Namespace

Namespaces

Kubernetes supports multiple virtual clusters backed by the same physical cluster. These **virtual clusters** are called namespaces.

- Namespaces **logically separates** your cluster
- The standard namespaces is called "**default**" & that's where all the resources launched by default
- There is also namespace for kubernetes specific resources, called **kube-system**
- Namespaces are intended for use in environments with many users spread across multiple **teams**, or **projects**

Namespaces

- Namespaces provide a scope for **names**.
- Names of resources need to be **unique** within a namespace, but not across namespaces.
- Namespaces cannot be nested inside one another and each Kubernetes resource can only be in one namespace.
- Namespaces are a way to divide cluster resources between multiple users via **resource quotas**

View Namespaces

```
> kubectl get namespaces
NAME          STATUS   AGE
default       Active   20m
kube-system   Active   20m
```

View Resources in a Namespace

```
> kubectl get pods
NAME                  READY     STATUS    RESTARTS   AGE
myapp-pod             1/1       Running   0          20s
```

```
> kubectl get pods --namespace=kube-system
NAME                                    READY     STATUS    RESTARTS   AGE
etcd-kubemaster                         1/1       Running   2          14d
kube-apiserver-kubemaster               1/1       Running   4          14d
kube-controller-manager-kubemaster      1/1       Running   4          14d
kube-dns-86f4d74b45-zgh8p               3/3       Running   10         14d
kube-flannel-ds-gmg5r                   1/1       Running   4          14d
kube-flannel-ds-kt74d                   1/1       Running   4          14d
kube-flannel-ds-qtlg8                   1/1       Running   4          14d
kube-proxy-4nkb6                        1/1       Running   3          14d
kube-proxy-b6wnm                        1/1       Running   3          14d
kube-proxy-rph7b                        1/1       Running   2          14d
kube-scheduler-kubemaster               1/1       Running   3          14d
metrics-server-6fbfb84cdd-jt5q9         1/1       Running   0          3d
```

Create Namespace

namespace-definition.yml

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: team1
```

```
kubectl create -f namespace-definition.yml
namespace “team1” created
```

Assign Objects to Namespace

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: helloworld-deployment
  namespace: myspace
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: helloworld
    spec:
      containers:
      - name: k8s-demo
        image: amitvashist7/k8s-demo
        ports:
        - name: nodejs-port
          containerPort: 3000
```

```
# kubectl run nginx --image=nginx --namespace=myspace
```

Demo

Resource Quotas

Resource Quotas

- When a Kubernetes cluster is used by multiple **people** or **teams**, **resource management** becomes more important
  - You want to be able to **manage the resources** you give to a person or a team
  - You don't want one person or team **taking up all the resources** (e.g. CPU/Memory) of the cluster
- You can divide your cluster in **namespaces** (explained in next lecture) and enable resource quotas on it
  - You can do this using the **ResourceQuota** and **ObjectQuota** objects

Resource Quotas

- Each container can specify **request capacity** and **capacity limits**
  - **Request capacity** is an **explicit request** for resources
    - The scheduler can use the **request capacity** to make decisions on where to put the pod on
    - You can see it as a **minimum amount of resources the pod needs**
  - **Resource limit** is a limit imposed to the container
    - The container will not be able to utilize more resources than specified

Resource Quotas

- If a capacity quota (e.g. mem / cpu) has been specified by the administrator, then each pod needs to specify capacity quota during creation
  - The administrator can specify default request values for pods that don't specify any values for capacity
  - The same is valid for limit quotas
- If a resource is requested more than the allowed capacity, the server API will give an error 403 FORBIDDEN - and kubectl will show an error

Resource Quotas

- The administrator can set the following resource limits within a namespace:

Resource	Description
requests.cpu	The sum of **CPU requests** of all pods cannot exceed this value
requests.mem	The sum of **MEM requests** of all pods cannot exceed this value
requests.storage	The sum of **storage requests** of all persistent volume claims cannot exceed this value
limits.cpu	The sum of **CPU limits** of all pods cannot exceed this value
limits.memory	The sum of **MEM limits** of all pods cannot exceed this value

Resource Quotas

- The administrator can set the following object limits:

Resource	Description
configmaps	total number of **configmaps** that can exist in a namespace
persistentvolumeclaims	total number of **persistent volume claims** that can exist in a namespace
pods	total number of **pods** that can exist in a namespace
replicationcontrollers	total number of **replicationcontrollers** that can exist in a namespace
resourcequotas	total number of **resource quotas** that can exist in a namespace
services	total number of **services** that can exist in a namespace
services.loadbalancer	total number of **load balancers** that can exist in a namespace
services.nodeports	total number of **nodeports** that can exist in a namespace
secrets	total number of secrets that can exist in a namespace

Demo

User Management

RBAC

Demo

Labels

Labels

Labels are key/values 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 org. structure

