0 to n-1 (n = replication factor)
        - When **scaling down** it starts with the highest number (n-1) to 0
    - This is useful if you first need to **drain** the data from a node before it can be shut down

Demo

StatefulSets - Cassandra

Daemon Sets

Daemon Sets

- Daemon Sets ensure that **every single node** in the Kubernetes cluster runs the same pod resource
  - This is useful if you want to **ensure** that a certain pod is running on every single kubernetes node
- When a node is **added** to the cluster, a new pod will be **started** automatically
- Same when a node is **removed**, the pod will not be **rescheduled** on another node

Daemon Sets

- Typical **use cases**:
  - Logging aggregators
  - Monitoring
  - Load Balancers / Reverse Proxies / API Gateways
  - Running a daemon that only needs one instance per physical instance

Daemon Sets

- This is an example Daemon Set specification:

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: monitoring-agent
  labels:
    app: monitoring-agent
spec:
  template:
    metadata:
      labels:
        name: monitor-agent
    spec:
      containers:
      - name: k8s-demo
        image: wardviaene/k8s-demo
        ports:
        - name: nodejs-port
          containerPort: 3000
```

Affinity and anti-affinity

Affinity and anti-affinity

- In a previous demo I showed you how to use nodeSelector to make sure pods get scheduled on specific nodes:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: helloworld-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: helloworld
    spec:
      containers:
      - name: k8s-demo
        image: wardviaene/k8s-demo
        […]
     nodeSelector:
        hardware: high-spec
```

Affinity and anti-affinity

- The affinity/anti-affinity feature allows you to do **more complex scheduling** than the nodeSelector and also **works on Pods**
  - The language is **more expressive**
  - You can create **rules that are not hard requirements**, but rather a **preferred rule**, meaning that the scheduler will still be able to schedule your pod, even if the rules cannot be met
  - You can create rules that take other pod labels into account
    - For example, a rule that makes sure 2 different pods will never be on the same node

Affinity and anti-affinity

- Kubernetes can do **node affinity** and **pod affinity/anti-affinity**
  - Node affinity is similar to the nodeSelector
  - Pod affinity/anti-affinity allows you to create rules **how pods should be scheduled taking into account other running pods**
  - Affinity/anti-affinity mechanism is **only relevant during scheduling**, once a pod is running, it'll need to be recreated to apply the rules again
- I'll first cover **node affinity** and will then cover pod affinity/anti-affinity

Affinity and anti-affinity

- There are currently 2 types you can use for node affinity:
  - 1) requiredDuringSchedulingIgnoredDuringExecution
  - 2) preferredDuringSchedulingIgnoredDuringExecution
- The **first one** sets a **hard requirement** (like the nodeSelector)
  - The rules must be met before the pod can be scheduled
- The **second type** will try to enforce the rule, but it will not guarantee it
  - Even if the rule is not met, the pod can still be scheduled, it's a soft requirement, a preference

Affinity and anti-affinity

```
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: env
            operator: In
            values:
            - dev
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: team
            operator: In
            values:
            - engineering-project1
  containers:
  […]
```

Affinity and anti-affinity

- I also supplied **a weighting** to the preferredDuringSchedulingIgnoredDuringExecution statement
- The **higher this weighting**, the **more weight is given to that rule**
- When scheduling, Kubernetes will score every node by summarizing the weightings per node
  - For example if you have **2 different rules with weights 1 and 5**
  - If both rules match, the node will have a score of **6**
  - If only the rule with weight **1 matches**, then the score will **only be 1**
- The node that has the **highest total score**, that's where the pod will be scheduled on

Built-in node labels

- In addition to the labels that you can add yourself to nodes, there are **pre-populated labels** that you can use:
  - kubernetes.io/hostname
  - failure-domain.beta.kubernetes.io/zone
  - failure-domain.beta.kubernetes.io/region
  - beta.kubernetes.io/instance-type
  - beta.kubernetes.io/os
  - beta.kubernetes.io/arch

Affinity and anti-affinity

Demo

Interpod Affinity and anti-affinity

Interpod Affinity and anti-affinity

- This mechanism allows you to **influence scheduling based on the labels of other pods** that are **already running** on the cluster

- **Pods belong to a namespace**, so your affinity rules will **apply to a specific namespace** (if no namespace is given in the specification, it defaults to the namespace of the pod)

- Similar to node affinity, you have **2 types** of pod affinity / anti-affinity:

