Kubernetes – Core Concepts and Constructs

This chapter will cover the core **Kubernetes** constructs, such as **pods**, **services**, **replication controllers**, and **labels**. A few simple application examples will be included to demonstrate each construct. The chapter will also cover basic operations for your cluster. Finally, **health checks** and **scheduling** will be introduced with a few examples.

This chapter will discuss the following topics:

- Kubernetes' overall architecture
- Introduction to core Kubernetes constructs, such as pods, services, replication controllers, and labels
- Understand how labels can ease management of a Kubernetes cluster
- Understand how to monitor services and container health
- Understand how to set up scheduling constraints based on available cluster resources

The architecture

Although **Docker** brings a helpful layer of abstraction and tooling around container management, Kubernetes brings similar assistance to orchestrating containers at scale as well as managing full application stacks.

K8s moves up the stack giving us constructs to deal with management at the application or service level. This gives us automation and tooling to ensure high availability, application stack, and service-wide portability. K8s also allows finer control of resource usage, such as CPU, memory, and disk space across our infrastructure.

Kubernetes provides this higher level of orchestration management by giving us key constructs to combine multiple containers, endpoints, and data into full application stacks and services. K8s then provides the tooling to manage the *when*, *where*, and *how many* of the stack and its components.

Figure 2.1. Kubernetes core architecture

In the preceding figure (Figure 2.1), we see the core architecture for Kubernetes. Most administrative interactions are done via the kubectl script and/or RESTful service calls to

the API.

Note the ideas of the *desired state* and *actual state* carefully. This is key to how Kubernetes manages the cluster and its workloads. All the pieces of K8s are constantly working to monitor the current *actual state* and synchronize it with the *desired state* defined by the administrators via the API server or kubectl script. There will be times when these states do not match up, but the system is always working to reconcile the two.

Master

Essentially, **master** is the brain of our cluster. Here, we have the core API server, which maintains RESTful web services for querying and defining our desired cluster and workload state. It's important to note that the control pane only accesses the master to initiate changes and not the nodes directly.

Additionally, the master includes the **scheduler**, which works with the API server to schedule workloads in the form of pods on the actual minion nodes. These pods include the various containers that make up our application stacks. By default, the basic Kubernetes scheduler spreads pods across the clusters and uses different nodes for matching pod replicas. Kubernetes also allows specifying necessary resources for each container, so scheduling can be altered by these additional factors.

The replication controller works with the API server to ensure that the correct number of pod replicas are running at any given time. This is exemplary of the *desired state* concept. If our replication controller is defining three replicas and our *actual state* is two copies of the pod running, then the scheduler will be invoked to add a third pod somewhere on our cluster. The same is true if there are too many pods running in the cluster at any given time. In this way, K8s is always pushing towards that *desired state*.

Finally, we have **etcd** running as a distributed configuration store. The Kubernetes state is stored here and etcd allows values to be watched for changes. Think of this as the brain's shared memory.

Node (formerly minions)

In each node, we have a couple of components. The **kublet** interacts with the API server to update state and to start new workloads that have been invoked by the scheduler.

Kube-proxy provides basic load balancing and directs traffic destined for specific services to the proper pod on the backend. See the *Services* section later in this chapter.

Finally, we have some default pods, which run various infrastructure services for the node. As we explored briefly in the previous chapter, the pods include services for **Domain Name System** (**DNS**), logging, and pod health checks. The default pod will run alongside our scheduled pods on every node.

> Note that in v1.0, **minion** was renamed to **node**, but there are still remnants of the term minion in some of the machine naming scripts and documentation that exists on the Web. For clarity, I've added the term minion in addition to node in a few places throughout the book.

Core constructs

Now, let's dive a little deeper and explore some of the core abstractions Kubernetes provides. These abstractions will make it easier to think about our applications and ease the burden of life cycle management, high availability, and scheduling.

Pods

Pods allow you to keep related containers close in terms of the network and hardware infrastructure. Data can live near the application, so processing can be done without incurring a high latency from network traversal. Similarly, common data can be stored on volumes that are shared between a number of containers. Pods essentially allow you to logically group containers and pieces of our application stacks together.

