Enabling Microservices

Containers & Orchestration Explained
March 2016

Table of Contents

Introduction

Want to try out MongoDB on your laptop? Execute a single command and you have a lightweight, self-contained sandbox; another command removes all trace when you're done.

Need an identical copy of your application stack in multiple environments? Build your own container image and then your entire development, test, operations, and support teams can launch an identical clone environment.

Containers are revolutionizing the entire software lifecycle: from the earliest technical experiments and proofs of concept through development, test, deployment, and support.

Orchestration tools manage how multiple containers are created, upgraded and made highly available. Orchestration also controls how containers are connected to build sophisticated applications from multiple, microservice containers.

The rich functionality, simple tools, and powerful APIs make container and orchestration functionality a favorite for DevOps teams who integrate them into Continuous Integration (CI) and Continuous Delivery (CD) workflows.

This white paper introduces the concepts behind containers and orchestration, then explains the available technologies and how to use them with MongoDB.

What are Containers?

To illustrate the concepts associated with software containers, it is helpful to consider a similar example from the physical world – shipping containers.

Shipping containers are efficiently moved using different modes of transport – perhaps initially being carried by a truck to a port, then neatly stacked alongside thousands of other shipping containers on a huge container ship that carries them to the other side of the world. At no point in the journey do the contents of that container need to repacked or modified in any way.

Shipping containers are ubiquitous, standardized, and available anywhere in the world, and they're extremely simple to use – just open them up, load in your cargo, and lock the doors shut.

The contents of each container are kept isolated from that of the others; the container full of Mentos can safely sit next to the container full of soda without any risk of a reaction. Once a spot on the container ship has been booked, you can be confident that there's room for all of your packed cargo for the whole trip – there's no way for a neighboring container to steal more than its share of space.

Software containers fulfill a similar role for your application. Packing the container involves defining what needs to be there for your application to work – operating system, libraries, configuration files, application binaries, and other parts of your technology stack. Once the container has been defined, that *image* is used to create containers that run in any environment, from the developer's laptop to your test/QA rig, to the production data center, on-premises or in the cloud, without any changes. This consistency can be very useful: for example, a support engineer can spin up a container to replicate an issue and be confident that it exactly matches what's running in the field.

Containers are very efficient and many of them can run on the same machine, allowing full use of all available resources. Linux containers and cgroups are used to make sure that there's no cross-contamination between containers: data files, libraries, ports, namespaces, and memory contents are all kept isolated. They also enforce upper boundaries on how much system resource (memory, storage, CPU, network bandwidth, and disk I/O) a container can consume so that a critical application isn't squeezed out by noisy neighbors.

Metaphors tend to fall apart at some point, and that's true with this one as well. There are exceptions but shipping containers typically don't interact with each other – each has its job to fulfill (keep its contents together and safe during shipping) and it doesn't need help from any of its peers to achieve that. In contrast, it can be very powerful to have software containers interact with each other through well defined interfaces – e.g., one container provides a database service that an application running in another container can access through an agreed port. The modular container model is a great way to implement microservice architectures.

Containers Compared to Virtual Machines (VMs)

There are a number of similarities between virtual machines (VMs) and containers – in particular, they both allow you to create an image and spin up one or more instances, then safely work in isolation within each one. Containers, however, have a number of advantages which make them better suited to building and deploying applications.

Each instance of a VM must contain an entire operating system, all required libraries, and of course the actual application binaries. All of that software consumes several Gigabytes of storage and memory. In contrast, each container holds its application and any dependencies, but the same Linux kernel and libraries can be shared between multiple containers running on the host. The fact that each container imposes minimal overhead on storage, RAM, and CPU means that many can run on the same host, and each takes just a couple of seconds to launch.

Running many containers allows each one to focus on a specific task; multiple containers then work in concert to implement sophisticated applications. In such *microservice* architectures, each container can use different versions of programming languages and libraries that can be upgraded independently.

Due to the isolation of capabilities within containers, the effort and risk associated with updating any given container is far lower than with a more monolithic architecture. The lends itself to Continuous Delivery – an approach that involves fast software development iterations and frequent, safe updates to the deployed application.

The tools and APIs provided with container technologies such as Docker are very powerful and more developer-focused than those available with VMs. These APIs allow the management of containers to be integrated into automated systems – such as Chef and Puppet – used by DevOps teams to cover the entire software development lifecycle. This has led to wide scale adoption by DevOps-oriented groups.

Virtual machines still have an essential role to play, as you'll very often be running your containers within VMs –

including when using the cloud services provided by Amazon, Google, or Microsoft.

How Containers Benefit Your business

DevOps & Continuous Delivery. When the application consists of multiple containers with clear interfaces between them, it is a simple and low-risk matter to update a container, assess the impact, and then either revert to the old version or roll the update out across similar containers. By having multiple containers provide the same capability, upgrading each container can be done without negatively affecting service.

Replicating Environments. When using containers, it's a trivial matter to instantiate identical copies of your full application stack and configuration. These can then be used by new hires, partners, support teams, and others to safely experiment in isolation.

Accurate Testing. You can have confidence that your QA environment exactly matches what will be deployed – down to the exact version of every library.

Scalability. By architecting an application to be built from multiple container instances, adding more containers scales out capacity and throughput. Similarly, containers can be removed when demand falls. Using orchestration frameworks – such as Kubernetes and Apache Mesos – further simplifies elastic scaling.

Isolation. Every container running on the same host is independent and isolated from the others as well as from the host itself. The same equipment can simultaneously host development, support, test, and production versions of your application – even running different versions of tools, languages, databases, and libraries without any risk that one environment will impact another.

