CONTENTS

Kubernetes

BY ARUN GUPTA

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.

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:
  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 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.

REPLICATION CONTROLLER

A replication controller ensures that a specified number of pod replicas are running on worker nodes at all times. 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.

A Replication Controller creating two instances of a Couchbase pod can be defined as:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: couchbase-controller
spec:
  # Two replicas of the Pod to be created
  replicas: 2
  # Identifies the label key and value on the Pod that
  # this Replication Controller is responsible for
  managing
  selector:
    app: couchbase-rc-pod
  # 'cookie cutter' used for creating new pods when
  necessary
  template:
    metadata:
      labels:
        # label key and value on the pod.
        # These must match the selector above.
        app: couchbase-rc-pod
    spec:
      containers:
      - name: couchbase
        image: couchbase
        ports:
        - containerPort: 8091
```

SERVICE

Each Pod is assigned a unique IP address. If the Pod is inside a Replication Controller, then the pod is recreated but 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. Typically, the Pods belonging to a Service are defined by a label `selector`.

For example, a Couchbase service might be defined as:

DC/OS

Use Containers at Scale

The easiest way to run Docker & JAR Java apps at scale. 100% Open Source.

Build apps quickly using a rich ecosystem. Get Kafka, Cassandra, Jenkins, Spark, ArangoDB, and other datacenter-scale services running in minutes. DC/OS is 100% open source and built with the experience of Mesosphere, Yelp, Twitter, and Airbnb. Run your infrastructure agnostic application with DC/OS on any bare-metal, private and public clouds, or your laptop.

Get Started

dcos.io/get-started

amazon web services

Microsoft

VAGRANT

packet

CoreOS

CentOS

```
apiVersion: v1
kind: Service
metadata:
  name: couchbase-service
  labels:
    app: couchbase-service-pod
spec:
  ports:
    - port: 8091
  # label keys and values of the Pod started elsewhere
  selector:
    app: couchbase-rc-pod
```

Note that the labels used in `selector` **must** match the metadata used for creating the Pod by the Replication Controller.

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

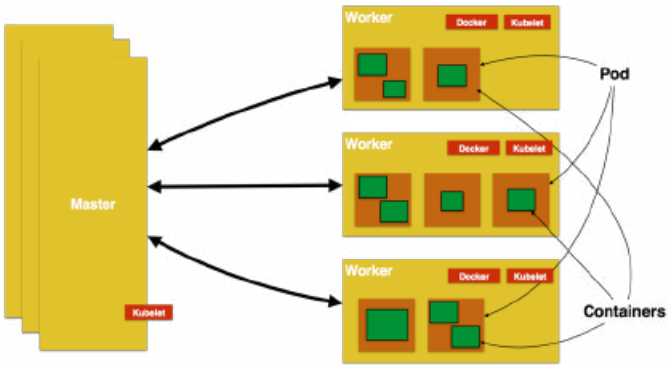
A Volume is specified in the Pod configuration file as shown:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: couchbase-controller
spec:
  replicas: 1
  # In-line template of the Pod
  template:
    metadata:
      app: couchbase-rc-pod
    spec:
      containers:
        - name: couchbase-rc-pod
          image: arungupta/couchbase
          ports:
          - containerPort: 8091
          volumeMounts:
          # name must match the volume name below
          - name: nfs
            mountPath: /usr/share/couchbase
      volumes:
        - name: nfs
          persistentVolumeClaim:
            claimName: nfs
```

This configuration file also shows that a Pod template can be specified within a `ReplicationController` specification.

KUBERNETES ARCHITECTURE

Kubernetes architecture with key components is shown:



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

START THE KUBERNETES CLUSTER

A Kubernetes cluster can be started in multiple ways. The most common ones use Vagrant, Amazon Web Service (AWS), Google Compute Engine (GCE), and Azure. This link provides complete details about different options.

The latest Kubernetes release can be downloaded here. This includes the binary to start the cluster and the kubectl script to manage this cluster.

Alternatively, the cluster can also be started as `curl -sS https://get.k8s.io | bash`.

The `KUBERNETES_PROVIDER` environment variable defines which variant to use. A cluster can be started as:

```
./cluster/kube-up.sh
```

Additional worker nodes can be created by setting the environment variable `NUM_MINIONS`, for example:

```
export NUM_MINIONS=6
```

A cluster can be shut down with:

```
./cluster/kube-down.sh
```

Variant-specific configurations for Vagrant, Amazon, GCE, and Azure are shown next.

START THE CLUSTER USING VAGRANT

Running Kubernetes with Vagrant is an easy way to run, develop, and test on your local machine.

A Kubernetes cluster using Vagrant can be started with:

```
export KUBERNETES_PROVIDER=vagrant
./cluster/kube-up.sh
```

By default, the Vagrant will create two Fedora VMs—one for the master node and one for the worker node. The status of the created VMs can be seen using the `vagrant status` command, for example:

```
vagrant status
Current machine states:

master                    running (virtualbox)
minion-1                  running (virtualbox)
```

By default, each VM is assigned 1GB memory. A different number can be assigned by setting the `KUBERNETES_MEMORY` environment variable, for example:

```
export KUBERNETES_MEMORY=2048
```

Complete instructions to run and manage a Kubernetes cluster using Vagrant are available at: kubernetes.io/v1.1/docs/getting-started-guides/vagrant.html.

START THE CLUSTER USING AWS

Running Kubernetes with AWS requires:

- An AWS account
- Installation and configuration of AWS CLI
- An AWS instance and profile with EC2 full access

Set `KUBERNETES_PROVIDER` to `aws` with:

```
export KUBERNETES_PROVIDER=aws
```

Start and configure the cluster as explained earlier.