While pods may run one or more containers inside, the pod itself may be one of many that is running on a Kubernetes (minion) node. As we'll see, pods give us a logical group of containers that we can then replicate, schedule, and balance service endpoints across.

Pod example

Let's take a quick look at a pod in action. We will spin up a **Node.js** application on the cluster. You'll need a GCE cluster running for this, so see *Chapter 1, Our First Cluster*, if you don't already have one started.

Now, let's make a directory for our definitions. In this example, I will create a folder in the /book-examples subfolde under our home directory.

```
$ mkdir book-examples
$ cd book-examples
$ mkdir 02_example
$ cd 02_example
```

> **Downloading the example code** You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/ support and register to have the files e-mailed directly to you.

Use your favorite editor to create the following file:

```
    apiVersion: v1 kind:
    Pod metadata:
name: node-js-pod spec:
        containers: - name: node-js-
        pod
          image: bitnami/apache:latest ports: -
          containerPort: 80
```

Listing 2-1: nodejs-pod.yaml

This file creates a pod name `node-js-pod` with the latest `bitnami/apache` container running on port `80`. We can check this using the following command:

```
$ kubectl create -f nodejs-pod.yaml
```

The output is as follows:

```
pods/node-js-pod
```

This gives us a pod running the specified container. We can see more information on the pod by running the following command:

```
$ kubectl describe pods/node-js-pod
```

You'll see a good deal of information, such as the pod's status, IP address, and even relevant log events. You'll note the pod IP address is a private `10.x.x.x` address, so we cannot access it directly from our local machine. Not to worry as the `kubectl exec` command mirrors Docker's `exec` functionality. Using this feature, we can run a command inside a pod:

```
$ kubectl exec node-js-pod -- curl <private ip address>
```

> By default, this runs a command in the first container it finds, but you can select a specific one using the -c argument.

After running, the command you should see some HTML code. We'll have a prettier view later in the chapter, but for now, we can see that our pod is indeed running as expected.

Labels

Labels give us another level of categorization, which becomes very helpful in terms of everyday operations and management. Similar to tags, labels can be used as the basis of service discovery as well as a useful grouping tool for day-to-day operations and management tasks.

Labels are just simple key-value pairs. You will see them on pods, replication controllers, services, and so on. The label acts as a selector and tells Kubernetes which resources to work with for a variety of operations. Think of it as a *filtering* option.

We will take a look at labels more in depth later in this chapter, but first, we will explore the remaining two constructs, services, and replication controllers.

The container's afterlife

As anyone in operations can attest, failures happen all the time. Containers and pods can and will crash, become corrupted, or maybe even just get accidentally shut off by a clumsy admin poking around on one of the nodes. Strong policy and security practices like enforcing least privilege curtail some of these incidents, but "involuntary workload slaughter happens" and is simply a fact of operations.

Luckily, Kubernetes provides two very valuable constructs to keep this somber affair all tidied up behind the curtains. Services and replication controllers give us the ability to keep our applications running with little interruption and graceful recovery.

Services

Services allow us to abstract access away from the consumers of our applications. Using a reliable endpoint, users and other programs can access pods running on your cluster seamlessly.

K8s achieves this by making sure that every node in the cluster runs a proxy named kube-proxy. As the name suggests, kube-proxy's job is to proxy communication from a service endpoint back to the corresponding pod that is running the actual application.

Figure 2.2. The kube-proxy architecture

Membership in the service load balancing pool is determined by the use of selectors and labels. Pods with matching labels are added to the list of candidates where the service forwards traffic. A virtual IP address and port are used as the entry point for the service, and traffic is then forwarded to a random pod on a target port defined by either K8s or your definition file.

Updates to service definitions are monitored and coordinated from the K8s cluster master and propagated to the **kube-proxy daemons** running on each node.

> At the moment, kube-proxy is running on the node host itself. There are plans to containerize this and the kubelet by default in the future.

Replication controllers

Replication controllers (**RCs**), as the name suggests, manage the number of nodes that a pod and included container images run on. They ensure that an instance of an image is being run with the specific number of copies.

As you start to operationalize your containers and pods, you'll need a way to roll out updates, scale the number of copies running (both up and down), or simply ensure that at least one instance of your stack is always running. RCs creates a high- level mechanism to make sure that things are operating correctly across the entire application and cluster.

