O'REILLY®

3rd Edition

Kubernetes Up & Running

Dive into the Future of Infrastructure

Early Release
Raw & Unedited
Compliments of
VMware Tanzu™

Brendan Burns, Joe Beda,
Kelsey Hightower & Lachlan Evenson

THIRD EDITION

Kubernetes: Up and Running

Dive into the Future of Infrastructure

Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

VMware Tanzu™

vmware®

Free your apps. Simplify your ops.

Ease Kubernetes adoption and run modern apps at scale. Learn to bring Dev and Ops together to continuously build, run, and manage apps your users love on any cloud you choose.

LEARN MORE

Kubernetes: Up and Running

by Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisition Editor: John Devins
Development Editor: Sarah Gray
Production Editor: Katherine Tozer
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

September 2017:	First Edition
August 2019:	Second Edition
September 2022:	Third Edition

Revision History for the Early Release

2021-10-20:	First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492046530* for release details.

978-1-098-11013-0

Table of Contents

CHAPTER 1

Pods

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sgrey@oreilly.com*.

In earlier chapters we discussed how you might go about containerizing your application, but in real-world deployments of containerized applications you will often want to colocate multiple applications into a single atomic unit, scheduled onto a single machine.

A canonical example of such a deployment is illustrated in Figure 1-1, which consists of a container serving web requests and a container synchronizing the filesystem with a remote Git repository.

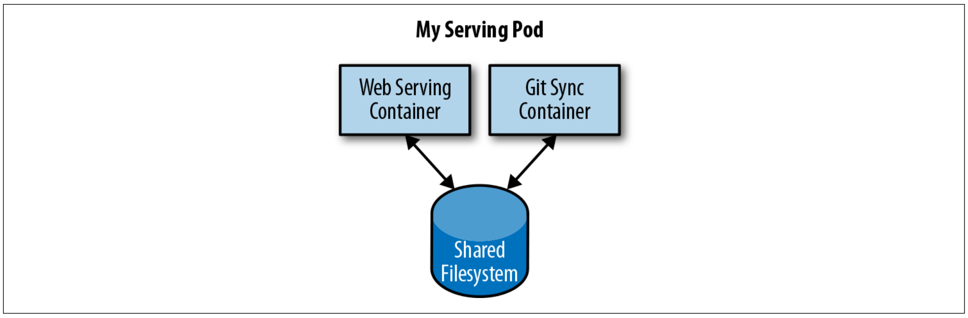

Figure 1-1. An example Pod with two containers and a shared filesystem

At first, it might seem tempting to wrap up both the web server and the Git synchronizer into a single container. After closer inspection, however, the reasons for the separation become clear. First, the two different containers have significantly different requirements in terms of resource usage. Take, for example, memory. Because the web server is serving user requests, we want to ensure that it is always available and responsive. On the other hand, the Git synchronizer isn't really user-facing and has a "best effort" quality of service.

Suppose that our Git synchronizer has a memory leak. We need to ensure that the Git synchronizer cannot use up memory that we want to use for our web server, since this can affect web server performance or even crash the server.

This sort of resource isolation is exactly the sort of thing that containers are designed to accomplish. By separating the two applications into two separate containers, we can ensure reliable web server operation.

Of course, the two containers are quite symbiotic; it makes no sense to schedule the web server on one machine and the Git synchronizer on another. Consequently, Kubernetes groups multiple containers into a single atomic unit called a *Pod*. (The name goes with the whale theme of Docker containers, since a Pod is also a group of whales.)

Though the concept of such sidecars seemed controversial or confusing when it was first introduced in Kubernetes, it has subsequently been adopted by a variety of different applications to deploy their infrastructure. For example, several Service Mesh implementations use sidecars to inject network management into an application's Pod.

Pods in Kubernetes

A Pod represents a collection of application containers and volumes running in the same execution environment. Pods, not containers, are the smallest deployable artifact in a Kubernetes cluster. This means all of the containers in a Pod always land on the same machine.

Each container within a Pod runs in its own cgroup, but they share a number of Linux namespaces.

Applications running in the same Pod share the same IP address and port space (network namespace), have the same hostname (UTS namespace), and can communicate using native interprocess communication channels over System V IPC or POSIX message queues (IPC namespace). However, applications in different Pods are isolated from each other; they have different IP addresses, different hostnames, and more. Containers in different Pods running on the same node might as well be on different servers.

Thinking with Pods

One of the most common questions that occurs in the adoption of Kubernetes is "What should I put in a Pod?"

Sometimes people see Pods and think, "Aha! A WordPress container and a MySQL database container should be in the same Pod." However, this kind of Pod is actually an example of an anti-pattern for Pod construction. There are two reasons for this. First, WordPress and its database are not truly symbiotic. If the WordPress container and the database container land on different machines, they still can work together quite effectively, since they communicate over a network connection. Secondly, you don't necessarily want to scale WordPress and the database as a unit. WordPress itself is mostly stateless, and thus you may want to scale your WordPress frontends in response to frontend load by creating more WordPress Pods. Scaling a MySQL database is much trickier, and you would be much more likely to increase the resources dedicated to a single MySQL Pod. If you group the WordPress and MySQL containers together in a single Pod, you are forced to use the same scaling strategy for both containers, which doesn't fit well.

In general, the right question to ask yourself when designing Pods is, "Will these containers work correctly if they land on different machines?" If the answer is "no," a Pod is the correct grouping for the containers. If the answer is "yes," multiple Pods is probably the correct solution. In the example at the beginning of this chapter, the two containers interact via a local filesystem. It would be impossible for them to operate correctly if the containers were scheduled on different machines.

In the remaining sections of this chapter, we will describe how to create, introspect, manage, and delete Pods in Kubernetes.

The Pod Manifest

Pods are described in a Pod manifest. The Pod manifest is just a text-file representation of the Kubernetes API object. Kubernetes strongly believes in *declarative configuration*. Declarative configuration means that you write down the desired state of the world in a configuration and then submit that configuration to a service that takes actions to ensure the desired state becomes the actual state.

Declarative configuration is different from *imperative configuration*, where you simply take a series of actions (e.g., `apt-get install foo`) to modify the world. Years of production experience have taught us that maintaining a written record of the system's desired state leads to a more manageable, reliable system. Declarative configuration enables numerous advantages, including code review for configurations as well as documenting the current state of the world for distributed teams. Additionally, it is the basis for all of the self-healing behaviors in Kubernetes that keep applications running without user action.

The Kubernetes API server accepts and processes Pod manifests before storing them in persistent storage (`etcd`). The scheduler also uses the Kubernetes API to find Pods that haven't been scheduled to a node. The scheduler then places the Pods onto nodes depending on the resources and other constraints expressed in the Pod manifests. Multiple Pods can be placed on the same machine as long as there are sufficient resources. However, scheduling multiple replicas of the same application onto the same machine is worse for reliability, since the machine is a single failure domain. Consequently, the Kubernetes scheduler tries to ensure that Pods from the same application are distributed onto different machines for reliability in the presence of such failures. Once scheduled to a node, Pods don't move and must be explicitly destroyed and rescheduled.

Multiple instances of a Pod can be deployed by repeating the workflow described here. However, ReplicaSets (???) are better suited for running multiple instances of a Pod. (It turns out they're also better at running a single Pod, but we'll get into that later.)

Creating a Pod

The simplest way to create a Pod is via the imperative `kubectl run` command. For example, to run our same `kuard` server, use:

```
$ kubectl run kuard --generator=run-pod/v1 \
  --image=gcr.io/kuar-demo/kuard-amd64:blue
```

You can see the status of this Pod by running:

```
$ kubectl get pods
```

You may initially see the container as `Pending`, but eventually you will see it transition to `Running`, which means that the Pod and its containers have been successfully created.

For now, you can delete this Pod by running:

```
$ kubectl delete pods/kuard
```

We will now move on to writing a complete Pod manifest by hand.

Creating a Pod Manifest

Pod manifests can be written using YAML or JSON, but YAML is generally preferred because it is slightly more human-editable and has the ability to add comments. Pod manifests (and other Kubernetes API objects) should really be treated in the same way as source code, and things like comments help explain the Pod to new team members who are looking at them for the first time.

Pod manifests include a couple of key fields and attributes: namely a `metadata` section for describing the Pod and its labels, a `spec` section for describing volumes, and a list of containers that will run in the Pod.

In ??? we deployed `kuard` using the following Docker command:

```
$ docker run -d --name kuard \
  --publish 8080:8080 \
  gcr.io/kuar-demo/kuard-amd64:blue
```

A similar result can be achieved by instead writing Example 1-1 to a file named *kuard-pod.yaml* and then using `kubectl` commands to load that manifest to Kubernetes.

Example 1-1. kuard-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:blue
      name: kuard
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Running Pods

In the previous section we created a Pod manifest that can be used to start a Pod running `kuard`. Use the `kubectl apply` command to launch a single instance of `kuard`:

```
$ kubectl apply -f kuard-pod.yaml
```

The Pod manifest will be submitted to the Kubernetes API server. The Kubernetes system will then schedule that Pod to run on a healthy node in the cluster, where it will be monitored by the `kubelet` daemon process. Don't worry if you don't understand all the moving parts of Kubernetes right now; we'll get into more details throughout the book.

Listing Pods

Now that we have a Pod running, let's go find out some more about it. Using the `kubectl` command-line tool, we can list all Pods running in the cluster. For now, this should only be the single Pod that we created in the previous step:

```
$ kubectl get pods
NAME       READY     STATUS    RESTARTS   AGE
kuard      1/1       Running   0          44s
```

You can see the name of the Pod (`kuard`) that we gave it in the previous YAML file. In addition to the number of ready containers (`1/1`), the output also shows the status, the number of times the Pod was restarted, as well as the age of the Pod.

If you ran this command immediately after the Pod was created, you might see:

```
NAME       READY     STATUS    RESTARTS   AGE
kuard      0/1       Pending   0          1s
```

The `Pending` state indicates that the Pod has been submitted but hasn't been scheduled yet.

If a more significant error occurs (e.g., an attempt to create a Pod with a container image that doesn't exist), it will also be listed in the status field.