Performance. Unlike VMs (whether used directly or through Vagrant), containers are lightweight and have minimal impact on performance.

High Availability. By running with multiple containers, redundancy can be built into the application. If one container fails, then the surviving peers – which are providing the same capability – continue to provide service. With the addition of some automation (see the orchestration section of this paper), failed containers can be automatically recreated (rescheduled) either on the same or a different host, restoring full capacity and redundancy.

Docker – the Most Popular Container Technology

The simplicity of Docker and its rich ecosystem make it extremely powerful and easy to use.

Specific Docker containers are created from images which have been designed to provide a particular capability – whether that be, for example, just a base operating system, a web server, or a database. Docker images are constructed from layered filesystems so they can share common files, reducing disk usage and speeding up image download. Docker Hub provides thousands of images, that can be extended or used as is, to quickly create a container that's running the software you want to use – for example, all it takes to get MongoDB up and running is the command `docker run --name my-mongodb -d mongo` which will download the image (if it's not already on the machine) and use it to start the container. Proprietary images can be made available within the enterprise using a local, private registry rather than Docker Hub.

Docker containers are based on open standards, allowing containers to run on all major Linux distributions. They support bare metal, VMs, and cloud infrastructure from vendors such as Amazon, Google, and Microsoft. Integration with cloud services – e.g., with the Google Container Engine (GCE) – means that running your software in a scalable, highly available configuration is just a few clicks away.

Docker provides strong isolation where each container has its own root filesystem, processes, memory, network ports, namespace, and devices. But to be of use, containers need to be able to communicate with the outside world as well as other containers. To this end, Docker containers can be configured to expose ports as well as map volumes to directories on the host. Alternatively, Docker containers can be linked so that they communicate without opening up these resources to other systems.

Orchestration

Clearly, the process of deploying multiple containers to implement an application can be optimized through automation. This becomes more and more valuable as the number of containers and hosts grow. This type of automation is referred to as orchestration. Orchestration can include a number of features, including:

- Provisioning hosts
- Instantiating a set of containers
- Rescheduling failed containers
- Linking containers together through agreed interfaces
- Exposing services to machines outside of the cluster
- Scaling out or down the cluster by adding or removing containers

A common term used in orchestration is ***scheduling*** – this refers to the orchestration framework deciding on which host a container should run and then starting the container there. ***Rescheduling*** refers to restarting a container, either on the same host or elsewhere – e.g., when the container's existing host restarts.

There are many orchestration tools available for Docker; some of the most common are described here.

Docker Machine

Docker Machine provisions hosts and installs Docker Engine (the lightweight runtime and tooling used to run Docker containers) software on them.

Docker Machine provides commands for starting, stopping, and restarting hosts in addition to upgrading the Docker client and daemon.

Docker Machine is particularly convenient when using a Windows or OS X machine as it can automatically create a Linux VM running on VirtualBox in which the Docker process runs. It is also able to create hosts in cloud environments, including: AWS, Azure, and Digital Ocean.

Docker Swarm

Docker Swarm produces a single, virtual Docker host by clustering multiple Docker hosts together. It presents the same Docker API; allowing it to integrate with any tool that works with a single Docker host.

A common practice is for Docker Swarm to employ Docker Machine to create the hosts making up the swarm – especially early on in the development process.

Docker Swarm can grow with your needs as it allows for pluggable scheduler backends. You can start off with the default scheduler but swap in Mesos (see below) for large, production deployments.

Docker Compose

Docker Compose takes a file defining a multi-container application (including dependencies) and deploys the described application by creating the required containers. It is mostly aimed at development, testing, and staging environments.

Benefits of using Docker Compose include:

- A single host can run multiple, isolated environments
- Data is preserved when containers are shut down and restarted.
- It determines which containers for a project are already running, and which need to be started
- Compose files can be reused and extended between projects

Kubernetes

Kubernetes was created by Google and is one of the most feature-rich and widely used orchestration frameworks; its key features include:

- Automated deployment and replication of containers
- Online scale-in or scale-out of container clusters
- Load balancing over groups of containers
- Rolling upgrades of application containers
- Resilience, with automated rescheduling of failed containers

- Controlled exposure of network ports to systems outside of the cluster

Kubernetes is designed to work in multiple environments, including bare metal, on-premises VMs, and public clouds. Google Container Engine provides a tightly integrated platform which includes hosting of the Kubernetes and Docker software, as well as provisioning the host VMs and orchestrating the containers.

The key components making up Kubernetes are:

- A **Cluster** is a collection of one or more bare-metal servers or virtual machines (referred to as *nodes*) providing the resources used by Kubernetes to run one or more applications.
- **Pods** are groups of containers and volumes co-located on the same host. Containers in the same Pod share the same network namespace and IP address (unique only within the cluster) and can communicate with each other using `localhost`. Pods are considered to be ephemeral, rather than durable entities, and are the basic scheduling unit. The structure, contents, and interfaces for a pod are defined using either a JSON or YAML configuration file.
- **Volumes** map ephemeral directories within a container to persistent storage which survives container restarts and rescheduling. Volumes also allow data to be shared amongst containers within a pod.
- **Services** act as basic load balancers and ambassadors for other containers, exposing them to the outside world. e.g., A service can provide a static, external IP Address & port which it maps to another (internal to the cluster) port on multiple containers.
- **Labels** are tags assigned to entities such as containers that allow them to be managed or referenced as a group. One resource can reference one or more other resources by including their label(s) in its `selector`. e.g., containers in a cluster might have labels for *environment*, *role* and *location*; a service could be setup to act as an interface to a subset of those containers by setting its selector to `environment=production, role=web-server, location=new-york`.
- A **Replication Controller** handles the scheduling of pods across the cluster. When configuring a Replication Controller, you specify the required numbers of pods, and of which type, which should exist at any point in time. If a pod fails then the Replication Controller creates a replacement; if a request is made to increase the size of the cluster then the Replication Controller starts the additional pods.