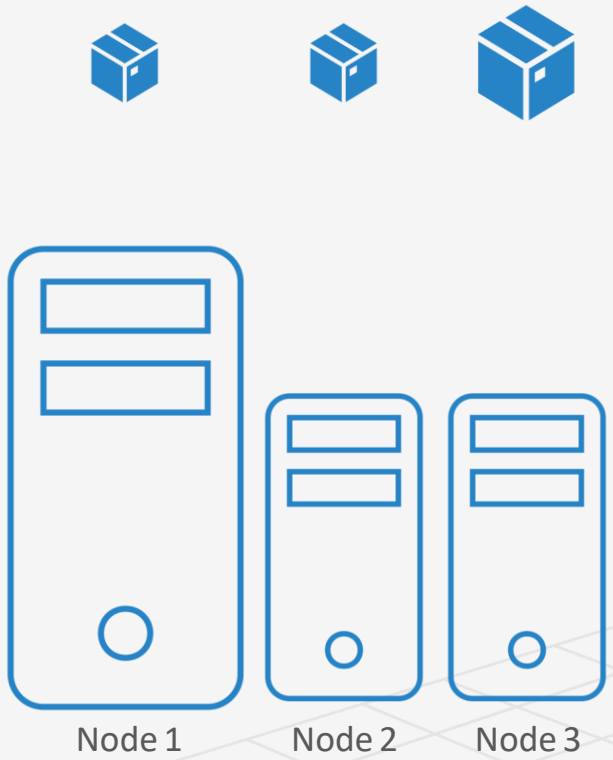
Key: environments - Value: Dev\ UAT\ QA\ PROD

Key : department - Value: R&D\ finance \ marketing

In Keyour previous examples, I already have been using labels to tag pods

Node Selectors

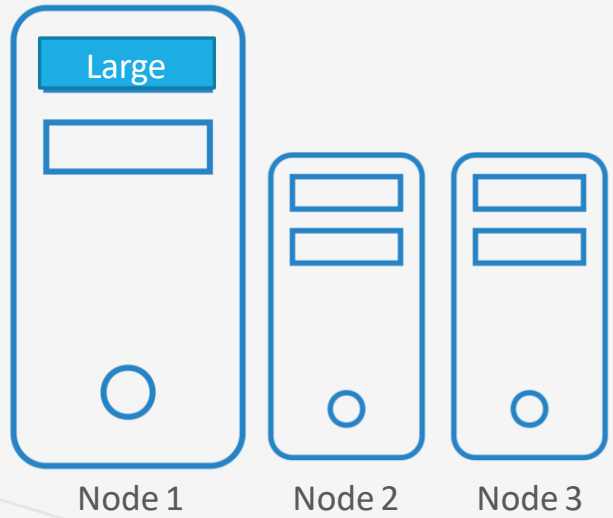
Node Selectors



Node Selectors

pod-definition.yml

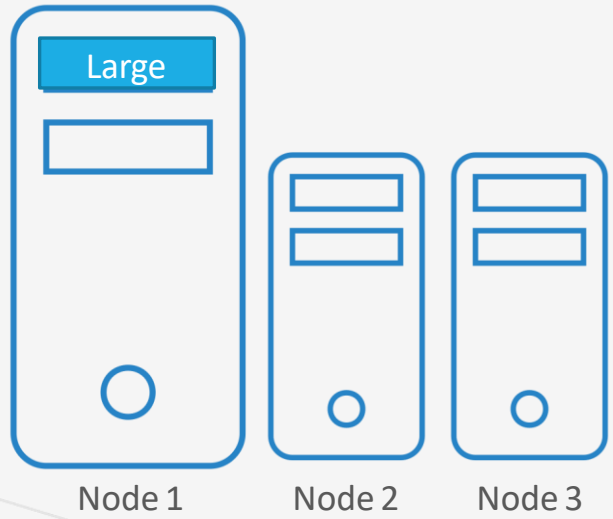
```
apiVersion:
kind: Pod
metadata:
 name: myapp-pod
spec:
  containers:
  - name: data-processor
    image: data-processor

  nodeSelector:
    size: Large
```