RCs are simply charged with ensuring that you have the desired scale for your application. You define the number of pod replicas you want running and give it a template for how to create new pods. Just like services, we will use selectors and labels to define a pod's membership in a replication controller.

> Kubernetes doesn't require the strict behavior of the replication controller. In fact, version 1.1 has a **job controller** in beta that can be used for short lived workloads which allow jobs to be run to a completion state.

Our first Kubernetes application

Before we move on, let's take a look at these three concepts in action. Kubernetes ships with a number of examples installed, but we will create a new example from scratch to illustrate some of the concepts.

We've already created a pod definition file, but as we learned, there are many advantages to running our pods via replication controllers. Again, using the book-examples/02_example folder we made earlier, we will create some definition files and start a cluster of Node.js servers using a replication controller approach. Additionally, we'll add a public face to it with a load-balanced service.

Use your favorite editor to create the following file:

```
apiVersion: v1 kind:
ReplicationController metadata:
  name: node-js
  labels:
name: node-js deployment: demo
spec:
  replicas: 3
  selector:
  name: node-js
  deployment: demo
  template:
    metadata:
    labels:
name: node-js spec:
      containers: - name:
      node-js
        image: jonbaier/node-express-info:latest ports: -
        containerPort: 80
```

Listing 2-2: nodejs-controller.yaml

This is the first resource definition file for our cluster, so let's take a closer look. You'll note that it has four first-level elements (kind, apiVersion, metadata, and spec). These are common among all top-level Kubernetes resource definitions:

- Kind tells K8s what type of resource we are creating. In this case, the type is ReplicationController. The kubectl script uses a single create command for all types of resources. The benefit here is that you can easily create a number of resources of various types without needing to specify individual parameters for each type. However, it requires that the definition files can identify what it is they are specifying.
- ApiVersion simply tells Kubernetes which version of the schema we are using. All examples in this book will be on v1.
- Metadata is where we will give the resource a name and also specify labels that will be used to search and select resources for a given operation. The metadata element also allows you to create annotations, which are for nonidentifying information that might be useful for client tools and libraries.

• Finally, we have spec, which will vary based on the kind or type of resource we are creating. In this case, it's ReplicationController, which ensures the desired number of pods are running. The replicas element defines the desired number of pods, the selector tells the controller which pods to watch, and finally, the template element defines a template to launch a new pod. The template section contains the same pieces we saw in our pod definition earlier. An important thing to note is that the selector values need to match the labels values specified in the pod template. Remember that this matching is used to select the pods being managed.

Now, let's take a look at the service definition:

```
apiVersion: v1 kind:
Service metadata:
  name: node-js
  labels:
name: node-js spec:
  type: LoadBalancer ports:
  - port: 80 selector:
    name: node-js
```

Listing 2-3: nodejs-rc-service.yaml

The YAML here is similar to the ReplicationController. The main difference is seen in the service spec element. Here, we define the Service type, listening port, and selector, which tells the Service proxy which pods can answer the service.

> Kubernetes supports both YAML and JSON formats for definition files.

Create the Node.js express replication controller:

$ kubectl create -f nodejs-controller.yaml

The output is as follows:

replicationcontrollers/node-js

This gives us a replication controller that ensures that three copies of the container are

always running:

$ kubectl create -f nodejs-rc-service.yaml

The output is as follows:

services/node-js

On GCE, this will create an external load balancer and forwarding rules, but you may need to add additional firewall rules. In my case, the firewall was already open for port 80. However, you may need to open this port, especially if you deploy a service with ports other than 80 and 443.

OK, now we have a running service, which means that we can access the Node.js servers from a reliable URL. Let's take a look at our running services:

$ kubectl get services

The following screenshot is the result of the preceding command:

Figure 2.3. Services listing

In the preceding figure (Figure 2.3), you should note that the **node - js** service running and, in the **IP(S)** column, you should have both a private and a public (**130.211.186.84** in the screenshot) IP address. Let's see if we can connect by opening up the public address in a browser:

Figure 2.4. Container info application

You should see something like Figure 2.4. If we visit multiple times, you should note that the container name changes. Essentially, the service load balancer is rotating between available pods on the backend.