  - requiredDuringSchedulingIgnoredDuringExecution

  - preferredDuringSchedulingIgnoredDuringExecution

- The **required type** creates **rules that *must* be met** for the pod to be scheduled, the **preferred type** is a “**soft**” type, and the **rules *may* be met**

Interpod Affinity and anti-affinity

- A good use case for **pod affinity** is **co-located pods**:
  - You might want that 1 pod is always co-located on the same node with another pod
  - For example you have an app that uses redis as cache, and you want to have the redis pod on the same node as the app itself
- Another use-case is to co-locate pods within the **same availability zone**

Interpod Affinity and anti-affinity

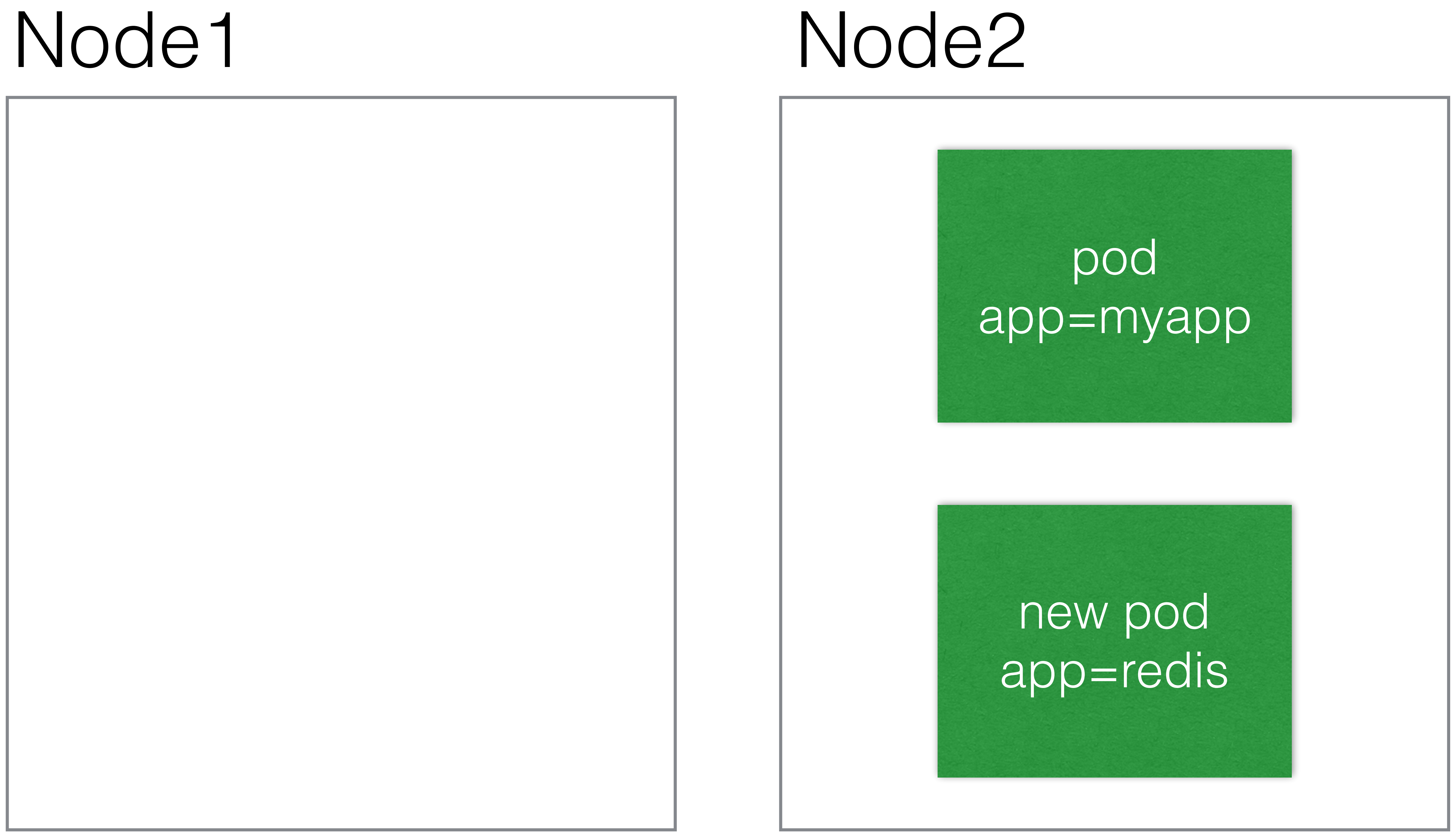
- When writing your pod affinity and anti-affinity rules, you need to specify a **topology domain**, called **topologyKey** in the rules

