Chapter 6. Labels and Annotations

Kubernetes was made to grow with you as your application scales both in size and complexity. With this in mind, labels and annotations were added as foundational concepts. Labels and annotations let you work in sets of things that map to how *you* think about your application. You can organize, mark, and cross-index all of your resources to represent the groups that make the most sense for your application.

Labels are key/value pairs that can be attached to Kubernetes objects such as Pods and ReplicaSets. They can be arbitrary, and are useful for attaching identifying information to Kubernetes objects. Labels provide the foundation for grouping objects.

Annotations, on the other hand, provide a storage mechanism that resembles labels: annotations are key/value pairs designed to hold nonidentifying information that can be leveraged by tools and libraries.

Labels

Labels provide identifying metadata for objects. These are fundamental qualities of the object that will be used for grouping, viewing, and operating.

> **NOTE**
>
> The motivations for labels grew out of Google's experience in running large and complex applications. There were a couple of lessons that emerged from this experience. See the great *Site Reliability Engineering* by Betsy Beyer et al. (O'Reilly) for some deeper background on how Google approaches production systems.
>
> The first lesson is that production abhors a singleton. When deploying software, users will often start with a single instance. However, as the application matures, these singletons often multiply and become sets of objects. With this in mind, Kubernetes uses labels to deal with *sets* of objects instead of single instances.
>
> The second lesson is that any hierarchy imposed by the system will fall short for many users. In addition, user grouping and hierarchy are subject to change over time. For instance, a user may start out with the idea that all apps are made up of many services. However, over time, a service may be shared across multiple apps. Kubernetes labels are flexible enough to adapt to these situations and more.

Labels have simple syntax. They are key/value pairs where both the key and value are represented by strings. Label keys can be broken down into two parts: an optional prefix and a name, separated by a slash. The prefix, if specified, must be a DNS subdomain with a 253-character limit. The key name is required and must be shorter than 63 characters. Names must also start and end with an alphanumeric character and permit the use of dashes (-), underscores (_), and dots (.) between characters.

Label values are strings with a maximum length of 63 characters. The contents of the label values follow the same rules as for label keys.

Table 6-1 shows valid label keys and values.

Table 6-1. Label examples

Key	Value
`acme.com/app-version`	`1.0.0`

appVersion	1.0.0
app.version	1.0.0
kubernetes.io/cluster-service	true

Applying Labels

Here we create a few deployments (a way to create an array of Pods) with some interesting labels. We'll take two apps (called `alpaca` and `bandicoot`) and have two environments for each. We will also have two different versions.

1. First, create the `alpaca-prod` deployment and set the `ver`, `app`, and `env` labels:

```
$ kubectl run alpaca-prod \
  --image=gcr.io/kuar-demo/kuard-amd64:1 \
  --replicas=2 \
  --labels="ver=1,app=alpaca,env=prod"
```

2. Next, create the `alpaca-test` deployment and set the `ver`, `app`, and `env` labels with the appropriate values:

```
$ kubectl run alpaca-test \
  --image=gcr.io/kuar-demo/kuard-amd64:2 \
  --replicas=1 \
  --labels="ver=2,app=alpaca,env=test"
```

3. Finally, create two deployments for `bandicoot`. Here we name the environments `prod` and `staging`:

```
$ kubectl run bandicoot-prod \
  --image=gcr.io/kuar-demo/kuard-amd64:2 \
  --replicas=2 \
  --labels="ver=2,app=bandicoot,env=prod"
$ kubectl run bandicoot-staging \
  --image=gcr.io/kuar-demo/kuard-amd64:2 \
  --replicas=1 \
  --labels="ver=2,app=bandicoot,env=staging"
```

At this point you should have four deployments — `alpaca-prod`, `alpaca-staging`, `bandicoot-prod`, and `bandicoot-staging`:

```
$ kubectl get deployments --show-labels

NAME                ... LABELS
alpaca-prod         ... app=alpaca,env=prod,ver=1
alpaca-test         ... app=alpaca,env=test,ver=2
bandicoot-prod      ... app=bandicoot,env=prod,ver=2
bandicoot-staging   ... app=bandicoot,env=staging,ver=2
```

We can visualize this as a Venn diagram based on the labels (Figure 6-1).