By default, the `kubectl` command-line tool tries to be concise in the information it reports, but you can get more information via command-line flags. Adding `-o wide` to any `kubectl` command will print out slightly more information (while still trying to keep the information to a single line). Adding `-o json` or `-o yaml` will print out the complete objects in JSON or YAML, respectively.

Pod Details

Sometimes, the single-line view is insufficient because it is too terse. Additionally, Kubernetes maintains numerous events about Pods that are present in the event stream, not attached to the Pod object.

To find out more information about a Pod (or any Kubernetes object) you can use the `kubectl describe` command. For example, to describe the Pod we previously created, you can run:

```
$ kubectl describe pods kuard
```

This outputs a bunch of information about the Pod in different sections. At the top is basic information about the Pod:

```
Name:           kuard
Namespace:      default
Node:           node1/10.0.15.185
Start Time:     Sun, 02 Jul 2017 15:00:38 -0700
Labels:         <none>
Annotations:    <none>
Status:         Running
IP:             192.168.199.238
Controllers:    <none>
```

Then there is information about the containers running in the Pod:

```
Containers:
  kuard:
    Container ID:  docker://055095…
    Image:         gcr.io/kuar-demo/kuard-amd64:blue
    Image ID:      docker-pullable://gcr.io/kuar-demo/kuard-amd64@sha256:a580…
    Port:          8080/TCP
    State:         Running
      Started:     Sun, 02 Jul 2017 15:00:41 -0700
    Ready:         True
    Restart Count: 0
    Environment:   <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-cg5f5
(ro)
```

Finally, there are events related to the Pod, such as when it was scheduled, when its image was pulled, and if/when it had to be restarted because of failing health checks:

```
Events:
  Seen From              SubObjectPath          Type      Reason    Message
  ---- ----              -------------          --------  ------    -------
  50s  default-scheduler                        Normal    Scheduled Success…
  49s  kubelet, node1    spec.containers{kuard} Normal    Pulling   pulling…
  47s  kubelet, node1    spec.containers{kuard} Normal    Pulled    Success…
  47s  kubelet, node1    spec.containers{kuard} Normal    Created   Created…
  47s  kubelet, node1    spec.containers{kuard} Normal    Started   Started…
```

Deleting a Pod

When it is time to delete a Pod, you can delete it either by name:

```
$ kubectl delete pods/kuard
```

or using the same file that you used to create it:

```
$ kubectl delete -f kuard-pod.yaml
```

When a Pod is deleted, it is *not* immediately killed. Instead, if you run `kubectl get pods` you will see that the Pod is in the `Terminating` state. All Pods have a termination *grace period*. By default, this is 30 seconds. When a Pod is transitioned to `Terminating` it no longer receives new requests. In a serving scenario, the grace period is important for reliability because it allows the Pod to finish any active requests that it may be in the middle of processing before it is terminated.

It's important to note that when you delete a Pod, any data stored in the containers associated with that Pod will be deleted as well. If you want to persist data across multiple instances of a Pod, you need to use `PersistentVolumes`, described at the end of this chapter.

Accessing Your Pod

Now that your Pod is running, you're going to want to access it for a variety of reasons. You may want to load the web service that is running in the Pod. You may want to view its logs to debug a problem that you are seeing, or even execute other commands in the context of the Pod to help debug. The following sections detail various ways that you can interact with the code and data running inside your Pod.

Using Port Forwarding

Later in the book, we'll show how to expose a service to the world or other containers using load balancers—but oftentimes you simply want to access a specific Pod, even if it's not serving traffic on the internet.

To achieve this, you can use the port-forwarding support built into the Kubernetes API and command-line tools.

When you run:

```
$ kubectl port-forward kuard 8080:8080
```

a secure tunnel is created from your local machine, through the Kubernetes master, to the instance of the Pod running on one of the worker nodes.

As long as the `port-forward` command is still running, you can access the Pod (in this case the `kuard` web interface) at *http://localhost:8080*.

Getting More Info with Logs

When your application needs debugging, it's helpful to be able to dig deeper than `describe` to understand what the application is doing. Kubernetes provides two commands for debugging running containers. The `kubectl logs` command downloads the current logs from the running instance:

```
$ kubectl logs kuard
```

Adding the `-f` flag will cause you to continuously stream logs.

The `kubectl logs` command always tries to get logs from the currently running container. Adding the `--previous` flag will get logs from a previous instance of the container. This is useful, for example, if your containers are continuously restarting due to a problem at container startup.

While using `kubectl logs` is useful for one-off debugging of containers in production environments, it's generally useful to use a log aggregation service. There are several open source log aggregation tools, like `fluentd` and `elasticsearch`, as well as numerous cloud logging providers. Log aggregation services provide greater capacity for storing a longer duration of logs, as well as rich log searching and filtering capabilities. Finally, they often provide the ability to aggregate logs from multiple Pods into a single view.

Running Commands in Your Container with exec

Sometimes logs are insufficient, and to truly determine what's going on you need to execute commands in the context of the container itself. To do this you can use:

```
$ kubectl exec kuard date
```

You can also get an interactive session by adding the `-it` flags:

```
$ kubectl exec -it kuard ash
```

Copying Files to and from Containers

At times you may need to copy files from a remote container to a local machine for more in-depth exploration. For example, you can use a tool like Wireshark to visualize `tcpdump` packet captures. Suppose you had a file called */captures/capture3.txt* inside a container in your Pod. You could securely copy that file to your local machine by running:

```
$ kubectl cp <pod-name>:/captures/capture3.txt ./capture3.txt
```

Other times you may need to copy files from your local machine into a container. Let's say you want to copy *$HOME/config.txt* to a remote container. In this case, you can run:

```
$ kubectl cp $HOME/config.txt <pod-name>:/config.txt
```

Generally speaking, copying files into a container is an anti-pattern. You really should treat the contents of a container as immutable. But occasionally it's the most immediate way to stop the bleeding and restore your service to health, since it is quicker than building, pushing, and rolling out a new image. Once the bleeding is stopped, however, it is critically important that you immediately go and do the image build and rollout, or you are guaranteed to forget the local change that you made to your container and overwrite it in the subsequent regularly scheduled rollout.

Health Checks

When you run your application as a container in Kubernetes, it is automatically kept alive for you using a *process health check*. This health check simply ensures that the main process of your application is always running. If it isn't, Kubernetes restarts it.

However, in most cases, a simple process check is insufficient. For example, if your process has deadlocked and is unable to serve requests, a process health check will still believe that your application is healthy since its process is still running.

To address this, Kubernetes introduced health checks for application *liveness*. Liveness health checks run application-specific logic (e.g., loading a web page) to verify that the application is not just still running, but is functioning properly. Since these liveness health checks are application-specific, you have to define them in your Pod manifest.

Liveness Probe

Once the `kuard` process is up and running, we need a way to confirm that it is actually healthy and shouldn't be restarted. Liveness probes are defined per container, which means each container inside a Pod is health-checked separately. In Example 1-2, we add a liveness probe to our `kuard` container, which runs an HTTP request against the `/healthy` path on our container.

Example 1-2. kuard-pod-health.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:blue
      name: kuard
      livenessProbe:
        httpGet:
          path: /healthy
          port: 8080
        initialDelaySeconds: 5
        timeoutSeconds: 1
        periodSeconds: 10
        failureThreshold: 3
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

The preceding Pod manifest uses an `httpGet` probe to perform an HTTP `GET` request against the `/healthy` endpoint on port 8080 of the `kuard` container. The probe sets an `initialDelaySeconds` of `5`, and thus will not be called until 5 seconds after all the containers in the Pod are created. The probe must respond within the 1-second timeout, and the HTTP status code must be equal to or greater than 200 and less than 400 to be considered successful. Kubernetes will call the probe every 10 seconds. If more than three consecutive probes fail, the container will fail and restart.

You can see this in action by looking at the `kuard` status page. Create a Pod using this manifest and then port-forward to that Pod:

```
$ kubectl apply -f kuard-pod-health.yaml
$ kubectl port-forward kuard 8080:8080
```

Point your browser to *http://localhost:8080*. Click the "Liveness Probe" tab. You should see a table that lists all of the probes that this instance of `kuard` has received. If you click the "Fail" link on that page, `kuard` will start to fail health checks. Wait long enough and Kubernetes will restart the container. At that point the display will reset and start over again. Details of the restart can be found with `kubectl describe pods kuard`. The "Events" section will have text similar to the following:

```
Killing container with id docker://2ac946...:pod "kuard_default(9ee84...)"
container "kuard" is unhealthy, it will be killed and re-created.
```

While the default response to a failed liveness check is to restart the Pod, the actual behavior is governed by the Pod's `restartPolicy`. There are three options for the restart policy: `Always` (the default), `OnFailure` (restart only on liveness failure or nonzero process exit code), or `Never`.

Readiness Probe

Of course, liveness isn't the only kind of health check we want to perform. Kubernetes makes a distinction between *liveness* and *readiness*. Liveness determines if an application is running properly. Containers that fail liveness checks are restarted. Readiness describes when a container is ready to serve user requests. Containers that fail readiness checks are removed from service load balancers. Readiness probes are configured similarly to liveness probes. We explore Kubernetes services in detail in ???.

Combining the readiness and liveness probes helps ensure only healthy containers are running within the cluster.

Types of Health Checks

In addition to HTTP checks, Kubernetes also supports `tcpSocket` health checks that open a TCP socket; if the connection is successful, the probe succeeds. This style of probe is useful for non-HTTP applications; for example, databases or other non–HTTP-based APIs.

Finally, Kubernetes allows `exec` probes. These execute a script or program in the context of the container. Following typical convention, if this script returns a zero exit code, the probe succeeds; otherwise, it fails. `exec` scripts are often useful for custom application validation logic that doesn't fit neatly into an HTTP call.

Resource Management

Most people move into containers and orchestrators like Kubernetes because of the radical improvements in image packaging and reliable deployment they provide. In addition to application-oriented primitives that simplify distributed system development, equally important is the ability to increase the overall utilization of the compute nodes that make up the cluster. The basic cost of operating a machine, either virtual or physical, is basically constant regardless of whether it is idle or fully loaded. Consequently, ensuring that these machines are maximally active increases the efficiency of every dollar spent on infrastructure.