> Browsers usually cache web pages, so to really see the container name change you may need to clear your cache or use a proxy like this one: https://hide.me/en/proxy

Let's try playing chaos monkey a bit and kill off a few containers to see what Kubernetes does. In order to do this, we need to see where the pods are actually running. First, let's list our pods:

$ kubectl get pods

The following screenshot is the result of the preceding command:

Figure 2.5. Currently running pods

Now, let's get some more details on one of the pods running a node-js container. You can do this with the describe command with one of the pod names listed in the last command:

$ kubectl describe pod/node-js-sjc03

The following screenshot is the result of the preceding command:

Figure 2.6. Pod description

You should see the preceding output. The information we need is the **Node:** section. Let's use the node name **SSH** (short for **Secure Shell**) into the (minion) node running this workload:

$ gcloud compute --project "<Your project ID>" ssh --zone "<your gce zone>" "<Node from pod describe>"

Once SSHed into the node, if we run a sudo docker ps command, we should see at least two containers: one running the pause image and one running the actual node- express-info image. You may see more if the K8s scheduled more than one replica on this node. Let's grab the container ID of the jonbaier/node-express-info image (not gcr.io/google_containers/pause) and kill it off to see what happens. Save this container ID somewhere for later:

$ sudo docker ps --filter="name=node-js"

$ sudo docker stop <node-express container id>

$ sudo docker rm <container id>

$ sudo docker ps --filter="name=node-js"

Unless you are really quick you'll probably note that there is still a node-express- info container running, but look closely and you'll note that the container id is different and the creation time stamp shows only a few seconds ago. If you go back to the service URL, it is functioning like normal. Go ahead and exit the SSH session for now.

Here, we are already seeing Kubernetes playing the role of on-call operations ensuring that our application is always running.

Let's see if we can find any evidence of the outage. Go to the **Events** page in the Kubernetes UI. You can find it on the main K8s dashboard under **Events** in the **Views** menu. Alternatively, you can just use the following URL, adding your master ip:

https://<your master ip>/api/v1/proxy/namespaces/kube-system/ services/kube-ui/#/dashboard/events

You will see a screen similar to the following screenshot:

Figure 2.7. Kubernetes UI event page

You should see three recent events. First, Kubernetes pulls the image. Second, it creates a new container with the pulled image. Finally, it starts that container again. You'll note that,

from the time stamps, this all happens in less than a second. Time taken may vary based on cluster size and image pulls, but the recovery is very quick.

More on labels

As mentioned previously, labels are just simple key-value pairs. They are available on pods, replication controllers, services, and more. If you recall our service YAML, in *Listing 2-3*: nodejs-rc-service.yaml, there was a selector attribute. The selector tells Kubernetes which labels to use in finding pods to forward traffic for that service.

K8s allows users to work with labels directly on replication controllers and services. Let's modify our replicas and services to include a few more labels. Once again, use your favorite editor and create these two files as follows:

```
apiVersion: v1 kind:
ReplicationController metadata:
  name: node-js-labels labels:
name: node-js-labels app: node-js-
express deployment: test spec:
  replicas: 3
  selector:
  name: node-js-labels app: node-
  js-express deployment: test
  template:
    metadata:
    labels:
    name: node-js-labels app: node-js-
    express deployment: test spec:
      containers: - name: node-js-
      labels
        image: jonbaier/node-express-info:latest ports: -
        containerPort: 80
```

Listing 2-4: nodejs-labels-controller.yaml

```
apiVersion: v1 kind:
Service metadata:
  name: node-js-labels labels:
name: node-js-labels app: node-js-
express deployment: test spec:
  type: LoadBalancer ports:
  - port: 80
```

selector:
name: node-js-labels app: node-js-express deployment: test

Listing 2-5: nodejs-labels-service.yaml

Create the replication controller and service as follows:

$ kubectl create -f nodejs-labels-controller.yaml

$ kubectl create -f nodejs-labels-service.yaml

Let's take a look at how we can use labels in everyday management. The following table shows us the options to select labels:

Operators Description Example = or == You can use either style to select keys with values equal to the string on the right

name = apache

!= Select keys with values that do not equal the string on the right

Environment != test

In Select resources whose labels have keys with values in this set

tier in (web, app)

