Getting Started With

Kubernetes

UPDATED BY **CHRIS JUDD**
ORIGINAL BY **ARUN GUPTA**

CONTENTS

WHAT IS KUBERNETES?

Kubernetes (kubernetes.io) is an open source orchestration system for managing containerized applications across multiple hosts, providing basic mechanisms for the deployment, maintenance, and scaling of applications. Originally created by Google, in March of 2016 it was donated to the Cloud Native Computing Foundation (CNCF).

Kubernetes, or "k8s" or "kube" for short, allows the user to declaratively specify the desired state of a cluster using high-level primitives. For example, the user may specify that they want three instances of the Couchbase server container running. Kubernetes' self-healing mechanisms, such as auto-restarting, re-scheduling, and replicating containers then converge the actual state towards the desired state.

Kubernetes supports Docker and Rocket containers. An abstraction around the containerization layer will allow for other container image formats and runtimes to be supported in the future.

KEY CONCEPTS OF KUBERNETES

POD

A Pod is the smallest deployable unit that can be created, scheduled, and managed. It's a logical collection of containers that belong to an application.

Each resource in Kubernetes is defined using a configuration file. For example, a Couchbase pod can be defined with the following `.yaml` file:

```
apiVersion: v1
kind: Pod
# labels attached to this Pod
metadata:
  name: couchbase-pod
  labels:
    name: couchbase-pod
spec:
  containers:
  - name: couchbase
    # Docker image that will run in this Pod
    image: couchbase
    ports:
    - containerPort: 8091
```

LABEL

A label is a key/value pair that is attached to objects, such as pods. In the previous example, `metadata.labels` define the labels attached to the pod.

Labels define identifying attributes for the object and is only meaningful and relevant to the user. Multiple labels can be attached to a resource. Labels can be used to organize and to select subsets of objects.

REPLICA SETS

A replica set ensures that a specified number of pod replicas are running on worker nodes at any one time. It allows both up- and down-scaling the number of replicas. It also ensures recreation of a pod when the worker node reboots or otherwise fails.

NOTE: Replica Sets replaces Replication Controllers.

A Replica Set creating two instances of a Couchbase pod can be defined as:

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: couchbase-rs
spec:
  # Two replicas of the Pod to be created
  replicas: 2
  # Identifies the label key and value on the Pod that
  # this Replica Set is responsible for managing
  selector:
    matchLabels:
      app: couchbase-rs-pod
    matchExpressions:
      - {key: tier, operator: In, values: ["backend"]}
  template:
    metadata:
      labels:
        # label key and value on the pod.
        # These must match the selector above.
        app: couchbase-rs-pod
        tier: backend
    spec:
      containers:
      - name: couchbase
        image: couchbase
        ports:
        - containerPort: 8091
```

SERVICE

Each Pod is assigned a unique IP address. If the Pod inside a Replication Set dies, when it the pod is recreated it may be given a different IP address. This makes it difficult for an application server, such as WildFly, to access a database, such as Couchbase, using its IP address.

A Service defines a logical set of Pods and a policy by which to access them. The IP address assigned to a Service does not change over time, and thus can be relied upon by other Pods. In addition, pods can find the services using service discovery either via environment variables or DNS.

NOTE: You can combine a Service and Replica Set in the same yaml file by separating them with ---.

For example, the following creates a comprehensive Couchbase Service and an application Service running in Wildfly exposed via port 30080 that discovers the Couchbase Service using the COUCHBASE_URI environment variable and the couchbase-svc DNS value:

```
apiVersion: v1
kind: Service
metadata:
  name: couchbase-svc
spec:
  selector:
    app: couchbase-rs-pod
  ports:
    - name: admin
      port: 8091
    - name: views
      port: 8092
    - name: query
      port: 8093
    - name: memcached
      port: 11210
---
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: couchbase-rs
spec:
  replicas: 1
  selector:
    matchLabels:
      app: couchbase-rs-pod
    matchExpressions:
      - {key: tier, operator: In, values: ["backend"]}
  template:
    metadata:
      labels:
        app: couchbase-rs-pod
        tier: backend
    spec:
      containers:
      - name: couchbase
        image: arungupta/couchbase:travel
        ports:
        - containerPort: 8091
        - containerPort: 8092
        - containerPort: 8093
        - containerPort: 11210
---
apiVersion: v1
kind: Service
metadata:
  name: wildfly-svc
spec:
  type: NodePort
  selector:
    app: wildfly-rs-pod
  ports:
    - name: http
port: 8080
      nodePort: 30080
```