- The topologyKey refers to a node label

- If the affinity rule matches, the **new pod** will only be **scheduled** on **nodes** that have the **same topologyKey** value as the **current running pod**

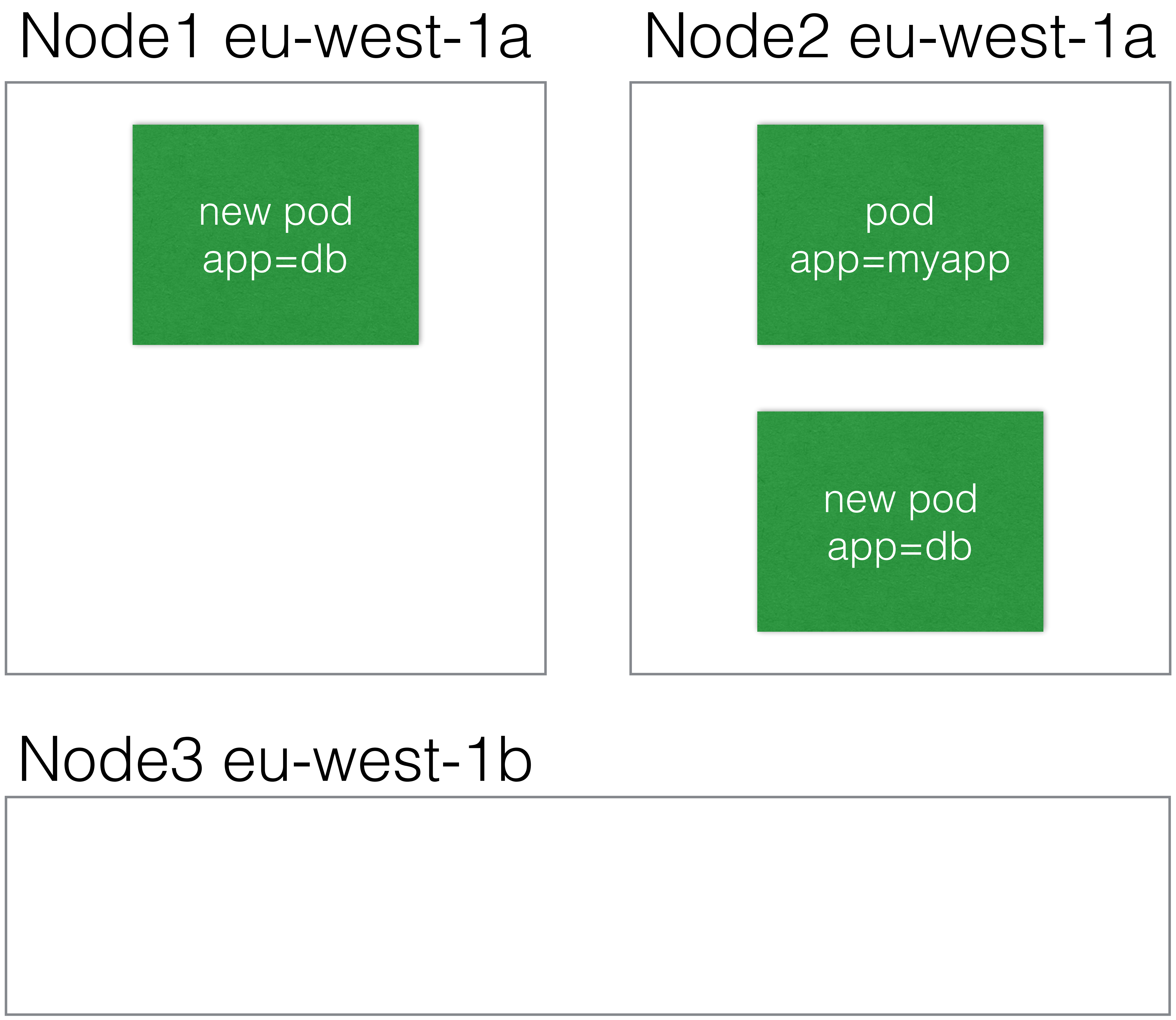
Interpod Affinity and anti-affinity


```
podAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
          - key: "app"
            operator: In
            values:
            - myapp
      topologyKey: "kubernetes.io/hostname"
```



