Namespaces, labels and annotations

What are they and why do we need them?

- **Namespaces** allow you to subdivide a single physical cluster into isolated sections, each of which is called a namespace.
- **Labels:** are key/value pairs that help for grouping objects, and attaching identifying information.
- **Annotations:** are also key/value pairs that hold non-identifying information which can be useful for other tools and libraries.

Namespaces

- The namespace provides the scope for names. Names of resources within one namespace need to be unique.
- Another reason is to better monitor resources by team.
- **Example:** development, QA, prod namespaces.

Namespaces (cont.)

By default, Kubernetes starts with the following three namespaces:

- **Default:** holds the default set of pods, services, and deployments used by the cluster.
- **Kube-public:** Namespace for resources that are publicly available/readable by all.
- **Kube-system:** Namespace for objects/resources created by Kubernetes systems.

```
kubectl get namespaces
```

```
kubectl get namespaces --show-labels
```

How do we create new namespaces?

- Get the current context

  ```
  kubectl config view --minify
  ```

How do we create new namespaces? (cont.)

- Set up new namespaces

```
kubectl config set-context dev --namespace=development \
  --cluster=YOUR_CLUSTER \
  --user=YOUR_USER

kubectl config set-context prod --namespace=production \
  --cluster=YOUR_CLUSTER\
  --user=YOUR_USER
```

How do we create new namespaces? (cont.)

- Switch context

  ```
  kubectl config use-context dev
  ```

- Check that we are using the right context

  ```
  kubectl config current-context
  ```

- To delete:
  - Exit to another context.
  - `kubectl config delete-context dev`

Common use cases for namespaces

- Roles and Responsibilities.
  - **Architect:** sets up namespace strategy, avoiding snowflake namespaces.
  - **Admin:** implement the namespaces strategy defined by the architect.
- Antipatterns:
  - Disordered mushrooming of clusters.
  - One large cluster that cannot be broken down.

Common use cases for namespaces (cont.)

- Partitioning landscapes: dev vs. test vs. prod
  - Consistent naming across environments.
  - Better resource control.
- Antipatterns:
  - Over-naming: if no staging is done, then no need of a staging namespace.
  - All projects land on a `dev` namespace. There are no hierarchies in namespaces, so it is better to have `projectABC-dev`, `projectABC-prod`, etc.

Common use cases for namespaces (cont.)

- Customer partitioning for non-multi-tenant scenarios.
  - Separate namespaces for clients/projects.
- Warning:
  - No official mechanism to enforce access controls across namespaces, even if you can hack your way through using service accounts.

When to avoid namespaces

- **No ability to enforce partitioning across namespaces**. Users may be able to access other resources in the cluster, regardless of their namespace.
- For billing/geographical/compliance constraints, the best way to enforce separation is to set up different clusters (and enforce suitable controls there).
- Namespaces are not hierarchies, and should not be used for versioning.

Resource quotas

- A resource quota limits the amount of resources that namespace can use.
- Resource quotas can limit anything in the namespace"
  - total count of each type of object
  - the total storage used
  - total memory or CPU usage of containers in the namespace.
- Good practice to set up **resources.requests** and **resources.limits** in the YAML file.