Generally speaking, we measure this efficiency with the *utilization* metric. Utilization is defined as the amount of a resource actively being used divided by the amount of a resource that has been purchased. For example, if you purchase a one-core machine, and your application uses one-tenth of a core, then your utilization is 10%.

With scheduling systems like Kubernetes managing resource packing, you can drive your utilization to greater than 50%.

To achieve this, you have to tell Kubernetes about the resources your application requires, so that Kubernetes can find the optimal packing of containers onto purchased machines.

Kubernetes allows users to specify two different resource metrics. Resource *requests* specify the minimum amount of a resource required to run the application. Resource *limits* specify the maximum amount of a resource that an application can consume. Both of these resource definitions are described in greater detail in the following sections.

Resource Requests: Minimum Required Resources

With Kubernetes, a Pod requests the resources required to run its containers. Kubernetes guarantees that these resources are available to the Pod. The most commonly requested resources are CPU and memory, but Kubernetes has support for other resource types as well, such as GPUs and more.

For example, to request that the `kuard` container lands on a machine with half a CPU free and gets 128 MB of memory allocated to it, we define the Pod as shown in Example 1-3.

Example 1-3. kuard-pod-resreq.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:blue
      name: kuard
      resources:
        requests:
          cpu: "500m"
          memory: "128Mi"
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Resources are requested per container, not per Pod. The total resources requested by the Pod is the sum of all resources requested by all containers in the Pod. The reason for this is that in many cases the different containers have very different CPU requirements. For example, in the web server and data synchronizer Pod, the web server is user-facing and likely needs a great deal of CPU, while the data synchronizer can make do with very little.

Request limit details

Requests are used when scheduling Pods to nodes. The Kubernetes scheduler will ensure that the sum of all requests of all Pods on a node does not exceed the capacity of the node. Therefore, a Pod is guaranteed to have at least the requested resources when running on the node. Importantly, "request" specifies a minimum. It does not specify a maximum cap on the resources a Pod may use. To explore what this means, let's look at an example.

Imagine that we have container whose code attempts to use all available CPU cores. Suppose that we create a Pod with this container that requests 0.5 CPU. Kubernetes schedules this Pod onto a machine with a total of 2 CPU cores.

As long as it is the only Pod on the machine, it will consume all 2.0 of the available cores, despite only requesting 0.5 CPU.

If a second Pod with the same container and the same request of 0.5 CPU lands on the machine, then each Pod will receive 1.0 cores.

If a third identical Pod is scheduled, each Pod will receive 0.66 cores. Finally, if a fourth identical Pod is scheduled, each Pod will receive the 0.5 core it requested, and the node will be at capacity.

CPU requests are implemented using the `cpu-shares` functionality in the Linux kernel.

Memory requests are handled similarly to CPU, but there is an important difference. If a container is over its memory request, the OS can't just remove memory from the process, because it's been allocated. Consequently, when the system runs out of memory, the `kubelet` terminates containers whose memory usage is greater than their requested memory. These containers are automatically restarted, but with less available memory on the machine for the container to consume.

Since resource requests guarantee resource availability to a Pod, they are critical to ensuring that containers have sufficient resources in high-load situations.

Capping Resource Usage with Limits

In addition to setting the resources required by a Pod, which establishes the minimum resources available to the Pod, you can also set a maximum on a Pod's resource usage via resource *limits*.

In our previous example we created a `kuard` Pod that requested a minimum of 0.5 of a core and 128 MB of memory. In the Pod manifest in Example 1-4, we extend this configuration to add a limit of 1.0 CPU and 256 MB of memory.

Example 1-4. kuard-pod-reslim.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:blue
      name: kuard
      resources:
        requests:
          cpu: "500m"
          memory: "128Mi"
        limits:
          cpu: "1000m"
          memory: "256Mi"
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

When you establish limits on a container, the kernel is configured to ensure that consumption cannot exceed these limits. A container with a CPU limit of 0.5 cores will only ever get 0.5 cores, even if the CPU is otherwise idle. A container with a memory limit of 256 MB will not be allowed additional memory (e.g., `malloc` will fail) if its memory usage exceeds 256 MB.

Persisting Data with Volumes

When a Pod is deleted or a container restarts, any and all data in the container's filesystem is also deleted. This is often a good thing, since you don't want to leave around cruft that happened to be written by your stateless web application. In other cases, having access to persistent disk storage is an important part of a healthy application. Kubernetes models such persistent storage.

Using Volumes with Pods

To add a volume to a Pod manifest, there are two new stanzas to add to our configuration. The first is a new `spec.volumes` section. This array defines all of the volumes that may be accessed by containers in the Pod manifest. It's important to note that not all containers are required to mount all volumes defined in the Pod. The second addition is the `volumeMounts` array in the container definition. This array defines the volumes that are mounted into a particular container, and the path where each volume should be mounted. Note that two different containers in a Pod can mount the same volume at different mount paths.

The manifest in Example 1-5 defines a single new volume named `kuard-data`, which the `kuard` container mounts to the `/data` path.

Example 1-5. kuard-pod-vol.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  volumes:
    - name: "kuard-data"
      hostPath:
        path: "/var/lib/kuard"
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:blue
      name: kuard
      volumeMounts:
        - mountPath: "/data"
          name: "kuard-data"
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Different Ways of Using Volumes with Pods

There are a variety of ways you can use data in your application. The following are a few, and the recommended patterns for Kubernetes.

Communication/synchronization

In the first example of a Pod, we saw how two containers used a shared volume to serve a site while keeping it synchronized to a remote Git location. To achieve this, the Pod uses an `emptyDir` volume. Such a volume is scoped to the Pod's lifespan, but it can be shared between two containers, forming the basis for communication between our Git sync and web serving containers.

Cache

An application may use a volume that is valuable for performance, but not required for correct operation of the application. For example, perhaps the application keeps prerendered thumbnails of larger images. Of course, they can be reconstructed from the original images, but that makes serving the thumbnails more expensive. You want such a cache to survive a container restart due to a health-check failure, and thus `emptyDir` works well for the cache use case as well.

Persistent data

Sometimes you will use a volume for truly persistent data—data that is independent of the lifespan of a particular Pod, and should move between nodes in the cluster if a node fails or a Pod moves to a different machine for some reason. To achieve this, Kubernetes supports a wide variety of remote network storage volumes, including widely supported protocols like NFS and iSCSI as well as cloud provider network storage like Amazon's Elastic Block Store, Azure's Files and Disk Storage, as well as Google's Persistent Disk.

Mounting the host filesystem

Other applications don't actually need a persistent volume, but they do need some access to the underlying host filesystem. For example, they may need access to the */dev* filesystem in order to perform raw block-level access to a device on the system. For these cases, Kubernetes supports the `hostPath` volume, which can mount arbitrary locations on the worker node into the container.

The previous example uses the `hostPath` volume type. The volume created is */var/lib/kuard* on the host.

Persisting Data Using Remote Disks

Oftentimes, you want the data a Pod is using to stay with the Pod, even if it is restarted on a different host machine.

To achieve this, you can mount a remote network storage volume into your Pod. When using network-based storage, Kubernetes automatically mounts and unmounts the appropriate storage whenever a Pod using that volume is scheduled onto a particular machine.

There are numerous methods for mounting volumes over the network. Kubernetes includes support for standard protocols such as NFS and iSCSI as well as cloud provider–based storage APIs for the major cloud providers (both public and private). In many cases, the cloud providers will also create the disk for you if it doesn't already exist.

Here is an example of using an NFS server:

```
...
# Rest of pod definition above here
volumes:
    - name: "kuard-data"
      nfs:
        server: my.nfs.server.local
        path: "/exports"
```

Persistent volumes are a deep topic that has many different details: in particular, the manner in which persistent volumes, persistent volume claims, and dynamic volume provisioning work together. There is a more in-depth examination of the subject in ???.

Putting It All Together

Many applications are stateful, and as such we must preserve any data and ensure access to the underlying storage volume regardless of what machine the application runs on. As we saw earlier, this can be achieved using a persistent volume backed by network-attached storage. We also want to ensure that a healthy instance of the application is running at all times, which means we want to make sure the container running `kuard` is ready before we expose it to clients.

Through a combination of persistent volumes, readiness and liveness probes, and resource restrictions, Kubernetes provides everything needed to run stateful applications reliably. Example 1-6 pulls this all together into one manifest.

Example 1-6. kuard-pod-full.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  volumes:
    - name: "kuard-data"
      nfs:
        server: my.nfs.server.local
        path: "/exports"
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:blue
      name: kuard
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
      resources:
        requests:
          cpu: "500m"
          memory: "128Mi"
        limits:
          cpu: "1000m"
          memory: "256Mi"
      volumeMounts:
        - mountPath: "/data"
          name: "kuard-data"
      livenessProbe:
```