By default, the script will provide a new VPC and a 4 node Kubernetes cluster in `us-west-2a` (Oregon) with `t2.micro` instances running on Ubuntu. These, and other values, such as memory allotment for Master and Worker node, can be configured in `cluster/aws/config-default.sh`.

START THE CLUSTER USING GCE

Running Kubernetes with GCE requires:

- A Google Cloud Platform account with billing enabled
- Installation and configuration of the Google Cloud SDK as explained at kubernetes.io/v1.1/docs/getting-started-guides/gce.html

Either unset `KUBERNETES_PROVIDER` or set it to `gce` as:

```
export KUBERNETES_PROVIDER=gce
```

Start and configure the cluster as explained earlier.

By default, the script will provision a single Master node and 4 Worker nodes in `us-central1-b` zone with `n1-standard-1` instances running on Debian. These, and other values, such as memory allotment for Master and Worker node, can be configured in `cluster/gce/config-default.sh`.

START THE CLUSTER USING AZURE

Running Kubernetes with Azure requires:

- Azure account
- Installation and configuration of the Azure CLI

Set `KUBERNETES_PROVIDER` to `azure` as:

```
export KUBERNETES_PROVIDER=azure
```

You also need to set `AZURE_TENANT_ID` and `AZURE_SUBSCRIPTION_ID`. Values for these can be obtained using the command `azure account show`.

Start and configure the cluster as explained earlier.

By default, the script will provision a single Master node and 3 Worker nodes in `westus` zone with `Standard_A1` instances. These, and other values can be configured in `cluster/azure/config-default.sh`.

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`, `describe`, `delete`, or `scale`.
- `[type]` specifies the Kubernetes resource type. For example, `pod`, `service`, `replicationcontroller`, or `node`. 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-rc.yml`	Create a Couchbase Replication Controller
`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.sh run couchbase --image=arungupta/couchbase
```

This command will start a pre-configured Couchbase container in a Pod wrapped inside a Replication Controller. The status of this RC can be seen:

```
kubectl.sh get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)
  SELECTOR        REPLICAS   AGE
couchbase    couchbase      arungupta/couchbase
  run=couchbase   1          16s
```

The status of the Pod can be seen:

```
kubectl.sh get po
NAME              READY     STATUS    RESTARTS   AGE
couchbase-0s8lx   1/1       Running   0          1m
```

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

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

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

SCALE APPLICATIONS

Pods in a replication controller can be scaled up and down:

```
kubectl.sh scale --replicas=3 rc couchbase
replicationcontroller "couchbase" scaled
```

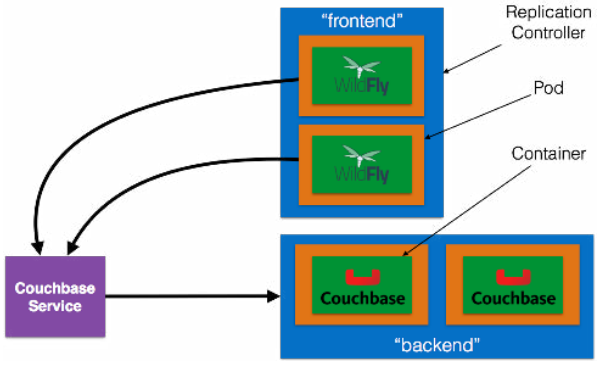
Then the updated number of replicas can be seen:

```
kubectl.sh get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)
  SELECTOR        REPLICAS   AGE
couchbase    couchbase      arungupta/couchbase
  run=couchbase   3          3m
```

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

APPLICATION USING MULTIPLE CONTAINERS

Typical applications consist of a "frontend" and a "backend." The "frontend" would typically be an application server, such as WildFly. The "backend" would typically be a database, such as Couchbase.



The steps involved are:

- **Start the "backend" Replication Controller:** The Couchbase Replication Controller 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" Replication Controller:** The WildFly Replication Controller 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.

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
- 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 replication controller in the default namespace can be created:

```
kubectl.sh create -f couchbase-rc.yml
replicationcontroller "couchbase" created
```

And a replication controller in the new namespace can be created:

```
kubectl.sh --namespace=development create -f couchbase-
  rc.yml
replicationcontroller "couchbase" created
```

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

```
kubectl.sh get rc --all-namespaces
NAMESPACE     CONTROLLER
  CONTAINER(S)            IMAGE(S)
  SELECTOR                              REPLICAS   AGE
default       couchbase                            couchbase
  arungupta/couchbase
  run=couchbase                         1          4m
development   couchbase                            couchbase
  arungupta/couchbase
  run=couchbase                         1          2m
```

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”
    replicationcontrollers: “20”
    resourcequotas: “1”
    services: “5”
```

Now a pod can be created by the 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

ARUN GUPTA is the vice president of developer advocacy at Couchbase. He has built and led developer communities for 10+ years at Sun, Oracle, and Red Hat. He has a deep expertise in leading cross-functional teams to develop and execute strategy, planning and execution of content, marketing campaigns, and programs. Prior to that he led engineering teams at Sun and is a founding member of the Java EE team.

Gupta has authored more than 2,000 blog posts on technology. He has extensive speaking experience in more than 40 countries on a myriad of topics and is a JavaOne RockStar for three years in a row. Gupta also founded the Devoxx4Kids chapter in the US and continues to promote technology education among children. An author of a best-selling book, an avid runner, a globe trotter, a Java Champion, a JUG leader, and a Docker Captain, he is easily accessible at @arungupta.

RESOURCES

Kubernetes docs:

kubernetes.io/docs

Kubernetes Issue Tracker:

github.com/kubernetes/kubernetes

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

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

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795

BROUGHT TO YOU IN PARTNERSHIP WITH
MESOSPHERE