Demo

Resource quotas

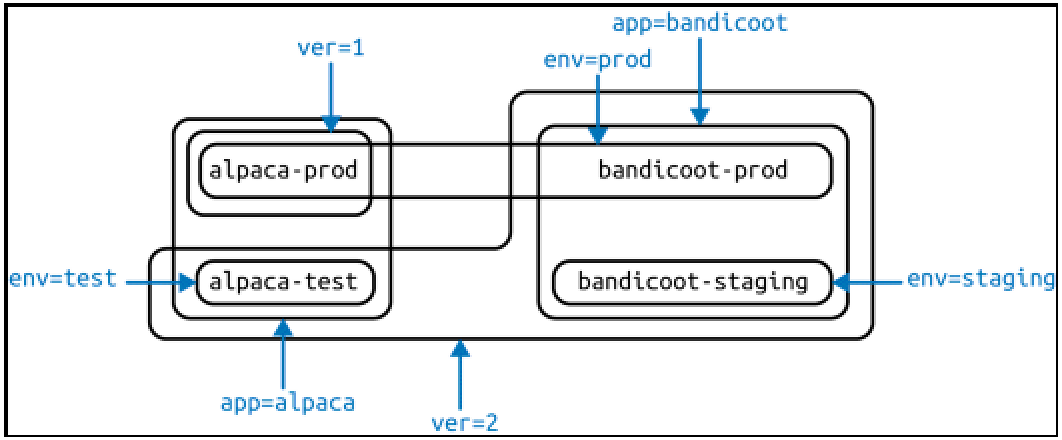
Labels

Why labels?

- **Production environment hates singletons:** you may start with a single instance, but then you need to multiply.
- **Hierarchies are not eternal:** Strict hierarchies may complicate scalability. Labels are flexible enough to adapt to these situations.

Labels syntax: Example

- Let's create two apps, with two environments each.



- Hint:

```
kubectl run alpaca-prod --image=nginx --labels="ver=1,app=alpaca,env=prod" --generator=run-pod/v1
```

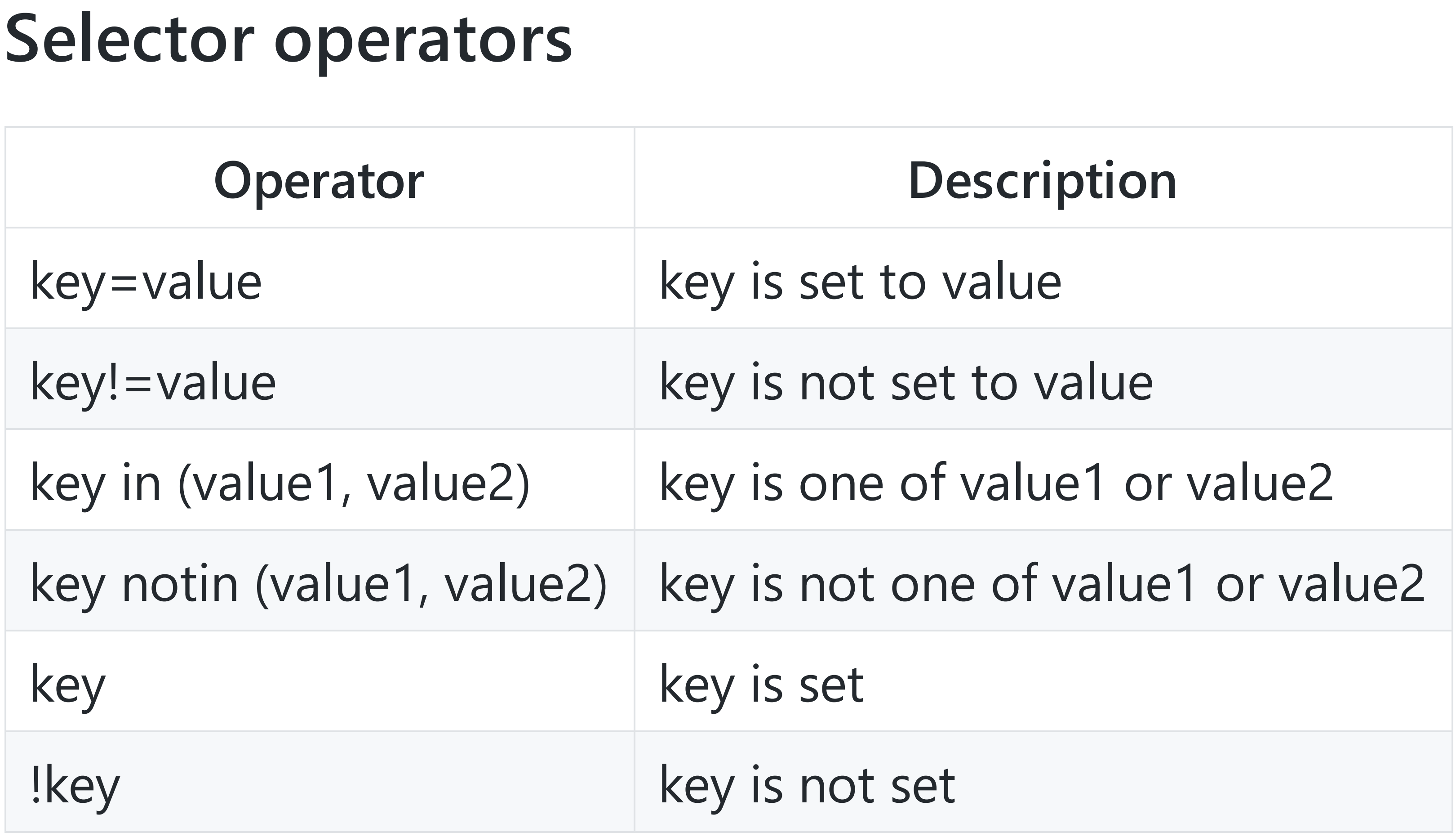
- Admire our masterpiece:
  - `kubectl get pods --show-labels`