```
      httpGet:
        path: /healthy
        port: 8080
      initialDelaySeconds: 5
      timeoutSeconds: 1
      periodSeconds: 10
      failureThreshold: 3
    readinessProbe:
      httpGet:
        path: /ready
        port: 8080
      initialDelaySeconds: 30
      timeoutSeconds: 1
      periodSeconds: 10
      failureThreshold: 3
```

Summary

Pods represent the atomic unit of work in a Kubernetes cluster. Pods are comprised of one or more containers working together symbiotically. To create a Pod, you write a Pod manifest and submit it to the Kubernetes API server by using the command-line tool or (less frequently) by making HTTP and JSON calls to the server directly.

Once you've submitted the manifest to the API server, the Kubernetes scheduler finds a machine where the Pod can fit and schedules the Pod to that machine. Once scheduled, the `kubelet` daemon on that machine is responsible for creating the containers that correspond to the Pod, as well as performing any health checks defined in the Pod manifest.

Once a Pod is scheduled to a node, no rescheduling occurs if that node fails. Additionally, to create multiple replicas of the same Pod you have to create and name them manually. In a later chapter we introduce the ReplicaSet object and show how you can automate the creation of multiple identical Pods and ensure that they are recreated in the event of a node machine failure.

CHAPTER 2

Accessing Kubernetes from Common Programming Languages

Though most of this book is dedicated to using declarative YAML configurations either directly via `kubectl` or through tools like Helm, there are situations when it is necessary to interact with the Kubernetes API directly from a programming language. For example the authors of the Helm tool itself needed to write that application in a programming language. More generally this is common if you need to write some additional tool, like a `kubectl` plugin, or a more complex piece of code, like a Kubernetes operator.

While much of the Kubernetes ecosystem is written in the Go programming language, and indeed the Go client for Kubernetes has the richest and most extensive client. There are a high quality clients for most common programming languages (and even some uncommon ones as well). Because there is so much documentation, code and examples of how to use the Go client already out on the internet, this chapter will cover the basics of interacting with the Kubernetes API server with examples in Python, Java and C#.

The Kubernetes API: A client's perspective

At the end of the day, the Kubernetes API server is just an HTTP(S) server and that is exactly how each client library perceives it. Though each client has a lot of additional logic that implements the various API calls and serializes to and from JSON. Given this, you might be tempted to simply use a plain HTTP client to work with the Kubernetes APIs, but the client libraries wrap these various HTTP calls into meaningful APIs (e.g. `readNamespacedPod(...)`) that make your code more readable, and meaningful typed object-models (e.g. `Deployment`) which facilitate static type-checking and therefor fewer bugs. Perhaps more importantly, the client libraries also implement Kubernetes specific capabilities like loading authorization information from a "Kubeconfig" file or from a Pod's environment. The clients also provide implementations of the non RESTful parts of the Kubernetes API surface area like port-forward, logs and watches. We'l describe these advanced capabilities in later sections.

OpenAPI and generated client libraries

The set of resources and functions in the Kubernetes API is huge. There are many different resources in different api groups and many different operations on each of these resources. Keeping up with all of these different resources and resource versions would be a massive (and unmistakeably boring) undertaking if developers had to hand-author all of these API calls. Especially when considering that clients have to be hand-written across each of the different programming languages. Instead the clients take a different approach and the basics of interacting with the Kubernetes API server are all generated by a computer program that is sort of like a compiler in reverse. The code generator for the API clients takes a data specification for the Kubernetes API and uses this specification to generate a client for a specific languag.

The Kubernetes API is expressed in a format known as OpenAPI (or previously as Swagger) which is the most common schema for representing REST-ful APIs. To give you a sense of the size of the Kubernetes API, the OpenAPI specification found on GitHub (*https://github.com/kubernetes/kubernetes/blob/master/api/openapi-spec/swagger.json*) is over four megabytes in size. That's a pretty big text file! The official Kubernetes client libraries are all generated using the same core code generation logic, which can be found on GitHub at *https://github.com/kubernetes-client/gen*. It is unlikely that you will actually have to generate the client libraries yourself, but nonetheless it is useful to understand the process by which these libraries are created. In particular, because most of the client code is generated, updates and fixes can't be made directly in the generated client code, since it would be overwritten the next time the API was generated. Instead, when an error in a client is found, fixes need to be made to either the OpenAPI specification (if the error is in the specification itself) or in the code generator (if the error is in the generated code). Although this process can seem excessively complex, it is the only way that a small number of Kubernetes client authors can keep up with the breadth of the Kubernetes API.

But what about `kubectl x ...`?

When you start implementing your own logic for interacting with the Kubernetes API, it probably won't be long before you find yourself asking how to do `kubectl x`. Most people start with the `kubectl` tool when they begin learning Kubernetes and they consequently expect that there is a 1-1 mapping between the capabilities in kubectl and the Kubernetes API. While it is the case that some commands (e.g. `kubectl get pods`) are directly represented in the Kubernetes API. Most of the more sophisticated features are actually a larger number of API calls with complex logic in the `kubectl` tool.

This balance between client side and server side features has been a design trade-off since the beginning of Kubernetes. Many features that are now present in the API server began as client side implementions in `kubectl`. For example, the rollout capabilities now implemented on the server by the Deployment resource were previously implemented in the client. Likewise until very recently `kubectl apply ...` was only available within the command line tool, but was migrated to the server as the server side apply capabilities that will be discussed later in this chapter.

Despite the general trajectory towards server side implementations, there are still significant capabilities which remain in the client. For these capabilities, there has been significant work in some of the clients (e.g. the `io.kubernetes.client.extended.kubectl` package in the Java client) that attempt to emulate many of the `kubectl` capabilities.

If you can't find the functionality that you are looking for in your client library, a useful trick is to add the `--v=10` flag to your `kubectl` command which will turn on verbose logging including all of the HTTP requests and responses sent to the Kubernetes API server. You can use this logging to reconstruct much of what `kubectl` is doing. If you still need to dig deeper, the kubectl source code is also available within the Kubernetes repository.

Programming the Kubernetes API

Now you have a deeper perspective about how the Kubernetes API works and the client and server interact. In the following sections we'll go through how to authenticated to the Kubernetes API server, interact with resources and finally close with advanced topics from writing operators to interacting with Pods for interactive operations.

Installing the client libraries

Before you can start programming with the Kubernetes API you need to find the client libraries. We will be using the official client libraries produced by the Kubernetes project itself, though there are also a number of high-quality clients developed as independent projects. The client libraries are all hosted under the `kubernetes-client` repository on Github:

- Python
- Java
- Javascript
- .NET

Each of these projects features a compatability matrix to show which versions of the client work with which versions of the Kubernetes API and also give instructions for installing the libraries using the package managers (e.g. `npm`) associated with a particular programming language.

Authenticating to the Kubernetes API

The Kubernetes API server wouldn't be very safe if it allowed anyone in the world to access it and read or write the resources that it orchestrates. Consequentally the first step in programming the Kubernetes API is connecting to it and identifying yourself for authentication. Because the API server is an HTTP server at it's core, these methods of authentication are core HTTP authentication methods. The very first implementations of Kubernetes used basic HTTP authentication via a user and password combination, but this approach has been deprecated in favor of more modern authentication infrastructure.

If you have been using the kubectl command line tool for your interactions with Kubernetes, you may not have considered the implementation details of authentication. Fortunately the client libraries generally make it easy to connect to the API. However, a basic understanding of how Kubernetes authentication works is still useful for debugging when things go wrong.

There are two basic ways that the `kubectl` tool and clients obtain authentication information: * From a "kubeconfig" file * From the context of a Pod within the Kubernetes cluster.

Code that is not running inside a Kubernetes cluster requires a "kubeconfig" file to provide the necessary information for authentication. By default the client searches for this file in `${HOME}/.kube/config` or the `$KUBECONFIG` environment variables. If the `KUBECONFIG` variable is present it takes precedence over any config located in the default home location. The kubeconfig file contains all of the information necessary to access the Kubernetes API server. The clients all have easy to use calls to create a client either from the default locations, or from a Kubeconfig file supplied in the code itself:

Java

```
ApiClient client = Config.defaultClient();
Configuration.setDefaultApiClient(client);
```

Python

```
config.load_kube_config()
```

.NET

```
var config = KubernetesClientConfiguration.BuildDefaultConfig();
var client = new Kubernetes(config);
```

Authentication for many cloud providers occurs via an external executable which knows how to generate a token for the Kubernetes cluster. This executable is often installed as part of the cloud providers command line tooling. When you write code to interact with the Kubernetes API, you need to make sure that this executable is also available in the context where the code is running so that it can be executed to obtain the token.

Within the context of a Pod in a Kubernetes cluster, the code running in the Pod has access to a Kubernetes service account which is associated with that Pod. The files containing the relevant token and certificate authority are placed into the Pod by Kubernetes as a volume when the Pod is created and within a Kubernetes cluster, the api server is always available at a fixed DNS name, generally `kubernetes`. Because all of the necessary data is present in the Pod a kubeconfig file is unnecessary and the client can synthesisze it's configuration from its context. The clients all have easy to use calls to create such an "in cluster" client:

Java

```
ApiClient client = ClientBuilder.cluster().build();
Configuration.setDefaultApiClient(client);
```

Python

```
config.load_incluster_config()
```

.NET

```
var config = KubernetesClientConfiguration.InClusterConfig()
var client = new Kubernetes(config);
```

The default service account associated with Pods has minimal roles (RBAC) granted to it. This means that by default the code running in a Pod can't do much with the Kubernetes API. If you are getting authorization errors you may need to adjust the service account to one that is specific to your code and has access to the necessary roles in the cluster.

Accessing the Kubernetes API

The most common ways that people interact with the Kubernetes API is via basic operations like creating, listing and deleting resources. Because all of the clients are generated from the same OpenAPI specification they all follow the same rough pattern. Before diving into the code, there are a couple more details of the Kubernetes API that are necessary to understand.

The first is that in Kubernetes there is a distinction between "namespaced" and "cluster" level resources. *Namespaced* resources exist within a Kubernetes namespace, for example a Pod or Deployment may exist in the `kube-system` namespace. *Cluster-level* resources exist once throughout the entire cluster. The most obvious example of such a resource is a Namespace, but other cluster-level resources include CustomResourceDefinitions and ClusterRoleBindings. This distinction is important because it is preserved in the function calls that you use to access the resources. For example, to list pods in the `default` namespace in Python you would write `api.list_namespaced_pods('default')`. To list Namespaces you would write `api.list_namespaces()`.

The second concept you need to understand is an *API group*. In Kubernetes all of the resources are grouped into different sets of APIs. This is largely hidden from users of the `kubectl` tool, though you may have seen it within the `apiVersion` field in a YAML specification of a Kubernetes object. When programming against the Kubernetes API this grouping becomes important, because often each API group has its own client for interacting with that set of resources. For example, create a client to interact with a Deployment resource (which exists in the "apps/v1" API group and version) you create a `new AppsV1Api()` object which knows how to interact with all resources in the `apps/v1` API group and version. An example of how to create a client for an API group is shown in the following section.

Putting it all together: Listing & Creating Pods in Python, Java and .NET

We're now ready to actually write some code. First we begin by creating a client object, then we use that to list the Pods in the "default" namespace, here is code to do that in Python, Java and .NET.

Python

```
config.load_kube_config()
api = client.CoreV1Api()
pod_list = api.list_namespaced_pod('default')
```

Java

```
ApiClient client = Config.defaultClient();
Configuration.setDefaultApiClient(client);
CoreV1Api api = new CoreV1Api();
V1PodList list = api.listNamespacedPod("default");
```

.NET