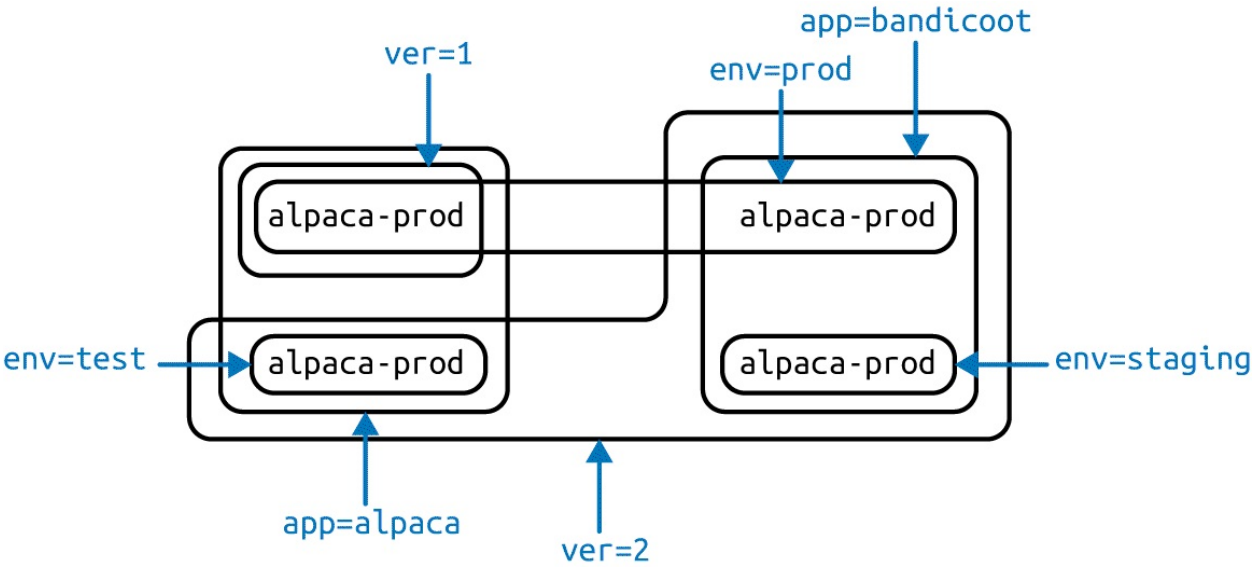

Figure 6-1. Visualization of labels applied to our deployments

Modifying Labels

Labels can also be applied (or updated) on objects after they are created.

```
$ kubectl label deployments alpaca-test "canary=true"
```

> **WARNING**
>
> There is a caveat to be aware of here. In this example, the `kubectl label` command will only change the label on the deployment itself; it won't affect the objects (ReplicaSets and Pods) the deployment creates. To change those, you'll need to change the template embedded in the deployment (see Chapter 12).

You can also use the `-L` option to `kubectl get` to show a label value as a column:

```
$ kubectl get deployments -L canary

NAME                DESIRED   CURRENT   ... CANARY
alpaca-prod         2         2         ... <none>
alpaca-test         1         1         ... true
bandicoot-prod      2         2         ... <none>
bandicoot-staging   1         1         ... <none>
```

You can remove a label by applying a dash suffix:

```
$ kubectl label deployments alpaca-test "canary-"
```

Label Selectors

Label selectors are used to filter Kubernetes objects based on a set of labels. Selectors use a simple Boolean language. They are used both by end users (via tools like `kubectl`) and by different types of objects (such as how ReplicaSet relates to its Pods).

Each deployment (via a ReplicaSet) creates a set of Pods using the labels specified in the template embedded in the deployment. This is configured by the `kubectl run` command.

Running the `kubectl get pods` command should return all the Pods currently running in the cluster. We should have a total of six `kuard` Pods across our three environments:

```
$ kubectl get pods --show-labels

NAME                                ... LABELS
alpaca-prod-3408831585-4nzfb        ... app=alpaca,env=prod,ver=1,...
alpaca-prod-3408831585-kga0a        ... app=alpaca,env=prod,ver=1,...
alpaca-test-1004512375-3r1m5        ... app=alpaca,env=test,ver=2,...
bandicoot-prod-373860099-0t1gp      ... app=bandicoot,env=prod,ver=2,...
bandicoot-prod-373860099-k2wcf      ... app=bandicoot,env=prod,ver=2,...
bandicoot-staging-1839769971-3ndv ... app=bandicoot,env=staging,ver=2,...
```

> **NOTE**
>
> You may see a new label that we haven't seen yet: `pod-template-hash`. This label is applied by the deployment so it can keep track of which pods were generated from which template versions. This allows the deployment to manage updates in a clean way, as will be covered in depth in Chapter 12.

If we only wanted to list pods that had the `ver` label set to `2` we could use the `--selector` flag:

```
$ kubectl get pods --selector="ver=2"

NAME                                READY     STATUS    RESTARTS   AGE
alpaca-test-1004512375-3r1m5        1/1       Running   0          3m
bandicoot-prod-373860099-0t1gp      1/1       Running   0          3m
bandicoot-prod-373860099-k2wcf      1/1       Running   0          3m
bandicoot-staging-1839769971-3ndv5  1/1       Running   0          3m
```

If we specify two selectors separated by a comma, only the objects that satisfy both will be returned. This is a logical AND operation:

```
$ kubectl get pods --selector="app=bandicoot,ver=2"

NAME                                READY     STATUS    RESTARTS   AGE
bandicoot-prod-373860099-0t1gp      1/1       Running   0          4m
bandicoot-prod-373860099-k2wcf      1/1       Running   0          4m
bandicoot-staging-1839769971-3ndv5  1/1       Running   0          4m
```

We can also ask if a label is one of a set of values. Here we ask for all pods where the `app` label is set to `alpaca` or `bandicoot` (which will be all six pods):