Label Nodes

```
kubectl label nodes <node-name> <label-key>=<label-value>
```

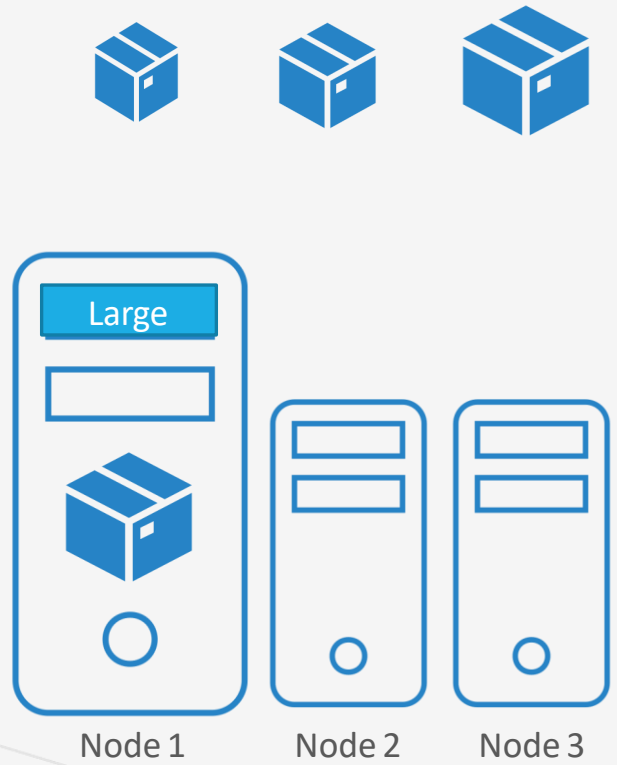
```
kubectl label nodes node-1 size=Large
```



Node Selectors

pod-definition.yml

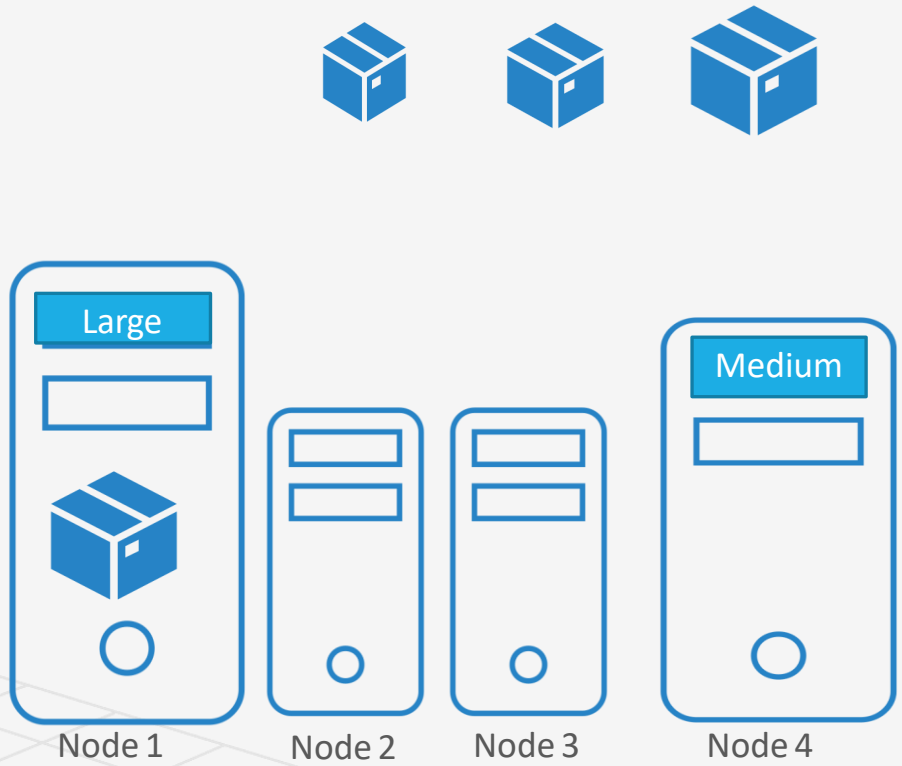
```
apiVersion:
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
  - name: data-processor
    image: data-processor

  nodeSelector:
    size: Large
```




```
kubectl create –f pod-definition.yml
```

Node Selectors - Limitations

- Large OR Medium?
- NOT Small



Taints And Tolerations

Taints And Tolerations

When you submit a workload to run in a cluster, the scheduler determines where to place the Pods associated with the workload.

The scheduler is **free** to place a Pod on any node that satisfies the **Pod's CPU, memory**, and **custom resource** requirements.

If your cluster runs a variety of workloads, you might want to exercise some **control** over which workloads can run on a particular pool of nodes.

A **node taint** lets you mark a node so that the scheduler avoids or prevents using it for certain Pods. A complementary feature, **tolerations**, lets you designate Pods that can be used on "**tainted**" nodes.

Taints And Tolerations

Node taints are key-value pairs associated with an effect. Here are the available effects:

- **NoSchedule:** Pods that do not tolerate this taint are not scheduled on the node; existing Pods are not evicted from the node.
- **PreferNoSchedule:** Kubernetes avoids scheduling Pods that do not tolerate this taint onto the node.
- **NoExecute:** Pod is evicted from the node if it is already running on the node and is not scheduled onto the node if it is not yet running on the node.
- **Note** that some system Pods (for example, kube-proxy and fluentd) tolerate all **NoExecute** and **NoSchedule** taints and will not be evicted.