Interpod Affinity and anti-affinity


```yaml
podAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
          - key: "app"
            operator: In
            values:
            - myapp
      topologyKey: "failure-domain.beta.kubernetes.io/zone"
```

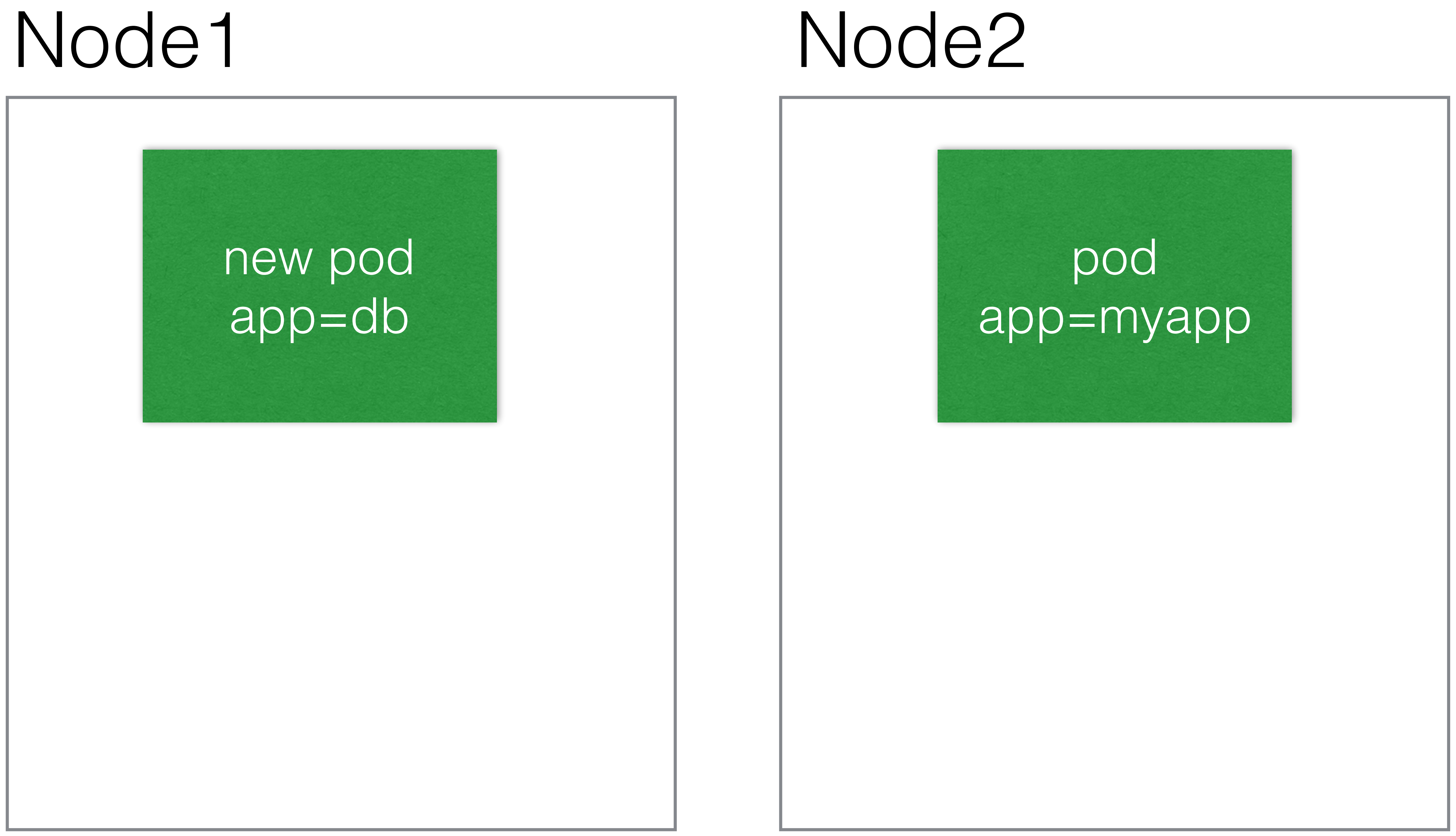


Pod Affinity and anti-affinity

- Contrary to affinity, you might want to use **pod anti-affinity**
- You can use anti-affinity to make sure a **pod is only scheduled once on a node**
    - For example you have 3 nodes and you want to schedule 2 pods, but they shouldn't be scheduled on the same node
    - Pod anti-affinity allows you to create a rule that says to **not schedule on the same host if a pod label matches**