Notin Select resources whose labels have keys

with values not in this set

tier not in (lb, app)

<Key name> Use a key name only to select resources

whose labels contain this key

tier

Table 1: Label selectors

Let's try looking for replicas with test deployments:

$ kubectl get rc-l deployment=test

The following screenshot is the result of the preceding command:

Figure 2.8. Replication controller listing

You'll notice that it only returns the replication controller we just started. How about services with a label named component? Use the following command:

$ kubectl get services -l component

The following screenshot is the result of the preceding command:

Figure 2.9. Listing of services with a label named "component"

Here, we see the core Kubernetes service only. Finally, let's just get the node-js servers we started in this chapter. See the following command:

$ kubectl get services -l "name in (node-js,node-js-labels)"

The following screenshot is the result of the preceding command:

Figure 2.10. Listing of services with a label name and a value of "node-js" or "nodejs-labels"

Additionally, we can perform management tasks across a number of pods and services. For example, we can kill all replication controllers that are part of the demo deployment (if we had any running) as follows:

$ kubectl delete rc -l deployment=demo

Otherwise, kill all services that are not part of a production or test deployment (again, if we had any running), as follows:

$ kubectl delete service -l "deployment notin (test, production)"

It's important to note that while label selection is quite helpful in day-to-day management tasks it does require proper deployment hygiene on our part. We need to make sure that we have a tagging standard and that it is actively followed in the resource definition files for everything we run on Kubernetes.

While we used service definition YAML files to create our services thus far, you can actually create them using a kubectl command only. To try this out, first run the get pods command and get one of the node-js pod names. Next, use the following expose command to create a service endpoint for just that pod: **$ kubectl expose pods/node-js-gxkix --port=80 --name=testing-vip --create-external-load-balancer=true** This will create a service named testing-vip and also a public vip (load balancer IP) that can be used to access this pod over port 80. There's a number of other optional parameters that can be used. These can be found with the following: **kubectl expose --help**

Health checks

Kubernetes provides two layers of health checking. First, in the form of HTTP or TCP checks, K8s can attempt to connect to a particular endpoint and give a status of healthy on a successful connection. Second, application-specific health checks can be performed using command line scripts.

Let's take a look at a few health checks in action. First, we'll create a new controller with a health check:

```
apiVersion: v1 kind:
ReplicationController metadata:
```

```
        name: node-js
        labels:
    name: node-js spec:
        replicas: 3
        selector:
name: node-js template:
            metadata:
            labels:
    name: node-js spec:
                containers: - name:
                node-js
                    image: jonbaier/node-express-info:latest ports:
```



```
                - containerPort: 80
                livenessProbe:
                    # An HTTP health check
                    httpGet:
                    path: /status/ port: 80
                    initialDelaySeconds: 30
                    timeoutSeconds: 1
```

Listing 2-6: nodejs-health-controller.yaml

Note the addition of the livenessprobe element. This is our core health check element. From there, we can specify httpGet, tcpScoket, or exec. In this example, we use httpGet to perform a simple check for a URI on our container. The probe will check the path and port specified and restart the pod if it doesn't successfully return.

> Status codes between 200 and 399 are all considered healthy by the probe.

Finally, initialDelaySeconds gives us the flexibility to delay health checks until the pod has finished initializing. timeoutSeconds is simply the timeout value for the probe.

Let's use our new health check-enabled controller to replace the old node-js RC. We can do this using the replace command, which will replace the replication controller definition:

```
$ kubectl replace -f nodejs-health-controller.yaml
```

Replacing the RC on it's own won't replace our containers because it still has three

healthy pods from our first run. Let's kill off those pods and let the updated ReplicationController replace them with containers that have health checks.

$ kubectl delete pods -l name=node-js

Now, after waiting a minute or two, we can list the pods in an RC and grab one of the pod IDs to inspect a bit deeper with the describe command:

$ kubectl describe rc/node-js

The following screenshot is the result of the preceding command:

Figure 2.11. Description of "node-js" replication controller

Then, using the following command for one of the pods:

$ kubectl describe pods/node-js-1m3cs

The following screenshot is the result of the preceding command:

Figure 2.12. Description of "node-js-1m3cs" pod

Depending on your timing, you will likely have a number of events for the pod. Within a minute or two, you'll note a pattern of *killing*, *started*, and *created* events repeating over and over again. You should also see an unhealthy event described as **Liveness probe failed: Cannot GET /status/**. This is our health check failing because we don't have a page responding at /status.