Taints And Tolerations

Node taints are key-value pairs associated with an effect. Here are the available effects:

- **NoSchedule:** Pods that do not tolerate this taint are not scheduled on the node; existing Pods are not evicted from the node.
- **PreferNoSchedule:** Kubernetes avoids scheduling Pods that do not tolerate this taint onto the node.
- **NoExecute:** Pod is evicted from the node if it is already running on the node and is not scheduled onto the node if it is not yet running on the node.
- **Note** that some system Pods (for example, kube-proxy and fluentd) tolerate all **NoExecute** and **NoSchedule** taints and will not be evicted.

Taints And Tolerations

The node controller automatically taints a Node when certain conditions are true. The following taints are built in:

- **node.kubernetes.io/not-ready**: Node is not ready. This corresponds to the NodeCondition Ready being "False".
- **node.kubernetes.io/unreachable**: Node is unreachable from the node controller. This corresponds to the NodeCondition Ready being "Unknown".
- **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 - Node

```
kubectl taint nodes node-name key=value:taint-effect
```

What happens to PODs
that do not tolerate this taint?

NoSchedule | PreferNoSchedule | NoExecute

```
kubectl taint nodes node1 app=myapp:NoSchedule
```

Tolerations - PODs

```
kubectl taint  nodes node1 app=myapp:NoSchedule
```

```
kubectl describe node kubemaster |  grep  Taint
Taints:             node-role.kubernetes.io/master:NoSchedule
```

pod-definition.yml

```yaml
apiVersion:
kind: Pod
metadata:
 name: myapp-pod
spec:
  containers:
  - name: nginx-container
    image: nginx

  tolerations:

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

Demo

Node Affinity

Node Affinity

Node affinity is a set of rules used by the scheduler to determine where a pod can be placed. The rules are defined using custom labels on nodes and label selectors specified in pods. Node affinity allows a pod to specify an affinity (or anti-affinity) towards a group of nodes it can be placed on. The node does not have control over the placement.

For example, you could configure a pod to only run on a node with a specific **CPU** or in a specific **availability zone**.

There are two types of node affinity rules**: required** and **preferred**.

Required rules must be met before a pod can be scheduled on a node. Preferred rules specify that, if the rule is met, the scheduler tries to enforce the rules, but does not guarantee enforcement.

Node Affinity Types

Available:

requiredDuringScheduling**Ignored**DuringExecution

preferredDuringScheduling**Ignored**DuringExecution

	DuringScheduling	DuringExecution
Type 1	Required	Ignored
Type 2	Preferred	Ignored

Configuring Node Affinity

You configure node affinity through the pod specification file. You can specify a required rule, a preferred rule, or both. If you specify both, the node must first meet the required rule, then attempts to meet the preferred rule.

In this example is a node specification with a requires rule that a node with a label whose key is **e2e-az-EastWest** and whose value is either e2e-az-East **or** e2e-az-West is preferred for the pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: e2e-az-NorthSouth
            operator: In
            values:
            - e2e-az-North
            - e2e-az-South
  containers:
  - name: with-node-affinity
    image: docker.io/ocpqe/hello-pod
```

Configuring Node Affinity

In this example is a node specification with a preferred rule that a node with a label whose key is **e2e-az-EastWest** and whose value is either e2e-az-East **or** e2e-az-West is preferred for the pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: e2e-az-NorthSouth
            operator: In
            values:
            - e2e-az-North
            - e2e-az-South
  containers:
  - name: with-node-affinity
    image: docker.io/ocpqe/hello-pod
```

Configuring a Required Node Affinity Rule

```
kubectl label node node1 e2e-az-name=e2e-az1
```

- Specify the key and values that must be met. If you want the new pod to be scheduled on the node you edited, use the same **key** and **value** parameters as the label in the node.

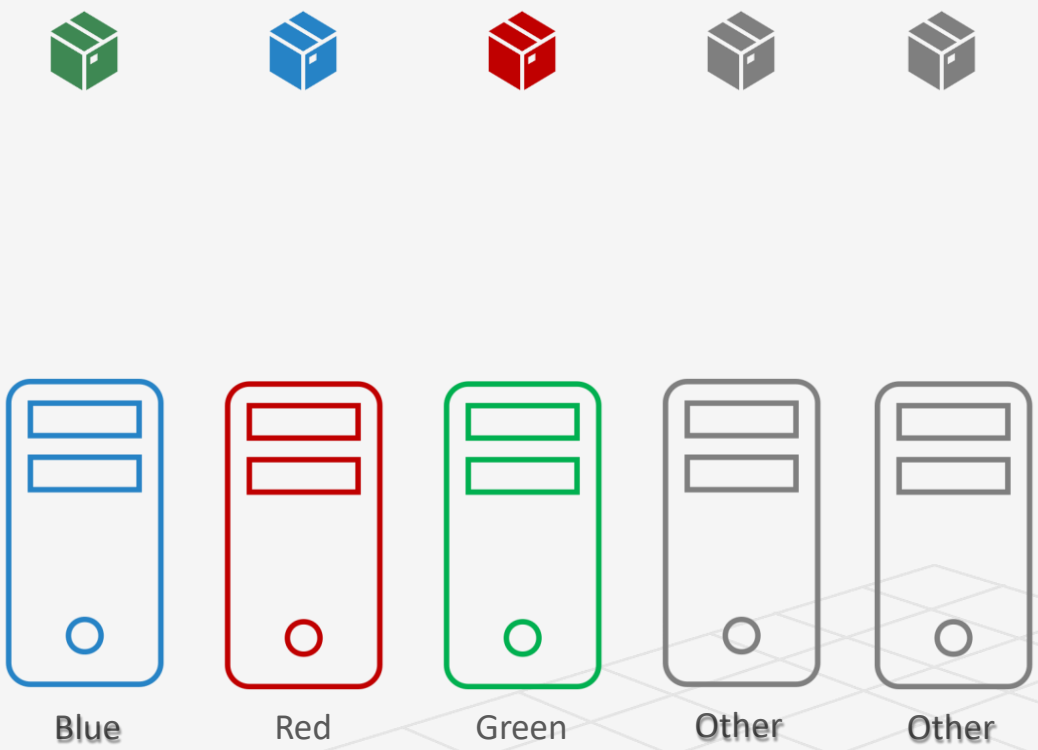
- Specify an operator. The operator can be **In, NotIn, Exists, DoesNotExist, Lt, or Gt**. For example, use the operator In to require the label to be in the node:

```
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: e2e-az-name
            operator: In
            values:
            - e2e-az1
            - e2e-az2
```

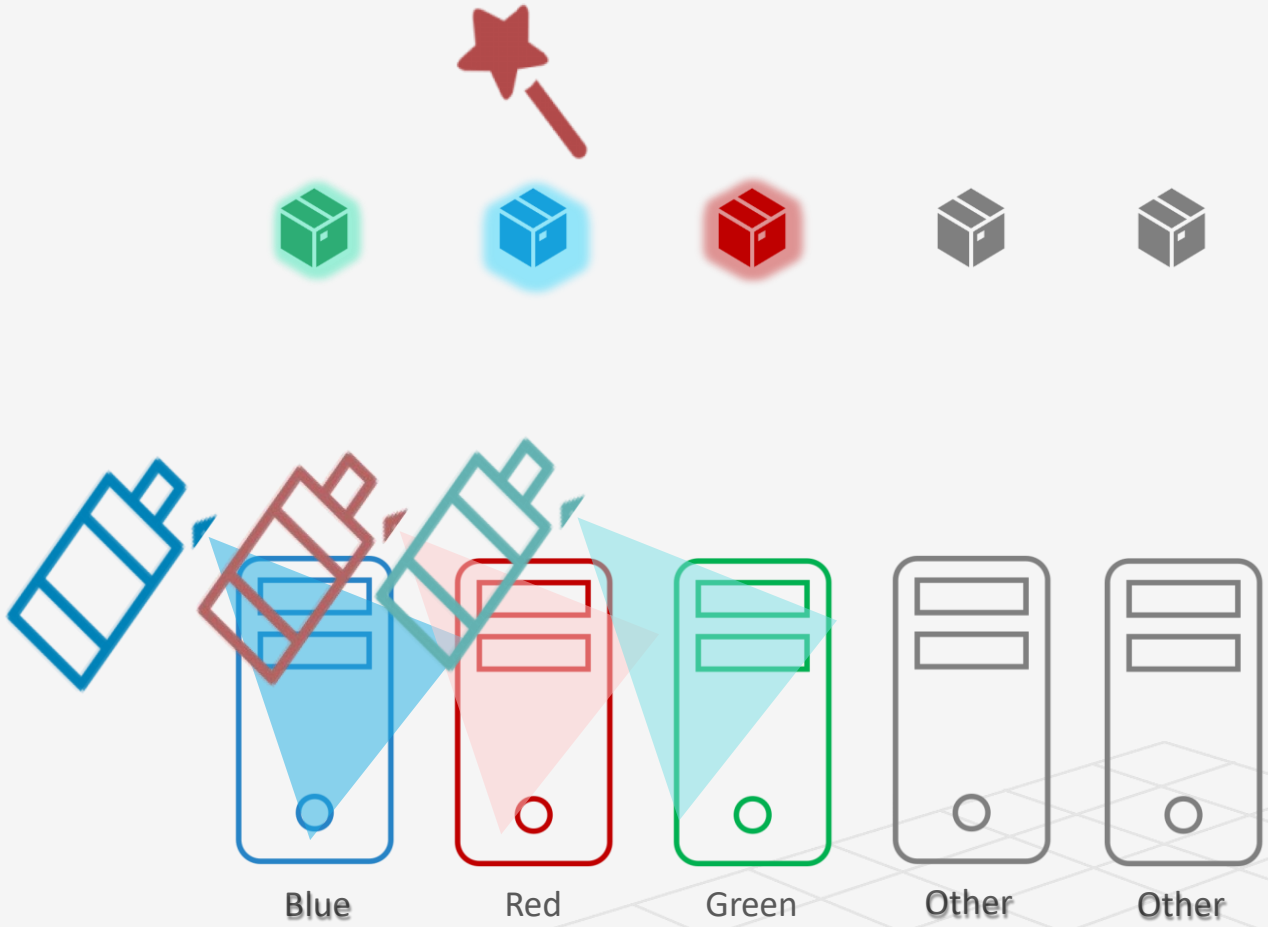
Demo

Node Affinity vs Taints and Tolerations

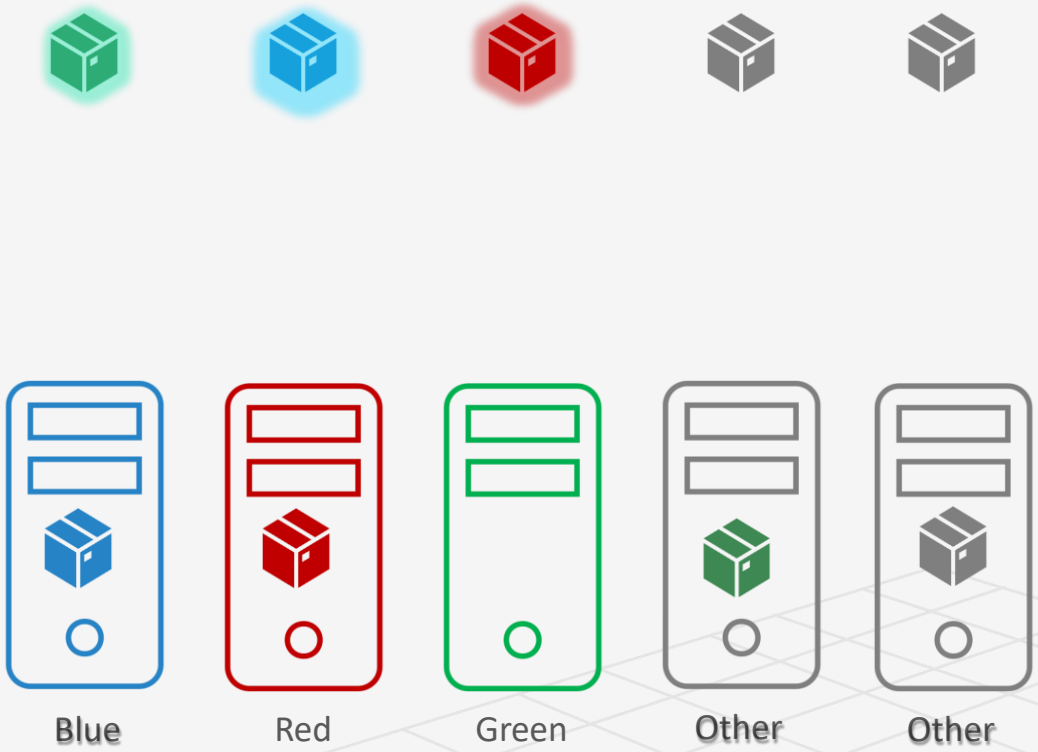