```
---
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: wildfly-rs
spec:
  replicas: 1
  selector:
    matchLabels:
      app: wildfly-rs-pod
    matchExpressions:
      - {key: tier, operator: In, values: ["frontend"]}
template:
    metadata:
      labels:
        app: wildfly-rs-pod
        tier: frontend
    spec:
      containers:
      - name: wildfly
        image: arungupta/wildfly-couchbase-javaee7
        env:
        - name: COUCHBASE_URI
          value: couchbase-svc
        ports:
        - containerPort: 8080
```

VOLUMES

A Volume is a directory on disk or in another container. A volume outlives any containers that run within the Pod, and the data is preserved across Container restarts. The directory, the medium that backs it, and the contents within it are determined by the particular volume type used.

Multiple types of volumes are supported. Some of the commonly used volume types are shown below:

VOLUME TYPE	MOUNTS INTO YOUR POD
hostPath	A file or directory from the host node's filesystem
nfs	Existing Network File System share
awsElasticBlockStore	An Amazon Web Service EBS Volume
gcePersistentDisk	A Google Compute Engine Persistent Disk

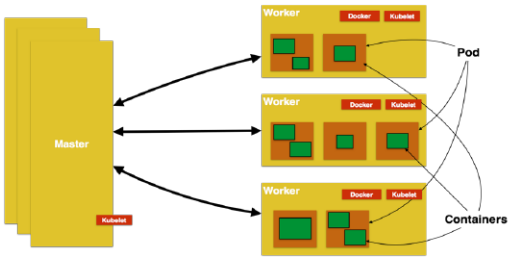
A Volume is specified in the Pod configuration file as shown:

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: couchbase-rs
spec:
  replicas: 1
  selector:
    matchLabels:
      app: couchbase-rs-pod
    matchExpressions:
      - {key: tier, operator: In, values: ["backend"]}
  template:
    metadata:
      labels:
        app: couchbase-rs-pod
        tier: backend
    spec:
      containers:
      - name: couchbase
        image: couchbase
        ports:
        - containerPort: 8091
        volumeMounts:
        - mountPath: /var/couchbase/lib
          name: couchbase-data
      volumes:
      - name: couchbase-data
        hostPath:
          path: /tmp/couchbase
```

NOTE: hostPath is a fine option for testing, but it is not suitable for production use.

KUBERNETES ARCHITECTURE

A Kubernetes architecture with some key components is shown here:



A Kubernetes cluster is a set of physical or virtual machines and other infrastructure resources that are used to run your applications. The machines that manage the cluster are called *Master Nodes* and the machines that run the containers are called *Worker Nodes*.

NODE

A Node is a physical or virtual machine. It has the necessary services to run application containers.

A Master Node is the central control point that provides a unified view of the cluster. Multiple masters can be setup to create a highly-available cluster.

A Worker Node runs tasks as delegated by the master. Each Worker Node can run multiple pods.

KUBELET

Kubelet is a service running on each Node that manages containers and is managed by the master. This service reads container manifests as YAML or JSON files that describe each Pod. A typical way to provide this manifest is using the configuration file as shown in the previous sections. Kubelet ensures that the containers defined in the Pods are started and continue running.

Kubelet is a Kubernetes-internal concept and generally does not require direct manipulation

GETTING STARTED WITH KUBERNETES

SETTING UP KUBERNETES

There are a variety of ways to setup, configure, and run Kubernetes. It can be run in the cloud using providers such as Amazon Web Services (AWS), Google Compute Engine, Azure, Packet, and others. It can be also run on-premise by building a cluster from scratch on physical hardware or via virtual machines. You can find out which solution is correct for you, as well as step-by-step instructions at kubernetes.io/docs/setup/. You can find a helpful quick-start guide to setting up Kubernetes on bare metal at blog.alexellis.io/kubernetes-in-10-minutes/. The recommended way to get started and run a single-node cluster for development and testing is to use Minikube.