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

NAME                                READY     STATUS    RESTARTS   AGE
alpaca-prod-3408831585-4nzfb        1/1       Running   0          6m
alpaca-prod-3408831585-kga0a        1/1       Running   0          6m
alpaca-test-1004512375-3r1m5        1/1       Running   0          6m
bandicoot-prod-373860099-0t1gp      1/1       Running   0          6m
bandicoot-prod-373860099-k2wcf      1/1       Running   0          6m
bandicoot-staging-1839769971-3ndv5  1/1       Running   0          6m
```

Finally, we can ask if a label is set at all. Here we are asking for all of the deployments with the `canary` label set to anything:

```
$ kubectl get deployments --selector="canary"

NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
alpaca-test   1         1         1            1           7m
```

There are also “negative” versions of each of these, as shown in Table 6-2.

Table 6-2. 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

Label Selectors in API Objects

When a Kubernetes object refers to a set of other Kubernetes objects, a label selector is used. Instead of a simple string as described in the previous section, a parsed structure is used.

For historical reasons (Kubernetes doesn't break API compatibility!), there are two forms. Most objects support a newer, more powerful set of selector operators.

A selector of `app=alpaca,ver in (1, 2)` would be converted to this:

```
selector:
  matchLabels:
    app: alpaca
  matchExpressions:
    - {key: ver, operator: In, values: [1, 2]} ❶
```

❶ Compact YAML syntax. This is an item in a list (`matchExpressions`) that is a map with three entries. The last entry (`values`) has a value that is a list with two items.

All of the terms are evaluated as a logical AND. The only way to represent the `!=` operator is to convert it to a `NotIn` expression with a single value.

The older form of specifying selectors (used in `ReplicationControllers` and services) only supports the `=` operator. This is a simple set of key/value pairs that must all match a target object to be selected.

The selector `app=alpaca,ver=1` would be represented like this:

```
selector:
  app: alpaca
  ver: 1
```

Annotations

Annotations provide a place to store additional metadata for Kubernetes objects with the sole purpose of assisting tools and libraries. They are a way for other programs driving Kubernetes via an API to store some opaque data with an object. Annotations can be used for the tool itself or to pass configuration information between external systems.

While labels are used to identify and group objects, annotations are used to provide extra information about where an object came from, how to use it, or policy around that object. There is overlap, and it is a matter of taste as to when to use an annotation or a label. When in doubt, add information to an object as an annotation and promote it to a label if you find yourself wanting to use it in a selector.

Annotations are used to:

- Keep track of a "reason" for the latest update to an object.
- Communicate a specialized scheduling policy to a specialized scheduler.
- Extend data about the last tool to update the resource and how it was updated (used for detecting changes by other tools and doing a smart merge).
- Build, release, or image information that isn't appropriate for labels (may include a Git hash, timestamp, PR number, etc.).
- Enable the `Deployment` object (Chapter 12) to keep track of ReplicaSets that it is managing for rollouts.
- Provide extra data to enhance the visual quality or usability of a UI. For example, objects could include a link to an icon (or a base64-encoded version of an icon).
- Prototype alpha functionality in Kubernetes (instead of creating a first-class API field, the parameters for that functionality are instead encoded in an annotation).

Annotations are used in various places in Kubernetes, with the primary use case being rolling deployments. During rolling deployments, annotations are used to track rollout status and provide the necessary information required to roll back a deployment to a previous state.

Users should avoid using the Kubernetes API server as a general-purpose database. Annotations are good for small bits of data that are highly associated with a specific resource. If you want to store data in Kubernetes but you don’t have an obvious object to associate it with, consider storing that data in some other, more appropriate database.

Defining Annotations

Annotation keys use the same format as label keys. However, because they are often used to communicate information between tools, the “namespace” part of the key is more important. Example keys include `deployment.kubernetes.io/revision` or `kubernetes.io/change-cause`.

The value component of an annotation is a free-form string field. While this allows maximum flexibility as users can store arbitrary data, because this is arbitrary text, there is no validation of any format. For example, it is not uncommon for a JSON document to be encoded as a string and stored in an annotation. It is important to note that the Kubernetes server has no knowledge of the required format of annotation values. If annotations are used to pass or store data, there is no guarantee the data is valid. This can make tracking down errors more difficult.

Annotations are defined in the common `metadata` section in every Kubernetes object:

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

Annotations are very convenient and provide powerful loose coupling. However, they should be used judiciously to avoid an untyped mess of data.

Cleanup

It is easy to clean up all of the deployments that we started in this chapter:

```
$ kubectl delete deployments --all
```

If you want to be more selective you can use the `--selector` flag to choose which deployments to delete.

Summary

Labels are used to identify and optionally group objects in a Kubernetes cluster. Labels are also used in selector queries to provide flexible runtime grouping of objects such as pods.

Annotations provide object-scoped key/value storage of metadata that can be used by automation tooling and client libraries. Annotations can also be used to hold configuration data for external tools such as third-party schedulers and monitoring tools.

Labels and annotations are key to understanding how key components in a Kubernetes cluster work together to ensure the desired cluster state. Using labels and annotations properly unlocks the true power of Kubernetes's flexibility and provides the starting point for building automation tools and deployment workflows.