You may note that if you open a browser to the service load balancer address, it still responds with a page. You can find the load balancer IP with a kubectl get services command.

This is happening for a number of reasons. First, the health check is simply failing because /status doesn't exist, but the page where the service is pointed is still functioning normally. Second, the livenessProbe is only charged with restarting the container on a health check fail. There is a separate readinessProbe that will remove a container from the pool of pods answering service endpoints.

Let's modify the health check for a page that does exist in our container, so we have a proper health check. We'll also add a readiness check and point it to the nonexistent status page. Open the nodejs-health-controller.yaml file and modify the spec section to match *Listing 2-7* and save it as nodejs-health-controller-2.yaml.

apiVersion: v1 kind:

```
ReplicationController metadata:
  name: node-js
  labels:
name: node-js spec:
  replicas: 3
  selector:
name: node-js template:
    metadata:
    labels:
name: node-js spec:
      containers: - name:
      node-js
        image: jonbaier/node-express-info:latest ports: -
        containerPort: 80 livenessProbe:
          # An HTTP health check
          httpGet:
        path: /status/ port: 80
        initialDelaySeconds: 30
        timeoutSeconds: 1 readinessProbe:
          # An HTTP health check
          httpGet:
          path: /status/ port: 80
          initialDelaySeconds: 30
          timeoutSeconds: 1
```


Listing 2-7: nodejs-health-controller-2.yaml

This time, we will delete the old RC, which will kill the pods with it, and create a new RC with our updated YAML file.

```
$ kubectl delete rc -l name=node-js
$ kubectl create -f nodejs-health-controller-2.yaml
```

Now when we describe one of the pods, we only see the creation of the pod and the container. However, you'll note that the service load balancer IP no longer works. If we run the describe command on one of the new nodes we'll note a **Readiness probe failed** error message, but the pod itself continues running. If we change the readiness probe path to path: /, we will again be able to fulfill requests from the main service. Open up nodejs-health-controller-2.yaml in an editor and make that update now. Then, once again remove and recreate the replication controller:

```
$ kubectl delete rc -l name=node-js
```

$ kubectl create -f nodejs-health-controller-2.yaml

Now the load balancer IP should work once again. Keep these pods around as we will use them again in *Chapter 3, Core Concepts – Networking, Storage, and Advanced Services*.

TCP checks

Kubernetes also supports health checks via simple TCP socket checks and also with custom command-line scripts. The following snippets are examples of what both use cases look like in the YAML file:

```
livenessProbe:
  exec:
  command: -/usr/bin/health/checkHttpServce.sh
  initialDelaySeconds:90 timeoutSeconds: 1
```

Listing 2-8: *Health check using command-line script*

```
livenessProbe: tcpSocket: port: 80
initialDelaySeconds: 15
timeoutSeconds: 1
```

Listing 2-9: *Health check using simple TCP Socket connection*

Life cycle hooks or graceful shutdown

As you run into failures in real-life scenarios, you may find that you want to take additional action before containers are shutdown or right after they are started. Kubernetes actually provides life cycle hooks for just this kind of use case.

The following example controller definition defines both a postStart and a preStop action to take place before Kubernetes moves the container into the next stage of its life cycle[1]:

```
apiVersion: v1 kind:
ReplicationController metadata:
  name: apache-hook
  labels:
name: apache-hook spec:
  replicas: 3
  selector:
name: apache-hook template:
    metadata:
    labels:
name: apache-hook spec:
```

```
containers: - name: apache-
hook
    image: bitnami/apache:latest ports: -
    containerPort: 80 lifecycle:
        postStart:
        httpGet:
        path: http://my.registration-server.com/register/ port: 80 preStop:
            exec:
                command: ["/usr/local/bin/apachectl","-k","graceful-
                    stop"]
```

Listing 2-10: apache-hooks-controller.yaml

You'll note for the postStart hook we define an httpGet action, but for the preStop hook, I define an exec action. Just as with our health checks, the httpGet action attempts to make an HTTP call to the specific endpoint and port combination while the exec action runs a local command in the container.