Modifying or updating our labels

- `kubectl label pods alpaca-test "canary=true"`
- `kubectl get pods -L canary`
- `kubectl label pods alpaca-test "canary-"`

Label selectors

- `kubectl get pods --selector="ver=2"`
- `kubectl get pods --selector="app=bandicoot,ver=2"`
- `kubectl get pods --selector="app in (alpaca,bandicoot)"`



Selector operators

Operator	Description
key=value	key is set to value
key!=value	key is not set to value
key in (value1, value2)	key is one of value1 or value2
key notin (value1, value2)	key is not one of value1 or value2
key	key is set
!key	key is not set

Annotations

- Add extra metadata:
  - Build, release or image information (Git hashes, timestamp, etc).
  - Keep track of why an object update happened.
  - Way to communicate with other tools (e.g. Ingress).
- Free-form string field.

```
metadata:
        annotations:
                icon-url:"https://example.com/icon.png"
```

ConfigMaps and Secrets

ConfigMaps

- Small filesystem.
- Set of environment variables/command line in containers.
- Combined with the pod right before, which makes the pod definition itself fully reusable in other environments, by just changing the ConfigMap.
- Can be created in an imperative way or in a declarative way by means of a manifest file.

Example (Imperative)

```
# my-config.txt sample config file
parameter1 = value1
parameter2 = value2
```

```
kubectl create configmap my-config \
--from-file = my-config.txt \
--from-literal = extra-param = extra-value \
--from-literal = another-param = another-value
```

```
# Equivalent YAML file
kubectl get configmaps my-config -o yaml
```

Using a ConfigMap

- **Filesystem:** You can mount a ConfigMap into a pod. A file is created for each entry based on the key name. The contents of the file are set to the value.
- **Environment variable/command line:** Dynamically create the command line for a container.

Example (Filesystem)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: kuard-config
spec:
  containers:
   - name: test-container
     image: gcr.io/kuar-demo/kuard-amd64:blue
     volumeMounts:
     - name: config-volume
       mountPath: /config
  volumes:
  - name: config-volume
    configMap:
        name: my-config
```

Example (Environment)

```
apiVersion: v1
kind: Pod
metadata:
  name: kuard-config
spec:
  containers:
   - name: test-container
     image: gcr.io/kuar-demo/kuard-amd64:blue
     command:
         - "/kuard"
        - "$(EXTRA_PARAM)"
     env:
        - name: ANOTHER_PARAM
          valueFrom:
             configMapKeyRef:
               name: my-config
               key: another-param
        - name: EXTRA_PARAM
           valueFrom:
             configMapKeyRef:
               name: my-config
               key: extra-param
```

Secrets

- Handle extra-sensitive data (passwords, security tokens, private keys).
- By default, K8s secrets are stored as plain text in the `etcd` storage of the cluster. So *anyone with cluster admin rights* can read everything.
- Recent versions have support for encrypting secrets with user-supplied keys (usually integrated into a cloud key store). This allows to skip Kubernetes secrets entirely and rely on the cloud provider's key store.
- **Slides from Google**.

Demo

Managing configuration maps and secrets.