```
var config = KubernetesClientConfiguration.BuildDefaultConfig();
var client = new Kubernetes(config);
var list = client.ListNamespacedPod("default");
```

Once you have figured out how to list, read and delete objects, the next common task is creating new objects. The API call to create the object is easy enough to figure out (e.g. `create_namespaced_pod` in Python), but actually defining the new Pod resources can be more complicated.

Here's how you create a Pod in Python, Java and .NET:

Python

```
container = client.V1Container(
     name="myapp",
     image="my_cool_image:v1",
 )


pod = client.V1Pod(
    metadata = client.V1ObjectMeta(
      name="myapp",
    ),
    spec=client.V1PodSpec(containers=[container]),
)
```

Java

```
V1Pod pod =
    new V1PodBuilder()
        .withNewMetadata().withName("myapp").endMetadata()
        .withNewSpec()
          .addNewContainer()
            .withName("myapp")
            .withImage("my_cool_image:v1")
          .endContainer()
        .endSpec()
        .build();
```

.NET

```
var pod = new V1Pod()
{
    Metadata = new V1ObjectMeta{ Name = "myapp", },
    Spec = new V1PodSpec
    {
        Containers = new[] { new V1Container() { Name = "myapp", Image =
"my_cool_image:v1", }, },
    }
 };
```

Creating & Patching objects

One thing that you will notice when you explore the client API for Kubernetes is that there are seemingly three different ways to manipulate resources, namely `create`, `replace` and `patch`. All three different verbs represent slightly different semantics for interacting with resources.

::Create as you can tell from the name creates a new resource, however it will fail if the resource already exists. ::Replace replaces an existing resource completely, without looking at the existing resource. When you use `replace` you have to specify a complete resource. ::Patch modifies an existing resource leaving untouched parts of the resource the same as they were. When using patch you use a special patch resource rather than sending the resource (e.g. the Pod) that you are modifying.

Patching a resource can be complicated. In many cases it is easier to just replace it. However, in some cases, especially with large resources, patching the resource can be much more efficient in terms of network bandwidth and API server processing. Additionally, multiple different actors can patch different parts of the resource simultaneously without worrying about write conflicts, which reduces overhead.

To patch a Kubernetes resource you have to create a Patch object representing the change that you want to make to the resource. There are three different formats for this patch supported by Kubernetes: "JSON patch", "JSON merge patch" and "strategic merge patch" The first two patch formats are RFC standards used in other places, the third is a Kubernetes developed patch format. Each of the patch formats has advantages and disadvantages. In these examples we will use JSON Patch because it is the simplest to understand.

Here's how you patch a Deployment to increase the replicas to three:

Java

```
// JSON-patch format
static String jsonPatch =
  "[{\"op\":\"replace\",\"path\":\"/spec/replicas\",\"value\":3}]";
```

```
V1Deployment patched =
          PatchUtils.patch(
              V1Deployment.class,
              () ->
                  api.patchNamespacedDeploymentCall(
                      "my-deployment",
                      "some-namespace",
                      new V1Patch(jsonPatchStr),
                      null,
                      null,
                      null,
                      null,
                      null),
              V1Patch.PATCH_FORMAT_JSON_PATCH,
              api.getApiClient());
```

Python

```
deployment.spec.replicas = 3

api_response = api_instance.patch_namespaced_deployment(
    name="my-deployment",
    namespace="some-namespace",
    body=deployment)
```

.NET

```
var jsonPatch = @"
[{
    ""op"": ""replace"",
    ""path"": ""/spec/replicas"",
    ""value"": 3
}]";

client.PatchNamespacedPod(new V1Patch(patchStr, V1Patch.PatchType.JsonPatch),
"my-deployment", "some-namespace");
```

In each of these code samples, the Deployment resource has been patched to set the number of replicas in the deployment to three.

Watching Kubernetes APIs for changes

Resources in Kubernetes are declarative. They represent the desired state of the system. To make that desired state a reality a program must watch the desired state for changes and take action to make the current state of the world match the desired state.

Because of this pattern, one of the most common tasks when programming against the Kubernetes API is to watch for changes to a resource and then take some action based on those changes. The easiest way to do this is through polling. *Polling* simply calls the list function described above at a constant interval (such as every 60 seconds) and enumerates all of the resources that the code is interested in. While this code is easy to write, it has numerous drawbacks for both the client code and the API server. Polling introduces unnecessary latency, since waiting for the polling cycle to come around introduces delays for changes that occur just after the previous poll completed. Additionally, polling causes heavier load on the API server, because it repeatedly returns resources that haven't changed. While many simple clients begin by using polling, to many clients polling the API server can overload it and add latency.

To solve this problem the Kuberentes API also provides "watch" or event-based semantics. Using a watch call, you can register interest in specific changes with the API server and instead of repeatedly polling, the API server will send notifications whenever a change occurs. In practicle terms, the client performs a hanging GET to the HTTP API Server. The TCP connection that underlies this HTTP request stays open for the duration of the watch and the server writes a response to that stream (but does not close the stream) whenever a change occurs.

From a programmatic perspective, Watch semantics enable event-based programming, changing a `while` loop that repeatedly polls into a collection of callbacks. Here are examples of watching Pods for changes:

Java

```
    ApiClient client = Config.defaultClient();
    CoreV1Api api = new CoreV1Api();

    Watch<V1Namespace> watch =
        Watch.createWatch(
            client,
            api.listNamespacedPodCall(
                "some-namespace",
                null,
                null,
                null,
                null,
                null,
                Integer.MAX_VALUE,
                null,
                null,
                60,
                Boolean.TRUE);
            new TypeToken<Watch.Response<V1Pod>>() {}.getType());

    try {
      for (Watch.Response<V1Pod> item : watch) {
        System.out.printf("%s : %s%n", item.type, item.object.getMetadata().get
Name());
      }
    } finally {
      watch.close();
    }
```

Python

```
config.load_kube_config()
api = client.CoreV1Api()
w = watch.Watch()

for event in w.stream(v1.list_namespaced_pods, "some-namespace"):
  print(event)
```

.NET

```
var config = KubernetesClientConfiguration.BuildConfigFromConfigFile();
var client = new Kubernetes(config);

var watch = client.ListNamespacedPodWithHttpMessagesAsync("default", watch:
true);
using (watch.Watch<V1Pod, V1PodList>((type, item) =>
{
  Console.WriteLine(item);
}
```

In each of these examples, rather than a repetative polling loop, the watch API call delivers each change to a resource to a callback provided by the user. This both reduces latency and load on the Kubernetes API server.

Interacting with Pods

The Kubernetes API also provides functions for directly interacting with the applications running in a Kubernetes Pod. The `kubectl` tool provides a number of commands for interacting with Pods, namely `logs`, `exec` and `port-forward` and it is possible to use each of these from within custom code as well.

Because the `logs`, `exec` and `port-forward` APIs are non-standard in a RESTful sense, they require custom logic in the client libraries and are thus somewhat less consistent between the different clients. Unfortunately there is no option other than learning the implementation for each language.

When getting the logs for a Pod you have to decide if you are going to read the pod logs to get a snapshot of their current state or if you are going to stream them to receive new logs as they happen. If you stream the logs (the equivalent of `kubectl logs -f ...`) then you create an open connection to the API server and new log lines are written to this stream as they are written to the pod. If not, you simply receive the current contents of the logs.

Here's how you both read and stream the logs:

Java

```
V1Pod pod = ...; // some code to define or get a Pod here
PodLogs logs = new PodLogs();
InputStream is = logs.streamNamespacedPodLog(pod);
```

Python

```
config.load_kube_config()
api = client.CoreV1Api()
log = api_instance.read_namespaced_pod_log(name="my-pod", namespace="some-
namespace")
```

.NET

```
IKubernetes client = new Kubernetes(config);
var response = await client.ReadNamespacedPodLogWithHttpMessagesAsync(
    "my-pod", "my-namespace", follow: true);
var stream = response.Body;
```

Another common task is to execute some command within a Pod and get the output of running that task. You can use the `kubectl exec ...` command on the command line. Under the hood the API that implements this is creating a WebSocket connection to the API server. WebSockets enable multiple streams of data (in this case `stdin`, `stdout` & `stderr`) to co-exist on the same HTTP connection. If you've never had experience with WebSockets before, don't worry, the details of interacting with WebSockets are handled by the client libraries.

Here's how you exec the `ls /foo` command in a Pod:

Java

```
ApiClient client = Config.defaultClient();
Configuration.setDefaultApiClient(client);
Exec exec = new Exec();
final Process proc =
  exec.exec("some-namespace", "my-pod", new String[] {"ls", "/foo"}, true,
true /*tty*/);
```

Python

```
cmd = [ 'ls', '/foo' ]
response = stream(
    api_instance.connect_get_namespaced_pod_exec,
    "my-pod",
    "some-namespace",
    command=cmd,
    stderr=True,
    stdin=False,
    stdout=True,
    tty=False)
```

.NET

```
var config = KubernetesClientConfiguration.BuildConfigFromConfigFile();
IKubernetes client = new Kubernetes(config);
var webSocket =
    await client.WebSocketNamespacedPodExecAsync("my-pod", "some-namespace",
"ls /foo", "my-container-name");
var demux = new StreamDemuxer(webSocket);
demux.Start();
var stream = demux.GetStream(1, 1);
```

In addition to running commands in a pod, you can also port-forward network connections from a Pod to code running on the local machine. Like exec, the port forwarded traffic goes over a WebSocket. It is up to your code what it does with this port forwarded socket. You could simply send a single request and receive a response as a string of bytes, or you could build a complete proxy server (like what `kubectl port-forward` does) to serve arbitrary requests through this proxy.

Regardless of what you intend to do with the connection, here's how you set up port-forwarding:

Java

```
PortForward fwd = new PortForward();

List<Integer> ports = new ArrayList<>();
int localPort = 8080;
int targetPort = 8080;
ports.add(targetPort);
final PortForward.PortForwardResult result =
    fwd.forward("some-namespace", "my-pod", ports);
```

Python

```
pf = portforward(
    api_instance.connect_get_namespaced_pod_portforward,
    'my-pod', 'some-namespace',
    ports='8080',
)
```

.NET