Mesos

Apache Mesos is designed to scale to tens of thousands of physical machines. Mesos is in production with a number of large enterprises such as Twitter, Airbnb, and Apple. An application running on top of Mesos is made up of one or more containers and is referred to as a *framework*. Mesos offers resources to each framework; and each framework must then decide which to accept. Mesos is less feature-rich than Kubernetes and may involve extra integration work – defining services or batch jobs for Mesos is programmatic while it is declarative for Kubernetes.

There is currently a project to run Kubernetes as a Mesos framework. Mesos provides the fine grained resource allocation of Kubernetes pods across the nodes in a cluster. Kubernetes adds the higher level functions such as: load balancing; high availability through failover (rescheduling); and elastic scaling.

Mesos is particularly suited to environments where the application needs to be co-located with other services such as Hadoop, Kafka, and Spark. Mesos is also the foundation for a number of distributed systems such as:

- Apache Aurora – a highly scalable service scheduler for long-running services and `cron` jobs; it's used by Twitter. Aurora extends Mesos by adding rolling updates, service registration, and resource quotas.
- Chronos – a fault tolerant service scheduler, to be used as a replacement for `cron`, to orchestrate scheduled jobs within Mesos.
- Marathon – a simple to use service scheduler; it builds upon Mesos and Chronos by ensuring that two Chronos instances are running.

Choosing an Orchestration Framework

Each orchestration platform has advantages relative to the others and so users should evaluate which are best suited to their needs. Aspects to consider include:

- Does your enterprise have an existing DevOps framework that the orchestration must fit within and what APIs does it require?
- How many hosts will be used? Mesos is proven to work over thousands of physical machines.
- Will the containers be run on bare metal, private VMs, or in the cloud? Kubernetes is widely used in cloud deployments.
- Are there requirements for automated high availability? A Kubernetes Replication Controller will automatically reschedule failed pods/containers; Mesos considers this the role of an application's framework code.
- Is grouping and load balancing required for services? Kubernetes provides this but Mesos considers it a responsibility of the application's framework code.
- What skills do you have within your organization? Mesos typically requires custom coding to allow your application to run as a framework; Kubernetes is more declarative.
- Setting up the infrastructure to run containers is simple but the same is not true for some of orchestration frameworks – including Kubernetes and Mesos. Consider using hosted services such as Google Container Engine for Kubernetes, particularly for proofs of concept.

Security Considerations

While many of the concerns when using containers are common to bare metal deployments, containers provide an opportunity to improve levels of security if used properly. Because containers are so lightweight and easy to use, it's easy to deploy them for very specific purposes, and the container technology helps ensure that only the minimum required capabilities are exposed.

Within a container, the ability for malicious or buggy software to cause harm can be reduced by using resource isolation and rationing.

It's important to ensure that the container images are regularly scanned for vulnerabilities, and that the images are digitally signed. There are now many projects that provide scripts and scanning tools that can check if images and packages are up to date and free of security defects. Note that updating the images has no impact on existing containers; fortunately, Kubernetes and Aurora have the ability to perform rolling updates of containers.

Considerations for MongoDB

Running MongoDB with containers and orchestration introduces some additional considerations:

- MongoDB database nodes are stateful. In the event that a container fails, and is rescheduled, it's undesirable for the data to be lost (it could be recovered from other nodes in the replica set, but that takes time). To solve this, features such as the *Volume* abstraction in Kubernetes can be used to map what would otherwise be an ephemeral MongoDB data directory in the container to a persistent location where the data survives container failure and rescheduling.
- MongoDB database nodes within a replica set must communicate with each other – including after rescheduling. All of the nodes within a replica set must know the addresses of all of their peers, but when a container is rescheduled, it is likely to be restarted with a different IP Address. For example, all containers within a Kubernetes Pod share a single IP address, which changes when the pod is rescheduled. With Kubernetes, this can be handled by associating a Kubernetes Service with each MongoDB node, which uses the Kubernetes DNS service to provide a `hostname` for the service that remains constant through rescheduling.
- Once each of the individual MongoDB nodes is running (each within its own container), the replica set must be initialized and each node added. This is likely to require some additional logic beyond that offered by off the shelf orchestration tools. Specifically, one MongoDB

node within the intended replica set must be used to execute the `rs.initiate` and `rs.add` commands.

- If the orchestration framework provides automated rescheduling of containers (as Kubernetes does, for instance) then this can increase MongoDB's resiliency as a failed replica set member can be automatically recreated, restoring full redundancy levels without human intervention.
- It should be noted that while the orchestration framework might monitor the state of the containers, it is unlikely to monitor the applications running within the containers, or backup their data. That means it's important to use a strong monitoring and backup solution such as MongoDB Cloud Manager, included with MongoDB Enterprise Advanced and MongoDB Professional. Consider creating your own image that contains both your preferred version of MongoDB and the MongoDB Automation Agent.

Implementing a MongoDB Replica Set using Docker and Kubernetes

As described in the previous section, distributed databases such as MongoDB require a little extra attention when being deployed with orchestration frameworks such as Kubernetes. This section goes to the next level of detail, showing how this can actually be implemented. For simplicity, the Google Container Engine (GCE) cloud environment is used but most of the information holds true if you deploy your own Kubernetes infrastructure.