The httpGet and exec action are both supported for the postStart and preStop hooks. In the case of preStop, a parameter named reason will be sent to the handler as a parameter. See the following table (Table 2.1) for valid values:

Reason parameter	Failure Description
Delete	Delete command issued via kubectl or the API
Health	Health check fails
Dependency	Dependency failure such as a disk mount failure or a default infrastructure pod crash

Table 2.1. Valid preStop reasons

It's important to note that hook calls are delivered *at least once*. Therefore, any logic in the action should gracefully handles multiple calls. Another important note is that postStart runs before a pod enters its ready state. If the hook itself fails, the pod will be considered unhealthy.

Application scheduling

Now that we understand how to run containers in pods and even recover from failure, it may be useful to understand how new containers are scheduled on our cluster nodes.

As mentioned earlier, the default behavior for the Kubernetes scheduler is to spread container replicas across the nodes in our cluster. In the absence of all other constraints, the scheduler will place new pods on nodes with the least number of other pods belonging to matching services or replication controllers.

Additionally, the scheduler provides the ability to add constraints based on resources available to the node. Today, that includes minimum CPU and memory allocations. In terms of Docker, these use the **cpu-shares** and **memory limit flags** under the covers.

When additional constraints are defined, Kubernetes will check a node for available resources. If a node does not meet all the constraints, it will move to the next. If no nodes can be found that meet the criteria, then we will see a scheduling error in the logs.

The Kubernetes roadmap also has plans to support networking and storage. Because scheduling is such an important piece of overall operations and management for containers, we should expect to see many additions in this area as the project grows.

Scheduling example

Let's take a look at a quick example of setting some resource limits. If we look at our K8s dashboard, we can get a quick snapshot of the current state of resource usage on our cluster using https://<your master ip>/api/v1/proxy/namespaces/kube- system/services/kube-ui, as shown in the following screenshot:

Figure 2.13. Kube UI dashboard

In this case, we have fairly low CPU utilization, but a decent chunk of memory in use. Let's see what happens when I try to spin up a few more pods, but this time, we will request 512 Mi for memory and 1500 m for the CPU. We'll use 1500 m to specify 1.5 CPUs, since each node only has 1 CPU, this should result in failure. Here's an example of RC definition:

```
apiVersion: v1 kind:
ReplicationController metadata:
  name: node-js-constraints labels:
name: node-js-constraints spec:
  replicas: 3
  selector:
name: node-js-constraints template:
    metadata:
    labels:
name: node-js-constraints spec:
      containers: - name: node-js-constraints
        image: jonbaier/node-express-info:latest ports: -
        containerPort: 80 resources: limits:
          memory: "512Mi"
          cpu: "1500m"
```

Listing 2-11: nodejs-constraints-controller.yaml

To open the preceding file, use the following command:

$ kubectl create -f nodejs-constraints-controller.yaml

The replication controller completes successfully, but if we run a get pods command, we'll note the node-js-constraints pods are stuck in a pending state. If we look a little closer with the describe pods/<pod-id> command, we'll note a scheduling error:

$ kubectl get pods

$ kubectl describe pods/<pod-id>

The following screenshot is the result of the preceding command:

Figure 2.14. Pod description

Note that the **failedScheduling** error listed in events is accompanied by **Failed for reason PodFitsResources and possibly others** on our screen. As you can see, Kubernetes could not find a fit in the cluster that met all the constraints we defined.

If we now modify our CPU constraint down to 500 m, and then recreate our replication controller, we should have all three pods running within a few moments.

Summary

We've taken a look at the overall architecture for Kubernetes as well as the core constructs provided to build your services and application stacks. You should have a better understanding of how these abstractions make it easier to manage the life cycle of your stack and/or services as a whole and not just the individual components. Additionally, we took a first-hand look at how to manage some simple day-to-day tasks using pods, services, and replication controllers. We also looked at how to use Kubernetes to automatically respond to outages via health checks. Finally, we explored the Kubernetes scheduler and some of the constraints users can specify to influence scheduling placement.

Footnotes

[1] https://github.com/GoogleCloudPlatform/kubernetes/blob/release-1.0/docs/user-guide/container-environment.md#container-hooks