```
var config = KubernetesClientConfiguration.BuildConfigFromConfigFile();
IKubernetes client = new Kubernetes(config);
var webSocket = await client.WebSocketNamespacedPodPortForwardAsync("some-
namespace", "my-pod", new int[] {8080}, "v4.channel.k8s.io");
var demux = new StreamDemuxer(webSocket, StreamType.PortForward);
demux.Start();
var stream = demux.GetStream((byte?)0, (byte?)0);
```

Each of these examples creates a connection for from port 8080 in a Pod to port 8080 in your program. The code returns the byte-streams necessary communicating across this port-forwarding channel. You can use these streams for sending and receiving messages.

Conclusion

The Kubernetes API provides rich and powerful functionality for you to write custom code. Writing your applications in the language that best suits a task or a persona shares the power of the orchestration API with as many Kubernetes users as possible. When you're ready to move beyond scripting calls to the `kubectl` executable, the Kubernetes client libraries provide a way to dive deep into the API to build an operator, a monitoring agent, a new user interface or what ever your imagination can dream up.

CHAPTER 3

Policy and Governance for Kubernetes Clusters

Throughout this book we have introduced many different Kubernetes resource types each with a specific purpose. It doesn't take long before the resources on a Kubernetes cluster go from several, representing a single microservice application, to hundreds and thousands for a complete distributed application. In the context of a production cluster it isn't hard to imagine the challenges associated with managing thousands of resources.

In this chapter we introduce the concept of policy and governance. Governance gives cluster administrators the ability to verify and enforce organizational policies for all resources deployed to a Kubernetes cluster. Typical challenges that you can address by adopting policy and governance include ensuring all resources utilize current best practices, are compliant with security policy, or adhere to company conventions.

Whatever your case may be, tooling needs to be flexible and scalable so that all resources defined on a cluster are compliant to defined policy

Why Policy and Governance Matter

There are many different types of policies in Kubernetes, whether it be the `NetworkPolicy` resource or even the `PodSecurityPolicy` resource you don't have to look far. `NetworkPolicy` allows you to specify what network services and endpoints a Pod can connect to. `PodSecurityPolicy` enables fine-grained control over the security elements of a Pod. Both these types of policy resources configure network or container runtime. Unlike these policies, you might be looking for ways to enforce policy before Kubernetes resources are even created. This is the problem policy and governance solves. At this point, you might be thinking "Isn't this what Role Based Access Control does?" however RBAC isn't granular enough to restrict specific fields within resources from being set. If you want to learn more about Role Based Access Control, I recommend you checkout the chapter.

Here are some common examples of policies that clusters administrators may want to configure:

- All containers MUST only come from a specific container registry
- All Pods MUST be labelled with the department name and contact information
- All Pods MUST have both CPU and memory resource limits set
- All Ingress hostnames must be unique across a cluster
- Service MUST not be made available on the Internet
- Containers MUST not listen on privileged ports

Cluster administrators may also want to audit existing resources on a cluster, perform dry-run policy evaluations to deploy new policy, or even mutate a resource based on a set of conditions. For example, applying labels to a Pod if that they aren't present. Having a way for cluster administrators to define policy and perform compliance audits while not interfering with the developers ability to deploy applications to Kubernetes is key to delivering policy and governance tooling. If the resources developers are creating aren't compliant, you need a system to make sure they get the feedback and remediation they need to bring their work into compliance.

Let's take a look at how to achieve policy and governance by leveraging core extensibility components of Kubernetes.

Admission Flow

To understand how policy and governance ensures resources are compliant before they are created your Kubernetes cluster, you must first understand the request flow through the Kubernetes API server. Figure 3-1 depicts the flow of an API request through the API server. Here, we'll focus on Mutation Admission, Validating Admission, and Webhooks.

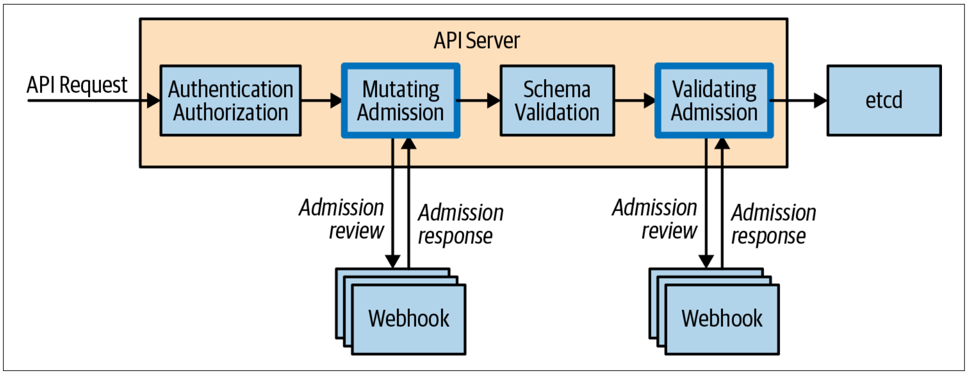

Figure 3-1. API Request flow through the Kuberenetes API

Admission controllers operate inline as an API request flows through the Kubernetes API server and are used to either mutate or validate the API request resource before it's saved to storage. Mutating admission controllers allow the resource to be modified as apposed to validating admission controllers which do not. There are many different types of admission controllers; however, we are going to be focusing on admission webhooks which are dynamically configurable. They allow a cluster administrator by creating either a `MutatingWebhookConfiguration` or `ValidatingWebookConfiguration` resource to configure an endpoint for the API server to send requests for evaluation. The admission webhook will respond with an "admit" or "deny" directive as to whether the API server should save the resource to storage.

Policy and Governance with Gatekeeper

Now that we have defined what policy and governance is let's dive into how to configure policies and ensure that Kubernetes resources are compliant. The Kubernetes project doesn't provide any controllers that enable policy and governance so we will focus on a open source ecosystem project called Gatekeeper. Gatekeeper isn't the only solution for this, but we'll focus on it for the purposes of this chapter.

Gatekeeper is a Kubernetes-native policy controller that evaluates resources based on defined policy and determines whether to admit or deny a Kubernetes resource from being created, or modified. These evaluations happen server-side as the API request flows through the Kubernetes API server which means there is a single point of processing per Kubernetes cluster. Having the policy evaluations processed server-side means that you can install Gatekeeper on existing Kubernetes clusters without the need of changing developer tooling, workflows, or continuous delivery pipelines.

Gatekeeper uses custom resource definitions (CRDs) to define a new set of Kubernetes resources specific to configuring Gatekeeper which allows cluster administrators to use familiar tools like kubectl to operate. In addition, Gatekeeper provides real-team meaningful feedback to the user on why a resource was denied and how to remediate. These Gatekeeper specific custom resources can also be stored in source control and managed using GitOps workflows.

Gatekeeper not only performs resource validation based on policy but it also performs resource mutation (resource modification based on defined conditions) and auditing. Gatekeeper is highly configurable and enables a cluster administrator fined grained control over what resources to evaluate and in which namespaces.

What is Open Policy Agent?

At the core of Gatekeeper is Open Policy Agent, a cloud native open source policy engine that is extensible and allows policy to be portable across different applications. Open Policy Agent is responsible for performing all policy evaluations and returning either an admit or deny. By leveraging Open Policy Agent, Gatekeeper has access to an ecosystem of policy tooling for example conftest which enables you to write policy tests and implement them in continuous integration pipelines prior to deployment.

Open Policy Agent uses a native query language called Rego. This means that all policies must be written in Rego in order to use Open Policy Agent. A dive deep into Rego is outside the scope of this book but it's important to understand that exists at the core of Gatekeeper.

One of the core tenets of Gatekeeper is to abstract the inner workings of Rego from the cluster administrator and present a structured API in the form of a Kubernetes CRD to create and apply policy. By doing this, parameterized policies may be shared across organizations and the community. The Gatekeeper project maintains an policy library solely for this purpose. We will cover the policy library later in this chapter.

Installing Gatekeeper

Before we start configuring policies, we need to install Gatekeeper. Gatekeeper components run as Pods in the gatekeeper-system namespace and configures a webhook admission controller.

Do not install Gatekeeper on a Kubernetes cluster without first understanding how to safely create policy and how to disable it. You should also review the installation YAML prior to installing Gatekeeper to ensure that you are comfortable with the resources the installation YAML creates.

You can install Gatekeeper with a simple one line invocation:

```
$ kubectl apply -f https://raw.githubusercontent.com/open-policy-agent/gate-
keeper/release-3.5/deploy/gatekeeper.yaml
namespace/gatekeeper-system created
resourcequota/gatekeeper-critical-pods created
customresourcedefinition.apiextensions.k8s.io/configs.config.gatekeeper.sh cre-
ated
customresourcedefinition.apiextensions.k8s.io/constraintpodstatuses.status.gate-
keeper.sh created
customresourcedefinition.apiextensions.k8s.io/constrainttemplatepodsta-
tuses.status.gatekeeper.sh created
customresourcedefinition.apiextensions.k8s.io/constrainttem-
plates.templates.gatekeeper.sh created
serviceaccount/gatekeeper-admin created
Warning: policy/v1beta1 PodSecurityPolicy is deprecated in v1.21+, unavailable
in v1.25+
podsecuritypolicy.policy/gatekeeper-admin created
role.rbac.authorization.k8s.io/gatekeeper-manager-role created
clusterrole.rbac.authorization.k8s.io/gatekeeper-manager-role created
rolebinding.rbac.authorization.k8s.io/gatekeeper-manager-rolebinding created
clusterrolebinding.rbac.authorization.k8s.io/gatekeeper-manager-rolebinding cre-
ated
secret/gatekeeper-webhook-server-cert created
service/gatekeeper-webhook-service created
deployment.apps/gatekeeper-audit created
deployment.apps/gatekeeper-controller-manager created
Warning: policy/v1beta1 PodDisruptionBudget is deprecated in v1.21+, unavail-
able in v1.25+; use policy/v1 PodDisruptionBudget
poddisruptionbudget.policy/gatekeeper-controller-manager created
validatingwebhookconfiguration.admissionregistration.k8s.io/gatekeeper-
validating-webhook-configuration created
```

Gatekeeper installation requires cluster-admin permissions and is version specific. Please refer to the official documentation for the latest release of Gatekeeper

Once the installation is complete, confirm that Gatekeeper is up and running:

```
$ kubectl get pods -n gatekeeper-system
NAME                                            READY   STATUS    RESTARTS
AGE
gatekeeper-audit-54c9759898-ljwp8               1/1     Running   0
```

```
1m
gatekeeper-controller-manager-6bcc7f8fb5-4nbkt   1/1     Running   0
1m
gatekeeper-controller-manager-6bcc7f8fb5-d85rn   1/1     Running   0
1m
gatekeeper-controller-manager-6bcc7f8fb5-f8m8j   1/1     Running   0
1m
```

You can also review how the webhook is configured using the command below:

```
$ kubectl get validatingwebhookconfiguration -o yaml
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
  labels:
    gatekeeper.sh/system: "yes"
  name: gatekeeper-validating-webhook-configuration
webhooks:
- admissionReviewVersions:
  - v1
  - v1beta1
  clientConfig:
    service:
      name: gatekeeper-webhook-service
      namespace: gatekeeper-system
      path: /v1/admit
  failurePolicy: Ignore
  matchPolicy: Exact
  name: validation.gatekeeper.sh
  namespaceSelector:
    matchExpressions:
    - key: admission.gatekeeper.sh/ignore
      operator: DoesNotExist
  rules:
  - apiGroups:
    - '*'
    apiVersions:
    - '*'
    operations:
    - CREATE
    - UPDATE
    resources:
    - '*'
  sideEffects: None
  timeoutSeconds: 3
        ....
```

Under the `rules` section of the output above we see that all resources are being sent to the webhook admission controller running as a service named `gatekeeper-webhook-service` in the `gatekeeper-system` namespace. Only resources from namespaces that aren't labelled `admission.gatekeeper.sh/ignore` will be considered for policy evaluation. Finally, the `failurePolicy` is set to `Ignore` which means that this is a fail open configuration. In the case that the Gatekeeper service doesn't respond within the configured timeout of 3 seconds the request will be admitted.

Configuring policies

Now that we have Gatekeeper installed, we can start configuring policies. We will first go through a canoncial example and demonstrate how policies are created by the cluster administrator and then the developer experience when creating compliant and non-compliant resources. We will then expand on each step to gain a deeper understanding. We're going to create a sample policy which states that container images can only come from one specific registry. This example is based on the Gatekeeper policy library.

In order to configure the policy we first need to create a custom resource called a constraint template. Constraint templates are Gatekeeper specific and can be thought of as a policy template. The creation of a constraint template would typically be performed by a cluster administrator. The constraint template in Example 3-1 takes a list of container repositories that are allowed to be used by Kubernetes resources.

Example 3-1. allowedrepos-constraint-template.yaml

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: k8sallowedrepos
  annotations:
    description: Requires container images to begin with a repo string from a speci
fied
      list.
spec:
  crd:
    spec:
      names:
        kind: K8sAllowedRepos
      validation:
        # Schema for the `parameters` field
        openAPIV3Schema:
          properties:
            repos:
              type: array
              items:
                type: string
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8sallowedrepos

        violation[{"msg": msg}] {
          container := input.review.object.spec.containers[_]
          satisfied := [good | repo = input.parameters.repos[_] ; good = starts
with(container.image, repo)]
          not any(satisfied)
          msg := sprintf("container <%v> has an invalid image repo <%v>, allowed
repos are %v", [container.name, container.image, input.parameters.repos])
        }

        violation[{"msg": msg}] {
          container := input.review.object.spec.initContainers[_]
          satisfied := [good | repo = input.parameters.repos[_] ; good = starts
with(container.image, repo)]
          not any(satisfied)
          msg := sprintf("container <%v> has an invalid image repo <%v>, allowed
repos are %v", [container.name, container.image, input.parameters.repos])
        }
```

Create the constraint template using the following command:

```
$ kubectl apply -f allowedrepos-constraint-template.yaml
constrainttemplate.templates.gatekeeper.sh/k8sallowedrepos created
```

Now that we've created the constraint template we must create a constraint resource which instantiates the policy (or puts it into effect). The creation of a constraint would typically be performed by a cluster administrator. The following constraint in Example 3-2 allows all containers with the prefix of `gcr.io/kuar-demo/` in the `default` namespace. The `enforcementAction` is set to deny which means that any resources that are non-compliant to this policy will be denied.

Example 3-2. allowedrepos-constraint.yaml

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sAllowedRepos
metadata:
  name: repo-is-kuar-demo
spec:
  enforcementAction: deny
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
    namespaces:
      - "default"
  parameters:
    repos:
      - "gcr.io/kuar-demo/"
```

```
$ kubectl create -f allowedrepos-constraint.yaml
k8sallowedrepos.constraints.gatekeeper.sh/repo-is-kuar-demo created
```

Now that both the constraint template and the constraint have been created lets create some Pods to test that the policy is indeed working. Example 3-3 creates a Pod using a container image `gcr.io/kuar-demo/kuard-amd64:blue` which is compliant to the constraint we defined in the previous step and will be created without issue. Workload resource creation would typically be performed by the developer responsible for operating the service or a continuous delivery pipeline.

Example 3-3. compliant-pod.yaml.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kuard
spec:
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:blue
      name: kuard
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

```
    $ kubectl apply -f compliant-pod.yaml
    pod/kuard created
```

Let's try creating a Pod that is non-compliant to the constraint we defined. Example 3-4 creates a Pod using a container image `nginx` which is NOT compliant to the constraint we defined in the previous step and will be created. Workload resource creation would typically be performed by the developer or continuous delivery pipeline responsible for operating the service. Note the output below:

Example 3-4. noncompliant-pod.yaml.yaml

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

```
    $ kubectl apply -f noncompliant-pod.yaml
    Error from server ([repo-is-kuar-demo] container <nginx> has an invalid image
    repo <nginx>, allowed repos are ["gcr.io/kuar-demo/"]): error when creating
    "noncompliant-pod.yaml": admission webhook "validation.gatekeeper.sh" denied
    the request: [repo-is-kuar-demo] container <nginx> has an invalid image repo
    <nginx>, allowed repos are ["gcr.io/kuar-demo/"]
```

We see an error is returned the user with the details on why the resource was not created and how to remediate. This message is configured by the cluster administrator via the constraint template.

When a constraint is defined with a scope of Pods and the user creates a resource that generates Pod resources such as `ReplicaSets` an error will not be returned to the user but rather the controller trying to create the Pod. In order to see these error message you must look in the event log for the resource responsible for generating the Pods.

Understanding Constraint Templates

Now that we have walked through a canoncial example let's take a closer look at the constraint template we used in Example 3-5 which takes a list of container repositories that are allowed in Kubernetes resources.

Example 3-5. allowedrepos-constraint-template.yaml

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: k8sallowedrepos
  annotations:
    description: Requires container images to begin with a repo string from a speci
fied
      list.
spec:
  crd:
    spec:
      names:
        kind: K8sAllowedRepos
      validation:
        # Schema for the `parameters` field
        openAPIV3Schema:
          properties:
            repos:
              type: array
              items:
                type: string
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8sallowedrepos

        violation[{"msg": msg}] {
          container := input.review.object.spec.containers[_]
          satisfied := [good | repo = input.parameters.repos[_] ; good = starts
with(container.image, repo)]
          not any(satisfied)
          msg := sprintf("container <%v> has an invalid image repo <%v>, allowed
repos are %v", [container.name, container.image, input.parameters.repos])
        }

        violation[{"msg": msg}] {
          container := input.review.object.spec.initContainers[_]
          satisfied := [good | repo = input.parameters.repos[_] ; good = starts
with(container.image, repo)]
          not any(satisfied)
          msg := sprintf("container <%v> has an invalid image repo <%v>, allowed
repos are %v", [container.name, container.image, input.parameters.repos])
        }
```

This constraint template has an `apiVersion` and `kind` that are part of the custom resources only used by Gatekeeper. Under the `spec` section a name of `K8sAllowedRepos` (keep note of this name as when creating constraints we use this name as the constraint `kind`) along with a schema that defines array of strings for the cluster administrator to configure by providing a list of container registries that are allowed. It also contains the raw Rego policy definition under the `target` section. This policy evaluates containers and initContainers to ensure that the container repo name starts with the values provided by the constraint. The `msg` section defines the message that is sent back to the user if the policy is violated.

Creating Constraints

In order to instantiate a policy you must create a constraint that provides the required parameters to the constraint template. There may be many constraints that match the kind of a specific constraint template. Let's take a closer look at the constraint we used in Example 3-6 which only allows container images that originate from `gcr.io/kuar-demo/`

Example 3-6. allowedrepos-constraint.yaml

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sAllowedRepos
metadata:
  name: repo-is-kuar-demo
spec:
  enforcementAction: deny
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
    namespaces:
      - "default"
  parameters:
    repos:
      - "gcr.io/kuar-demo/"
```

You may notice that the constraint is of `kind` "K8sAllowedRepos" which was defined as part of the constraint template. There is also an `enforcementAction` of "deny" defined. This means that resources that aren't compliant to this constraint will be denied. `enforcementAction` also accepts "dryrun" and "warn" as values. The `enforcementAction` of "dryrun" is used to test policies and verify the impact using the audit featured in Gatekeeper whereas "warn" sends a warning back to the user with the associated message but allows them to create or update. The `match` portion defines the scope that this constraint should operate on. In this case, all Pods in the default namespace. Finally, the parameters section is required to satisfy the constraint template (an array of strings). The following demonstrates the user experience when the `enforcementAction` is set to "warn":