This section starts by creating the entire MongoDB replica set on a single GCE cluster, meaning that all members of the replica set will be within a single availability zone. This clearly doesn't provide geographic redundancy. In reality, little has to be changed to run across multiple clusters and those steps are described in the "Multiple Availability Zone MongoDB Replica Set" section.

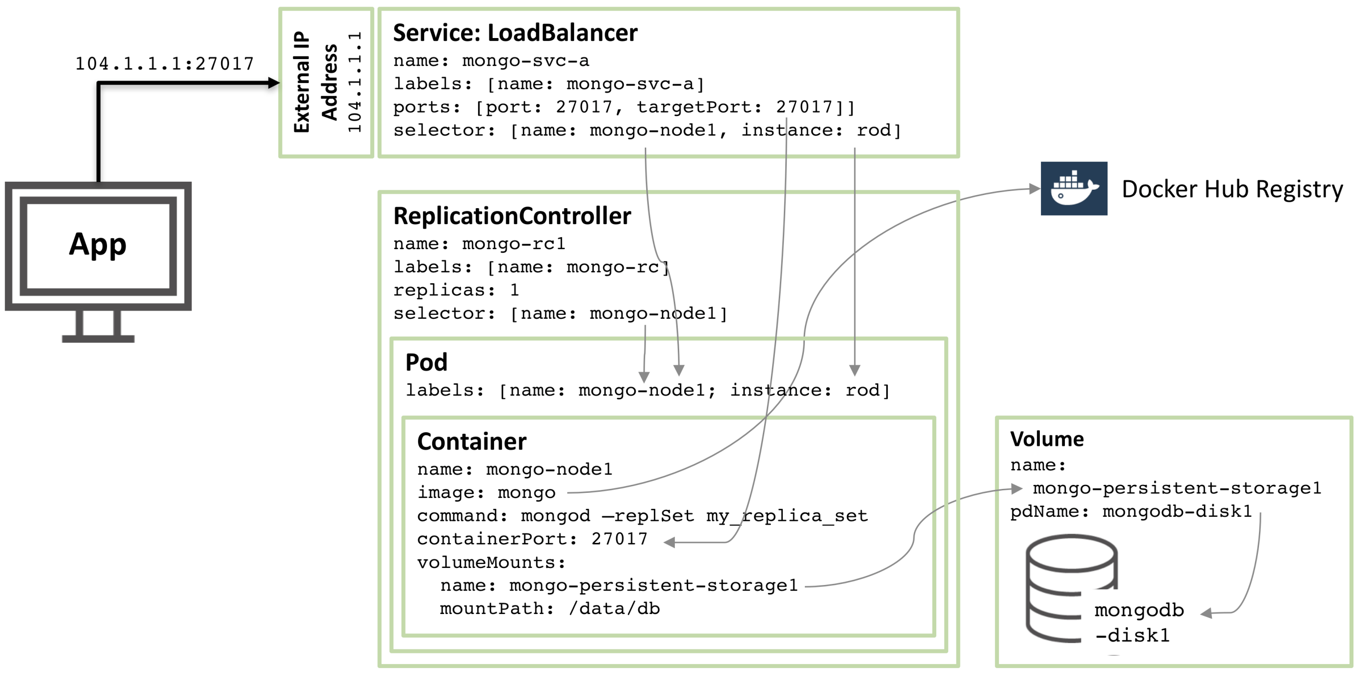


Figure 1: Pod for a Single Replica Set member

Each member of the replica set will be run as its own pod with a service exposing an external IP address and port. This 'fixed' IP addressed is important as both external applications and the other replica set members can rely on it remaining constant in the event that a pod is rescheduled.

Figure 1 illustrates one of these pods and the associated Replication Controller and service:

- Starting at the core there is a single container named `mongo-node1`. `mongo-node1` includes an image called `mongo` which is a publicly available MongoDB container image hosted on Docker Hub. The container exposes port `27107` within the cluster.
- The Kubernetes *volumes* feature is used to map the `/data/db` directory within the connector to the persistent storage element named `mongo-persistent-storage1`; which in turn is mapped to a disk named `mongodb-disk1` created in the Google Cloud. This is where MongoDB would store its data so that it is persisted over container rescheduling.
- The container is held within a pod which has the labels to name the pod `mongo-node` and provide an (arbitrary) instance name of `rod`.
- A Replication Controller named `mongo-rc1` is configured to ensure that a single instance of the `mongo-node1` pod is always running.
- The `LoadBalancer` service named `mongo-svc-a` exposes an IP Address to the outside world together with the port of `27017` which is mapped to the same port number in the container. The service identifies the correct pod using a selector that matches the pod's labels. That external IP Address and port will be used by both an application and for communication between the replica set members. There are also local IP addresses for each container, but those change when containers are moved or restarted, and so aren't of use for the replica set.