MINIKUBE

Minikube uses virtualization software like VirutalBox, VMware, kvm, or xhyve to run the cluster. It also depends on the kubectl for interacting with the cluster. For instructions to install Minikube and its dependencies, visit kubernetes.io/docs/tasks/tools/install-minikube.

Once Minikube is installed, you can use the minikube command-line to start a cluster by running the following command:

```
minikube start
```

To stop the cluster, you can run:

```
minikube stop
```

To determine the IP address of the cluster use:

```
minikube ip
```

If you are having problems, you can view the logs or ssh into the host to help debug the issue by using:

```
minikube logs
minikube ssh
```

Most interestingly, you can open a dashboard view in the browser to see and change what is going on in the cluster.

```
minikube dashboard
```

KUBECTL CLI

`kubectl` is a command-line utility that controls the Kubernetes cluster. This utility can be used in the following format:

```
kubectl [command] [type] [name] [flags]
```

- `[command]` specifies the operation that needs to be performed on the resource. For example, `create`, `get`, `describe`, `delete`, or `scale`.
- `[type]` specifies the Kubernetes resource type. For example, pod (`po`), service (`svc`), replicaset (`rs`), or node (`no`). Resource types are case-sensitive, and you can specify the singular, plural, or abbreviated forms.
- `[name]` Specifies the name of the resource. Names are case-sensitive. If the name is omitted, details for all resources will be displayed (for example, `kubectl get pods`).

Some examples of kubectl commands and their purpose:

COMMAND	PURPOSE
`kubectl create -f couchbase-pod.yml`	Create a Couchbase pod
`kubectl create -f couchbase-rs.yml`	Create a Couchbase Replica Set
`kubectl delete -f couchbase-pod.yml`	Deletes/Removes the Couchbase pod
`kubectl get pods`	List all the pods
`kubectl describe pod couchbase-pod`	Describe the Couchbase pod
`kubectl --help`	Shows the complete list of available commands

RUN YOUR FIRST CONTAINER

A Container can be started on a Kubernetes cluster using the kubectl script. The easiest way to do this is to specify the Docker image name to the run command:

```
kubectl run couchbase --image=arungupta/couchbase
deployment "couchbase" created
```

This command will start a pre-configured Couchbase container in a Pod wrapped inside a Replica Set wrapped inside a Deployment. The status of the Deployment can be seen:

```
kubectl get deploy
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
couchbase   1         1         1            1           1m
```

The status of this RS can be seen by using:

```
kubectl get rs
NAME                   DESIRED   CURRENT   READY     AGE
couchbase-3179999270   1         1         1         39s
```

The status of the Pod can be seen by using:

```
kubectl get po
NAME                         READY   STATUS RESTART AGE
couchbase-3179999270-8r1xs   1/1   Running   0     2m
```

Alternatively, the Container can also be started by using the configuration file:

```
kubectl create -f couchbase-pod.yaml
```

The file `couchbase-pod.yaml` contains the Pod definition, as explained earlier.

SCALE APPLICATIONS

Pods in a Replica Set can be scaled up and down:

```
kubectl scale --replicas=3 deploy/couchbase
deployment "couchbase" scaled
```

Then the updated number of deployments can be seen:

```
kubectl get deploy
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
couchbase   3         3         3            3           3m
```

Note, the updated number of replicas here is 3. The image, `arungupta/couchbase` in this case, will need to ensure that the cluster can be formed using three individual instances. You can display the instances by running:

```
kubectl get po
NAME                         READY   STATUS   RESTARTS AGE
couchbase-3179999270-54qvb   1/1   Running   0      14s
couchbase-3179999270-5h5jb   1/1   Running   0      14s
couchbase-3179999270-z9hl6   1/1   Running   0      3m
```

DELETE APPLICATIONS

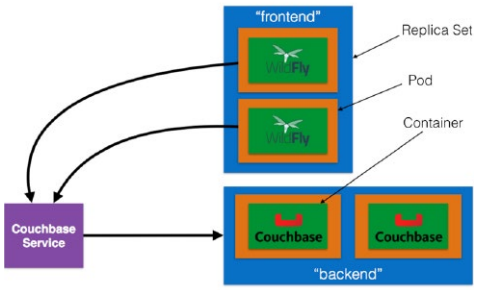
Once you are done using the application, you can destroy it with the delete command.

```
kubectl delete deployment couchbase
```

If for some reason an instance crashes or gets shut down, Kubernetes will immediately spin up another one. To destroy the whole thing, you must delete it at the deployment level.