```
$ kubectl apply -f noncompliant-pod.yaml
Warning: [repo-is-kuar-demo] container <nginx> has an invalid image repo
<nginx>, allowed repos are ["gcr.io/kuar-demo/"]
pod/nginx-noncompliant created
```

Constraints are only enforced on resource CREATE and UPDATE events. If you already have workloads running on a cluster, they will not be re-evaluated by Gatekeeper UNTIL a CREATE or UPDATE event takes place. Here is a real-world example to demonstrate. You create a policy that only allows containers from a specific registry. All workloads that are already running on the cluster will continue to do so. In the event that you scale the workload Deployment from 1 to 2, the ReplicaSet will attempt to create another Pod. If that Pod doesn't have a container from an allowed repository, then it will be denied. It's important to set the `enforcementAction` to "dryrun" and audit to confirm that any policy violations are known before setting the `enforcementAction` to "deny. .

Audit

Being able to enforce policy upon new resource creation is only one piece of the policy and governance story. Policies often change over time, so you can also use Gatekeeper to confirm that everything currently deployed is still compliant. Additionally, you may already have a cluster full of services and wish to install Gatekeeper to bring these resources into compliance. Gatekeeper ships with audit capabilities which allow a cluster administrator to get a list of resources that currently exist on a cluster that are not in compliance with the policy.

In order to demonstrate how audit works lets first update the `repo-is-kuar-demo` constraint to have an `enforcementAction` action of "dryrun". This will allow a user to create non-compliant resources. We will then determine which resources are not compliant to the constraint using audit. The constraint below Example 3-7 has the `enforcementAction` action set to “dryrun”:

Example 3-7. allowedrepos-constraint-dryrun.yaml

```yaml
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sAllowedRepos
metadata:
  name: repo-is-kuar-demo
spec:
  enforcementAction: dryrun
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
    namespaces:
      - "default"
  parameters:
    repos:
      - "gcr.io/kuar-demo/"
```

Update the constraint by running the following command:

```
$ kubectl apply -f allowedrepos-constraint-dryrun.yaml
k8sallowedrepos.constraints.gatekeeper.sh/repo-is-kuar-demo configured
```

Create a non-compliant Pod using the following command:

```
$ kubectl apply -f noncompliant-pod.yaml
pod/nginx-noncompliant created
```

To audit the list of resources that are non-compliant for a given constraint you can run a `kubectl get constraint` on the specific constraint specifying to output in YAML format as follows:

```
$ kubectl get constraint repo-is-kuar-demo -o yaml
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sAllowedRepos
....
spec:
  enforcementAction: dryrun
  match:
    kinds:
    - apiGroups:
      - ""
      kinds:
      - Pod
    namespaces:
    - default
  parameters:
    repos:
      - gcr.io/kuar-demo/
status:
  auditTimestamp: "2021-07-14T20:05:38Z"
        ....
  totalViolations: 1
  violations:
  - enforcementAction: dryrun
    kind: Pod
    message: container <nginx> has an invalid image repo <nginx>, allowed repos
are
      ["gcr.io/kuar-demo/"]
    name: nginx-noncompliant
    namespace: default
```

Under the `status` section we can see the `auditTimestamp` which is the last time the audit was run. The `totalViolations" lists the number of resources that vio late this constraint. The +violations` section contains a list of violations. We can see that the nginx-noncompliant Pod is in violation and the message with the details why. We can also see that this policy has an `enforcementAction` of "dryrun".

Using a constraint `enforcementAction` of "dryrun" along with audit is a powerful way to confirm that your policy is having the desired impact. It also creates a workflow to bring resources into compliance.

Mutation

So far we have covered how you can use constraints to validate if a resource is compliant. What about using Gatekeeper to modify resources on behalf of the user to make them compliant? This is handled via the mutation feature in Gatekeeper. Earlier in this chapter we discussed two different type of admission webhooks, mutating and validating. By default, Gatekeeper is only deployed as a validating admission webhook by may be configured to operate as a mutating admission webhook.

Mutation features in Gatekeeper are still in alpha state and are likely to change. We share them to demonstrate the upcoming capabilities of Gatekeeper and how they may be used to meet your policy and compliance needs. The installation steps in this chapter do not cover enabling mutation. Please refer to the Gatekeeper project for more information on enabling mutation.

Let's walk through an example to demonstrate the power of mutation. In this example we will set the `imagePullPolicy` to "Always" on all Pods. We will assume that Gatekeeper is configured correctly to support mutation. The following example in

Example 3-8 defines a mutation assignment that matches all Pods except in the "system" namespace and assigns the value of "Always" to `imagePullPolicy`:

Example 3-8. imagepullpolicyalways-mutation.yaml

```
apiVersion: mutations.gatekeeper.sh/v1alpha1
kind: Assign
metadata:
  name: demo-image-pull-policy
spec:
  applyTo:
  - groups: [""]
    kinds: ["Pod"]
    versions: ["v1"]
  match:
    scope: Namespaced
    kinds:
    - apiGroups: ["*"]
      kinds: ["Pod"]
    excludedNamespaces: ["system"]
  location: "spec.containers[name:*].imagePullPolicy"
  parameters:
    assign:
      value: Always
```

Create the mutation assignment:

```
$ kubectl apply -f imagepullpolicyalways-mutation.yaml
assign.mutations.gatekeeper.sh/demo-image-pull-policy created
```

Now create a Pod. This Pod doesn't have `imagePullPolicy` set. By default this field is set to "IfNotPresent". However in this case we expect Gatekeeper to mutate this field to "Always".

```
$ kubectl apply -f compliant-pod.yaml
pod/kuard created
```

Validate that the `imagePullPolicy` has been successfully mutated to "Always" by running the following:

```
$ $ kubectl get pods kuard -o=jsonpath="{.spec.containers[0].imagePullPolicy}"

Always
```

Mutation admission happens prior to validation admission so you should create constraints that validate the mutations you expect to be applied to the specific resource.

Delete the Pod using the following command:

```
$ kubectl delete -f compliant-pod.yaml
pod/kuard deleted
```

Delete the mutation assignment using the following command:

```
$ kubectl delete -f imagepullpolicyalways-mutation.yaml
assign.mutations.gatekeeper.sh/demo-image-pull-policy deleted
```

In contrast with validation, mutation provides a way to auto-remediate resources that aren't complaint on behalf of the cluster administrator. In this section we covered how to configure mutation assignment which allows you to modify any field in a resource.

Data Replication

When writing constraints you may want to compare the value of one field to the value of a field in another resource. A specific example of when you might need to do this is making sure that ingress hostnames are unique across a cluster. By default, Gatekeeper can only evaluate fields within the current resource and must be configured if comparisons across resources are required to fulfill a policy. Gatekeeper can be configured to cache specific resources into Open Policy Agent so that comparisons across resources can be made. The following config resource in Example 3-9 configures Gatekeeper to cache `Namespace` and `Pod` resources.

Example 3-9. config-sync.yaml

```yaml
apiVersion: config.gatekeeper.sh/v1alpha1
kind: Config
metadata:
  name: config
  namespace: "gatekeeper-system"
spec:
  sync:
    syncOnly:
      - group: ""
        version: "v1"
        kind: "Namespace"
      - group: ""
        version: "v1"
        kind: "Pod"
```

You should only cache the specific resources needed to perform a policy evaluation. Having hundreds or thousands of resources cached in OPA will require more memory and may also have security implications.

The following constraint template in Example 3-10 demonstrates how comparisons across resources may be made (in this case, unique ingress hostnames) in the Rego section. Specifically the "data.inventory" is referring to the cache resources as opposed to "input" which is the resource sent for evaluation from the Kubernetes API server as part of the admission flow. This example is based on the Gatekeeper policy library

Example 3-10. uniqueingresshost-constraint-template.yaml

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: k8suniqueingresshost
  annotations:
    description: Requires all Ingress hosts to be unique.
spec:
  crd:
    spec:
      names:
        kind: K8sUniqueIngressHost
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8suniqueingresshost

        identical(obj, review) {
          obj.metadata.namespace == review.object.metadata.namespace
          obj.metadata.name == review.object.metadata.name
        }

        violation[{"msg": msg}] {
          input.review.kind.kind == "Ingress"
          re_match("^(extensions|networking.k8s.io)$", input.review.kind.group)
          host := input.review.object.spec.rules[_].host
          other := data.inventory.namespace[ns][otherapiversion]["Ingress"][name]
          re_match("^(extensions|networking.k8s.io)/.+$", otherapiversion)
          other.spec.rules[_].host == host
          not identical(other, input.review)
          msg := sprintf("ingress host conflicts with an existing ingress <%v>",
[host])
        }
```

Data replication is a powerful tool that allows you to make comparisons across Kubernetes resources. We recommend only configuring if you have policies that require it to function. In addition, scope it only to the resources needed.

Metrics

Gatekeeper emits metrics in Prometheus format to enable continuous monitoring of resource compliance. Simple metrics regarding the overall health of Gatekeeper are available for example:

- Number of constraints
- Number of constraint templates
- Number of requests being sent to Gatekeeper

In addition, details on policy compliance and governance are also available:

- The total number of audit violations
- Number of constraints by enforcementAction
- Audit duration

Having the policy and governance process completely automated is the ideal goal state and as such it's strongly recommended that you monitor Gatekeeper from an external monitoring system and setup alerts based on resource compliance.

Policy Library

One of the core tenets of the Gatekeeper project is to create reuseable policy libraries that may be shared between organizations. Having the ability to share policies reduces the boilerplate policy work and allows cluster administrators to focus on applying policy rather than writing it. The Gatekeeper project has a great policy library which contains both a general library with the most common policies in addition to a pod-security-policy library which models the capabilities of the `PodSecurityPolicy` API as Gatekeeper policy. The great thing about this library is that it is ever expanding and is open source so feel free to contribute any policies that you write.

Summary

In this chapter we discussed policy and governance and why it is important as more and more resources are deployed to Kubernetes. We covered the Gatekeeper project, a Kubernetes-native policy controller built on Open Policy Agent and how it may be used to successfully meet your policy and governance requirements. From writing policies to auditing which resources are in compliance with policy you are now equipped with the know-how to meet your compliance needs.

About the Authors

Brendan Burns is the cofounder of the Kubernetes open source project. He is also a Distinguished Engineer at Microsoft on the Azure team.

Joe Beda is the lead engineer for the Google Compute Engine project. He has been at Google for ~8 years and, besides GCE, Joe has worked on Google Talk, Goog-411 and Adwords keyword suggestions. Before Google, Joe was an engineer at Microsoft working on IE and WPF.

Kelsey Hightower has worn every hat possible throughout his career in tech and enjoys leadership roles focused on making things happen and shipping software. Kelsey is a strong open source advocate focused on building simple tools that make people smile. When he isn't slinging Go code, you can catch him giving technical workshops covering everything from programming to system administration.

Lachlan Evenson is a principal program manager on the container compute team at Microsoft Azure. He's helped numerous people onboard to Kubernetes through both hands-on teaching and conference talks.