Example



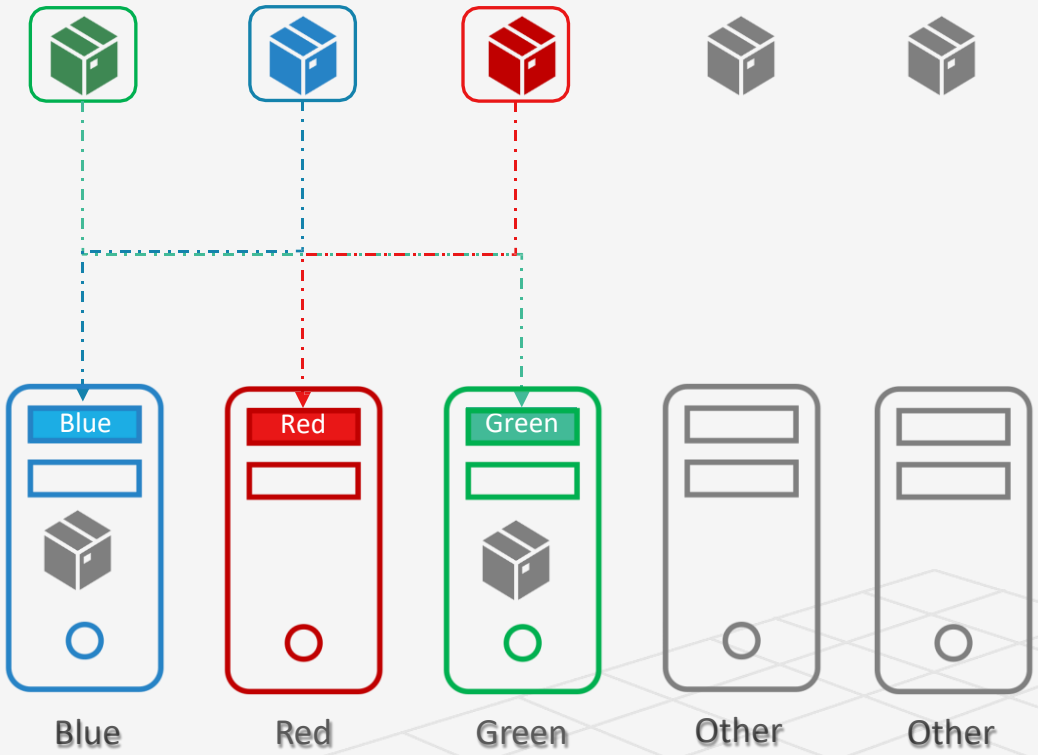
Taints & Tolerations



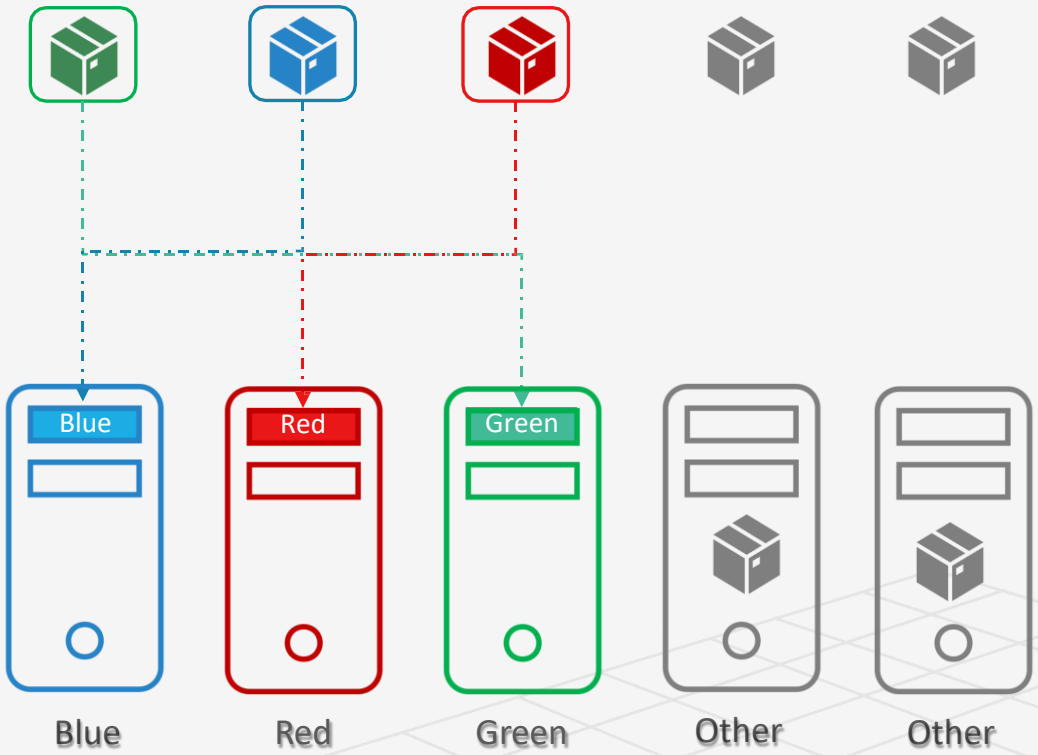
Taints & Tolerations



Node Affinity



Taints/Tolerations & Node Affinity



Demo

Image Security

Image

pod-nginx.yml

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

Image