APPLICATION USING MULTIPLE CONTAINERS

Typical applications consist of a "frontend" and a "backend." The "frontend" would typically be an application server, and the "backend" would typically be a database. For this example, we'll use WildFly for our application server and Couchbase for our database.



The steps involved are:

- Start the "backend" Replica set: The Couchbase Replica Set should contain the spec for a Couchbase Pod. The template should include metadata that will be used by the Service.
- Start the "backend" Service: The Couchbase Service uses the selector to select the previously started Pods.
- Start the "frontend" Replica Set: The WildFly Replica Set should contain the spec for the WildFly pod. The Pod should include the application predeployed. This is typically done by extending WildFly's Docker image, copying the WAR file in the /opt/jboss/wildfly/standalone/deployments directory, and creating a new Docker image. The application can connect to the database by discovering "backend" services using Environment Variables or DNS.

NOTE: The Service example above does this all in one file.

NAMESPACE, RESOURCE QUOTAS, AND LIMITS

By default, all user resources in the Kubernetes cluster are created in a namespace called `default`. Objects created by Kubernetes are in the `kube-system` namespace.

By default, a pod runs with unbounded CPU and memory requests/limits.

A resource can be created in a different namespace and assigned different memory requests/limits to meet the application's needs.

Resources created by the user can be partitioned into multiple namespaces. Resources created in one namespace are hidden from a different namespace. This allows for a logical grouping of resources.

Each namespace provides:

- A unique scope for resources to avoid name collisions
- Policies to ensure appropriate authority to trusted users
- The ability to specify constraints for resource consumption

A new namespace can be created using the following configuration file:

```
apiVersion: v1
kind: Namespace
metadata:
  name: development
  labels:
    name: development
```

A Replica Set in the default namespace can be created:

```
kubectl create -f couchbase-rs.yml
replicaset "couchbase" created
```

And a Replica Set in the new namespace can be created:

```
kubectl --namespace=development create -f couchbase-rs.yml
replicaset "couchbase" created
```

A list of replication controllers in all namespaces can be obtained:

```
kubectl get rs --all-namespaces
NAMESPACE   NAME                DESIRED  CURRENT  READY  AGE
default     couchbase-rs          2        2        2     1m
development couchbase-rs          2        2        2     46s
kube-system kube-dns-1301475494   1        1        1     56d
```

Specifying a quota allows you to restrict how much of a cluster's resources can be consumed across all pods in a namespace.

Resource quotas can be specified using a configuration file:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: quota
spec:
  hard:
    cpu: "20"
    memory: 1Gi
    pods: "10"
    resourcequotas: "1"
    services: "5"
```

Now a pod can be created specifying limits:

```
apiVersion: v1
kind: Pod
metadata:
  name: couchbase-pod
spec:
  containers:
  - name: couchbase
    image: couchbase
    ports:
    - containerPort: 8091
    resources:
      limits:
        cpu: "1"
        memory: 512Mi
```

Namespace, resource quota, and limits allow a Kubernetes cluster to share the resources of multiple groups and provide different levels of QoS for each group.

ABOUT THE AUTHOR

CHRISTOPHER M. JUDD is the CTO and a partner at Manifest Solutions (manifestcorp.com), an international speaker, an open source evangelist, the Central Ohio Java Users Group (www.cojug.org) and Columbus iPhone Developer User Group leader, and the co-author of Beginning Groovy and Grails (Apress, 2008), Enterprise Java Development on a Budget (Apress, 2003), and Pro Eclipse JST (Apress, 2005), as well as the author of the children's book "Bearable Moments." He has spent 20 years architecting and developing software for Fortune 500 companies in various industries, including insurance, health care, retail, government, manufacturing, service, and transportation. His currentfocus is on consulting, mentoring, and training with Java, Java EE,Groovy, Grails, Cloud Computing, and mobile platforms like iPhone, Android, Java ME, and mobile web.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513

888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com