Interpod Affinity and anti-affinity


```
podAntiAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
          - key: "app"
            operator: In
            values:
            - myapp
      topologyKey: "kubernetes.io/hostname"
```



Interpod Affinity and anti-affinity

- When writing pod affinity rules, you can use the following operators:
  - **In, NotIn** (does a label have one of the values)
  - **Exists, DoesNotExist** (does a label exist or not)
- Interpod affinity and anti-affinity currently requires a substantial amount of processing
  - You might have to take this into account if you have a lot of rules and a larger cluster (e.g. 100+ nodes)

Interpod affinity

Demo

Pod Anti-Affinity

Demo

Taints and tolerations

Taints and tolerations

- In the previous lectures I explained you the following concepts:
    - **Node** affinity (similar to the nodeSelector)
    - **Interpod** affinity / anti-affinity
- The next concept, **tolerations**, is the opposite of node affinity
    - Tolerations allow a node to **repel a set of pods**
    - **Taints mark** a node, tolerations are applied to pods to influence the scheduling of the pods

Taints and tolerations

- One use case for taints is to make sure that when you create a new pod, they're not scheduled on the master

    - **The master has a taint**: (node-role.kubernetes.io/master:NoSchedule)

- To add a new taint to a node, you can use kubectl taint:

```
kubectl taint nodes node1 key=value:NoSchedule
```

- This will make sure that **no pods will be scheduled** on node1, as long as they **don't have a matching toleration**

Taints and tolerations

- The following toleration would allow a new pod to be scheduled on the tainted node1:

```
tolerations:
- key: "key"
  operator: "Equal"
  value: "value"
  effect: "NoSchedule"
```

- You can use the following operators:
    - Equal: providing a key & value
    - Exists: only providing a key, checking only whether a key exists

Taints and tolerations

- Just like affinity, **taints** can also be a **preference** (or "soft") rather than a requirement:
  - **NoSchedule**: a hard requirement that a pod will not be scheduled unless there is a matching toleration
  - **PreferNoSchedule**: Kubernetes will try and avoid placing a pod that doesn't have a matching toleration, but it's not a hard requirement
- If the taint is applied while there are **already running pods**, these will **not be evicted**, unless the following taint type is used:
  - **NoExecute**: **evict** pods with non-matching tolerations

Taints and tolerations

- When using **NoExecute**, you can specify within your toleration **how long the pod can run** on a **tainted node** before being evicted:

```
tolerations:
- key: "key"
  operator: "Equal"
  value: "value"
  effect: "NoExecute"
  tolerationSeconds: 3600
```

- If you don't specify the tolerationSeconds, the toleration will match and the pod will keep running on the node

- In this example, the toleration will **only match for 1 hour** (3600 seconds), after that the **pod will be evicted** from the node

Taints and tolerations

- Example **use cases** are:
  - The existing node taints for **master nodes**
  - Taint nodes that are **dedicated** for a **team or a user**
  - If you have a few nodes with **specific hardware** (for example GPUs), you can taint them to avoid running non-specific applications on those nodes
  - An alpha (but soon to be beta) feature is to **taint nodes by condition**
    - This will automatically taint nodes that have node problems, allowing you to add tolerations to time the eviction of pods from nodes

Taints and tolerations

- You can enable alpha features by passing the --feature-gates to the Kubernetes controller manager, or in kops, you can use **kops edit** to add:

```
spec:
  kubelet:
    featureGates:
      TaintNodesByCondition: "true"
```

- In the next slide I'll show you a few taints that can be possibly added.

- This is an example of a toleration that could be used:

```
tolerations:
- key: "node.alpha.kubernetes.io/unreachable"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 6000
```

Taints and tolerations

- **node.kubernetes.io/not-ready**: Node is not ready
- **node.kubernetes.io/unreachable**: Node is unreachable from the node controller
- **node.kubernetes.io/out-of-disk**: Node becomes out of disk.
- **node.kubernetes.io/memory-pressure**: Node has memory pressure.
- **node.kubernetes.io/disk-pressure**: Node has disk pressure.
- **node.kubernetes.io/network-unavailable**: Node's network is unavailable.
- **node.kubernetes.io/unschedulable**: Node is unschedulable.

Taints and tolerations

Demo