The configuration in Figure 1 can be described in a Kubernetes YAML file (this will later be extended to include the other two members of the replica set):

```
apiVersion: v1
kind: Service
metadata:
  name: mongo-svc-a
  labels:
    name: mongo-svc-a

spec:
  type: LoadBalancer
  ports:
  - port: 27017
    targetPort: 27017
    protocol: TCP
    name: mongo-svc-a
  selector:
    name: mongo-node1
    instance: rod
---
apiVersion: v1
kind: ReplicationController
metadata:
  name: mongo-rc1
  labels:
    name: mongo-rc
spec:
  replicas: 1
  selector:
    name: mongo-node1
  template:
    metadata:
      labels:
        name: mongo-node1
        instance: rod
    spec:
      containers:
      - name: mongo-node
        image: mongo
        ports:
        - containerPort: 27017
        volumeMounts:
        - name: mongo-persistent-storage1
          mountPath: /data/db
      volumes:
      - name: mongo-persistent-storage1
        gcePersistentDisk:
          pdName: mongodb-disk1
          fsType: ext4
```

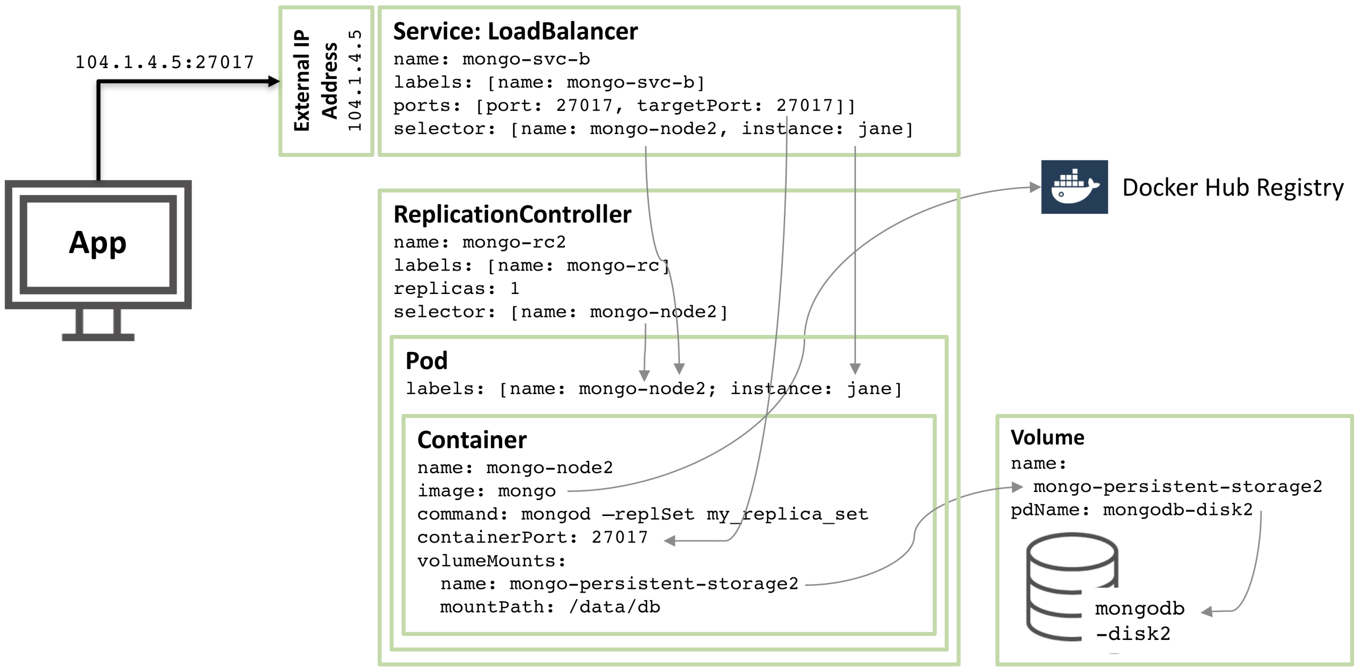


Figure 2: Pod for the second Replica Set member

Figure 2 shows the configuration for a second member of the replica set. 90% of the configuration is the same, with just these changes:

- The disk and volume names must be unique and so `mongodb-disk2` and `mongo-persistent-storage2` are used
- The Pod is assigned a label of `instance: jane` and `name: mongo-node2` so that the new service can distinguish it (using a selector) from the `rod` Pod used in Figure 1.
- The Replication Controller is named `mongo-rc2`
- The Service is named `mongo-svc-b` and gets a unique, external IP Address (in this instance, Kubernetes has assigned `104.1.5:2701`)

The configuration of the third replica set member follows the same pattern.

The final change required to the Kubernetes YAML file is to override the default command that will be run when each container is started (the `mongo` image executes the `mongod` binary unless instructed otherwise). This is achieved by adding a `command` attribute to the container definitions of `mongod --replSet my_replica_set`.