```
image: docker.io/nginx/nginx
```

Registry User/Account Image/Repository

```
gcr.io/ kubernetes-e2e-test-images/dnsutils

docker.io/ amitvashist7 /k8s-tiny-web
```

Private Repository

```
docker login private-registry.io
```

```
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to
https://hub.docker.com to create one.
Username: registry-user
Password:
WARNING! Your password will be stored unencrypted in /home/vagrant/.docker/config.json.

Login Succeeded
```

```
docker run private-registry.io/apps/internal-app
```

Private Repository

```
docker login private-registry.io
```

```
docker run private-registry.io/apps/internal-app
```

```
kubectl create secret docker-registry regcred \
  --docker-server= private-registry.io     \
  --docker-username= registry-user   \
  --docker-password= registry-password   \
  --docker-email= registry-user@org.com
```

pod-nginx.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image:
  imagePullSecrets:
  - name: regcred
```

Demo

Secrets

Secrets

- Secrets provides a way in kubernetes to distribute **credentials, Keys, passwords** or **"secret"** data to the pods
- Kubernetes itself uses this Secrets mechanism to provide the credentials to access the internal API
- You can also use the **same mechanism** to provide the secret to your application
- Secrets is one way to provide secret, native to Kubernetes
  - There are still other ways your container can get its secrets if you don't want to use Secrets (e.g using **external vault** services in your app)