This is the final Kubernetes YAML file for the full MongoDB replica set:

```
apiVersion: v1
kind: Service
metadata:
  name: mongo-svc-a
  labels:
    name: mongo-svc-a
spec:
  type: LoadBalancer
  ports:
  - port: 27017
    targetPort: 27017
    protocol: TCP
    name: mongo-svc-a
  selector:
    name: mongo-node1
    instance: rod
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: mongo-svc-b
  labels:
    name: mongo-svc-b
spec:
  type: LoadBalancer
  ports:
  - port: 27017
    targetPort: 27017
    protocol: TCP
    name: mongo-svc-b
  selector:
    name: mongo-node2
    instance: jane
---
apiVersion: v1
kind: Service
metadata:
  name: mongo-svc-c
  labels:
    name: mongo-svc-c
spec:
  type: LoadBalancer
  ports:
  - port: 27017
    targetPort: 27017
    protocol: TCP
    name: mongo-svc-c
  selector:
    name: mongo-node3
    instance: freddy
---
apiVersion: v1
kind: ReplicationController
metadata:
  name: mongo-rc1
  labels:
    name: mongo-rc
spec:
  replicas: 1
  selector:
    name: mongo-node1
  template:
    metadata:
      labels:
        name: mongo-node1
        instance: rod
    spec:
      containers:
      - name: mongo-node1
        image: mongo
        command:
         - mongod
         - "--replSet"
         - my_replica_set
        ports:
        - containerPort: 27017
        volumeMounts:
        - name: mongo-persistent-storage1
          mountPath: /data/db
      volumes:
      - name: mongo-persistent-storage1
        gcePersistentDisk:
          pdName: mongodb-disk1
          fsType: ext4
---
apiVersion: v1
kind: ReplicationController
metadata:
  name: mongo-rc2
  labels:
    name: mongo-rc
spec:
  replicas: 1
  selector:
    name: mongo-node2
  template:
    metadata:
      labels:
        name: mongo-node2
        instance: jane
    spec:
      containers:
      - name: mongo-node2
        image: mongo
        command:
         - mongod
         - "--replSet"
         - my_replica_set
        ports:
        - containerPort: 27017
        volumeMounts:
        - name: mongo-persistent-storage2
          mountPath: /data/db
      volumes:
      - name: mongo-persistent-storage2
        gcePersistentDisk:
          pdName: mongodb-disk2
          fsType: ext4
---
apiVersion: v1
kind: ReplicationController
metadata:
  name: mongo-rc3
  labels:
    name: mongo-rc
spec:
  replicas: 1
  selector:
    name: mongo-node3
  template:
    metadata:
      labels:
        name: mongo-node3
        instance: freddy
    spec:
      containers:
      - name: mongo-node3
        image: mongo
        command:
         - mongod
         - "--replSet"
         - my_replica_set
        ports:
        - containerPort: 27017
        volumeMounts:
        - name: mongo-persistent-storage3
      mountPath: /data/db
    volumes:
    - name: mongo-persistent-storage3
      gcePersistentDisk:
        pdName: mongodb-disk3
        fsType: ext4
```

Figure 3 shows the full target configuration.

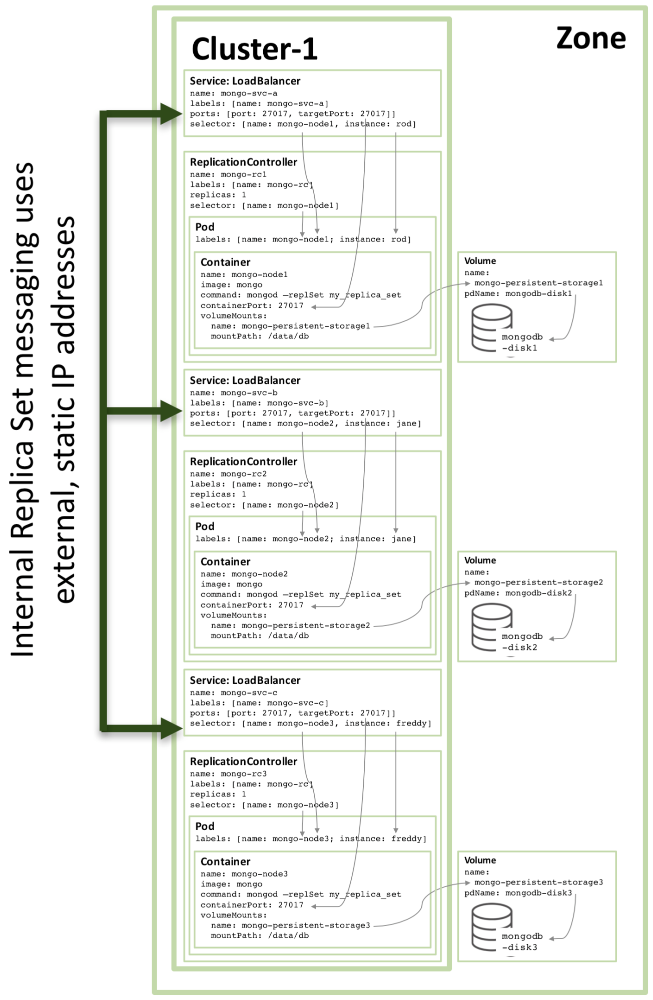

Internal Replica Set messaging uses external, static IP addresses
Zone
Cluster-1
Service: LoadBalancer
name: mongo-svc-a
labels: [name: mongo-svc-a]
ports: [port: 27017, targetPort: 27017]]
selector: [name: mongo-node1, instance: rod]
ReplicationController
name: mongo-rc1
labels: [name: mongo-rc]
replicas: 1
selector: [name: mongo-node1]
Pod
labels: [name: mongo-node1; instance: rod]
Container
name: mongo-node1
image: mongo
command: mongod —replSet my_replica_set
containerPort: 27017
volumeMounts:
name: mongo-persistent-storage1
mountPath: /data/db
Volume
name:
mongo-persistent-storage1
pdName: mongodb-disk1
mongodb -disk1
Service: LoadBalancer
name: mongo-svc-b
labels: [name: mongo-svc-b]
ports: [port: 27017, targetPort: 27017]]
selector: [name: mongo-node2, instance: jane]
ReplicationController
name: mongo-rc2
labels: [name: mongo-rc]
replicas: 1
selector: [name: mongo-node2]
Pod
labels: [name: mongo-node2; instance: jane]
Container
name: mongo-node2
image: mongo
command: mongod —replSet my_replica_set
containerPort: 27017
volumeMounts:
name: mongo-persistent-storage2
mountPath: /data/db
Volume
name:
mongo-persistent-storage2
pdName: mongodb-disk2
mongodb -disk2
Service: LoadBalancer
name: mongo-svc-c
labels: [name: mongo-svc-c]
ports: [port: 27017, targetPort: 27017]]
selector: [name: mongo-node3, instance: freddy]
ReplicationController
name: mongo-rc3
labels: [name: mongo-rc]
replicas: 1
selector: [name: mongo-node3]
Pod
labels: [name: mongo-node3; instance: freddy]
Container
name: mongo-node3
image: mongo
command: mongod —replSet my_replica_set
containerPort: 27017
volumeMounts:
name: mongo-persistent-storage3
mountPath: /data/db
Volume
name:
mongo-persistent-storage3
pdName: mongodb-disk3
mongodb -disk3

Figure 3: Complete Replica Set Deployment

In order to actually create the replica set, the first step is to create a GCE cluster of 3 machines, and 3 persistent disks using the `gcloud` command:

```
gcloud container clusters create cluster1

gcloud compute disks create mongodb-disk1
gcloud compute disks create mongodb-disk2
gcloud compute disks create mongodb-disk3
```

It is then simply a matter of using the `kubectl` command to process the Kubernetes YAML file:

```
kubectl create -f mongo-rs.yaml
```

At this point, each of the `mongod` processes is running it its own pod/container but there is a final step to form them into an active replica set. Before continuing, it is necessary to identify the external IP addresses of the three new services using the command `kubectl get svc`.

The `mongo` client is then used to access any one of the services to initiate the replica set and add the remaining members:

```
mongo --host 104.155.100.188
rs.initiate()
conf=rs.conf()
conf.members[0].host="104.155.100.188:27017"
rs.reconfig(conf)
rs.add("104.155.91.88")
rs.add("104.155.93.219")
```

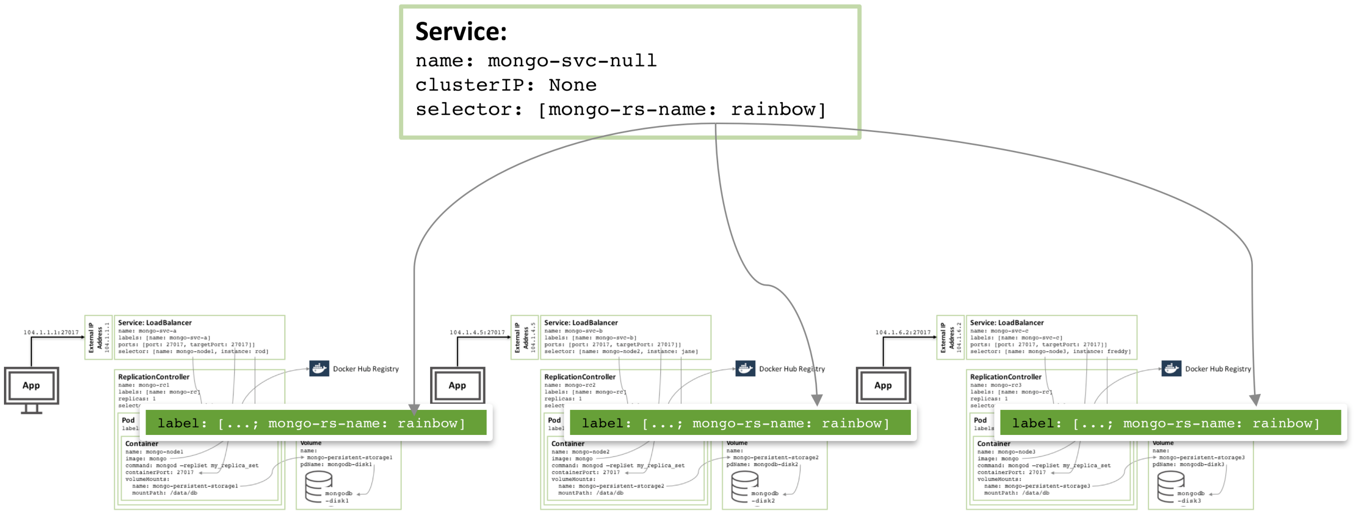
An application may now connect to the replica set as normal by including all three external IP addresses in its connect string; the MongoDB connector logic will ensure that writes and queries are directed to the correct pod.

All resources other than the disks can be removed by executing `kubectl delete -f mongo-rs.yaml`.

Note that even if running the configuration shown in Figure 3 on a Kubernetes cluster of three or more nodes then Kubernetes may (and often will) schedule two or more MongoDB replica set members on the same host. This is because Kubernetes view this as three independent services.

To increase redundancy (within the zone), an additional *headless* service can be created. The new service provides no capabilities to the outside world (and will not even have an IP address) but it serves to inform Kubernetes that the three MongoDB pods form a service and so Kubernetes will attempt to scheduled them on different nodes.