Secrets

- Secrets can be used in the following ways:
  - Use Secrets as **environment varia**bles
  - Use secrets **as a file** in a pod
    - This setup uses **volumes** to be mounted in a container
    - In this volume you have **files**
    - Can be used for instance for **dotenv** files or you app can just read this file

Secrets

- To generate secrets using files:

```
# echo -n "root" > ./username.txt
# echo -n "password" > ./password.txt

# kubctl create secret generic db-user-pass --from-file=./username.txt --from-file=./password.txt
```

- A Secret can also be an SSH Key or an SSL certificate

```
# kubctl create secret generic ssl-certificate --from-file=ssh-privatekey=~/.ssh/id_rsa  --ssl-cert=ssl-cert=mysslcert.crt
```

Secrets

- To generate secret using yaml definitions:

  secrets-db-secret.yml

```
-----------------------------------
apiVersion: v1
kind: Secret
metadata:
  name: db-secrets
type: Opaque
data:
  username: cm9vdA==
  password: cGFzc3dvcmQ=
-----------------------------------
```

```
# echo -n 'root' | base64
cm9vdA==
# echo -n 'password' | base64
cGFzc3dvcmQ=
```

- After creating the yml file, you can use kubectl create:

```
# kubectl create -f secrets-db-secret.yml
```

Secrets

- You can create a pod that expose the secret as environment variables:

```
env:
        - name: MYSQL_HOST
          value: database-service
        - name: MYSQL_USER
          value: root
        - name: MYSQL_PASSWORD
          valueFrom:
            secretKeyRef:
              name: helloworld-secrets
              key: rootPassword
        - name: MYSQL_DATABASE
          valueFrom:
            secretKeyRef:
              name: helloworld-secrets
              key: database
```

Demo

Secrets

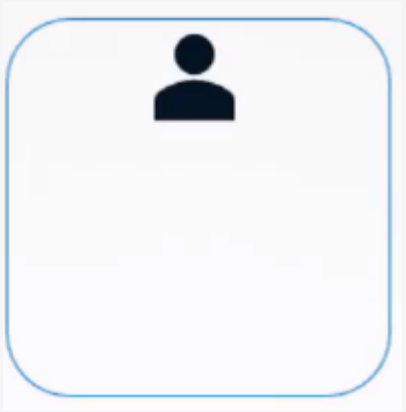
Setting up WordPress

Security Contexts

Container Security

```
docker run --user=1001 ubuntu sleep 3600
```

```
docker run --cap-add MAC_ADMIN ubuntu
```



Security Context

```
docker run --user=1001 ubuntu sleep 3600
```

```
docker run --cap-add NET_ADMIN ubuntu
```

pod-sec-coxt.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context
spec:
  containers:
  - name: sec-demo
    image: ubuntu
    command: ["sleep", "3600"]
    securityContext:
      runAsUser: 1000
      capabilities:
        add: ["NET_ADMIN", "SYS_TIME"]
```

Security Context

A security context defines privilege and access control settings for a Pod or Container. Security context settings include, but are not limited to:

- Discretionary Access Control: Permission to access an object, like a file, is based on **user ID (UID)** and **group ID (GID).**
- **Security Enhanced Linux (SELinux):** Objects are assigned security labels.
- Running as privileged or unprivileged.
- **Linux Capabilities**: Give a process some privileges, but not all the privileges of the root user.
- **AppArmor**: Use program profiles to restrict the capabilities of individual programs.

Demo

Network Policies

Network Policies

If you want to control traffic flow at the IP address or port level (OSI layer 3 or 4), then you might consider using Kubernetes NetworkPolicies for particular applications in your cluster.

NetworkPolicies are an application-centric construct which allow you to specify how a pod is allowed to communicate with various network "entities"over the network.

The entities that a Pod can communicate with are identified through a combination of the following 3 identifiers:

- Other pods that are allowed (exception: a pod cannot block access to itself)
- Namespaces that are allowed
- IP blocks (exception: traffic to and from the node where a Pod is running is always allowed, regardless of the IP address of the Pod or the node)

Network Policies

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

Behavior of to and from selectors

There are four kinds of selectors that can be specified in an **ingress from** section or **egress to** section:

podSelector: This selects particular Pods in the same namespace as the NetworkPolicy which should be allowed as ingress sources or egress destinations.

namespaceSelector: This selects particular namespaces for which all Pods should be allowed as ingress sources or egress destinations.

namespaceSelector and **podSelector**: A single **to/from** entry that specifies both **namespaceSelector** and **podSelector** selects particular Pods within particular namespaces.

```
...
ingress:
- from:
  - namespaceSelector:
      matchLabels:
        user: alice
    podSelector:
      matchLabels:
        role: client
...
```

Demo