Figure 4 illustrates the addition of the extra service (`mongo-svc-null`) which associates itself with the three MongoDB pods using a selector of `mongo-rs-name: rainbow` which matches with a new label added to each of the pod definitions for the MongoDB replica set members.



```
Service:
name: mongo-svc-null
clusterIP: None
selector: [mongo-rs-name: rainbow]
```

```
Service: LoadBalancer
name: mongo-svc-a
labels: [name: mongo-svc-a]
ports: [port: 27017, targetPort: 27017]]
selector: [name: mongo-node1, instance: rod]

ReplicationController
name: mongo-rc1
labels: [name: mongo-rc]
replicas: 1

Pod
label: [...; mongo-rs-name: rainbow]

Container
name: mongo-node1
image: mongo
command: mongod —replSet my_replica_set
containerPort: 27017
volumeMounts:
  name: mongo-persistent-storage1
  mountPath: /data/db

Volume
name:
 mongo-persistent-storage1
pdName: mongodb-disk1
```

```
Service: LoadBalancer
name: mongo-svc-b
labels: [name: mongo-svc-b]
ports: [port: 27017, targetPort: 27017]]
selector: [name: mongo-node2, instance: jane]

ReplicationController
name: mongo-rc2
labels: [name: mongo-rc]
replicas: 1

Pod
label: [...; mongo-rs-name: rainbow]

Container
name: mongo-node2
image: mongo
command: mongod —replSet my_replica_set
containerPort: 27017
volumeMounts:
  name: mongo-persistent-storage2
  mountPath: /data/db

Volume
name:
 mongo-persistent-storage2
pdName: mongodb-disk2
```

```
Service: LoadBalancer
name: mongo-svc-c
labels: [name: mongo-svc-c]
ports: [port: 27017, targetPort: 27017]]
selector: [name: mongo-node3, instance: freddy]

ReplicationController
name: mongo-rc3
labels: [name: mongo-rc]
replicas: 1

Pod
label: [...; mongo-rs-name: rainbow]

Container
name: mongo-node3
image: mongo
command: mongod —replSet my_replica_set
containerPort: 27017
volumeMounts:
  name: mongo-persistent-storage3
  mountPath: /data/db

Volume
name:
 mongo-persistent-storage3
pdName: mongodb-disk3
```

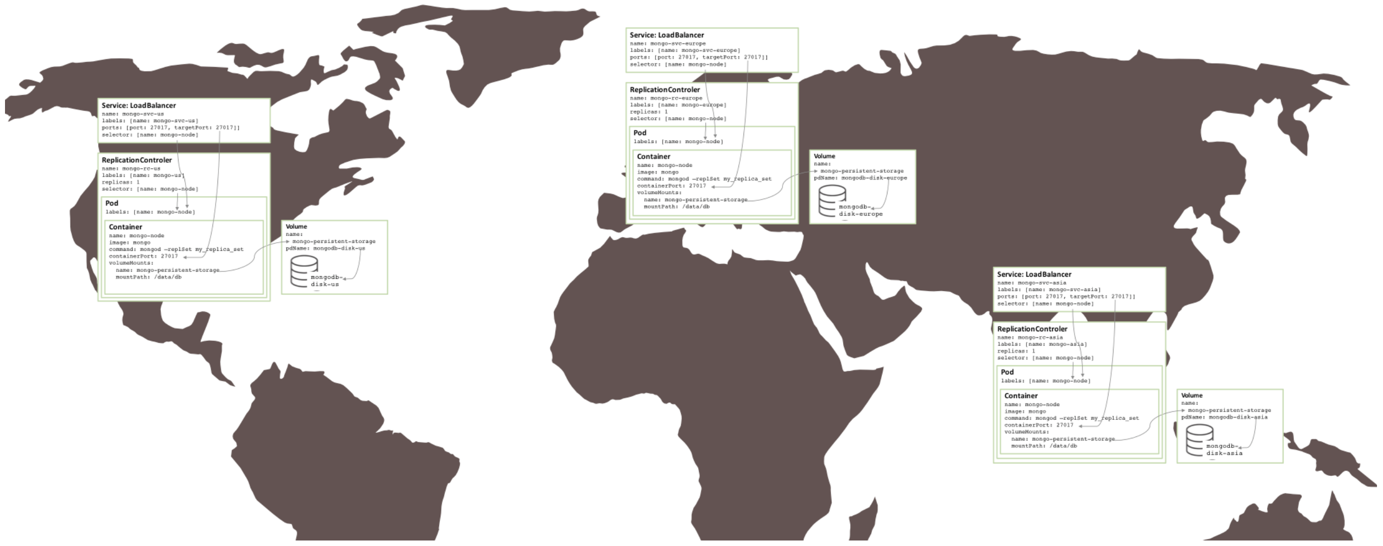
Figure 4: Headless service to avoid co-locating of MongoDB replica set members

Multiple Availability Zone MongoDB Replica Set

There is risk associated with the replica set created above in that everything is running in the same GCE cluster, and hence in the same availability zone. If there were a major incident that took the availability zone offline, then the MongoDB replica set would be unavailable. If geographic redundancy is required then the three pods should be run in three different availability zones or regions.

Surprisingly little needs to change in order to create a similar replica set that is split between three zones – which requires three clusters. Each cluster requires its own Kubernetes YAML file that defines just the pod, Replication Controller and service for one member of the replica set. It is then a simple matter to create a cluster, persistent storage, and MongoDB node for each zone, e.g.:

```
gcloud container clusters create "europe-1" \
        --zone "europe-west1-c" --num-nodes 1
gcloud compute disks create --size=200GB \
        --zone="europe-west1-c" mongodb-disk-europe
kubectl create -f mongo-europe.yaml
```



```
Service: LoadBalancer
name: mongo-svc-us
labels: [name: mongo-svc-us]
ports: [port: 27017, targetPort: 27017]]
selector: [name: mongo-node]

ReplicationControler
name: mongo-rc-us
labels: [name: mongo-us]
replicas: 1
selector: [name: mongo-node]

Pod
labels: [name: mongo-node]

Container
name: mongo-node
image: mongo
command: mongod —replSet my_replica_set
containerPort: 27017
volumeMounts:
  name: mongo-persistent-storage
  mountPath: /data/db

Volume
name:
 mongo-persistent-storage
pdName: mongodb-disk-us
```

```
Service: LoadBalancer
name: mongo-svc-europe
labels: [name: mongo-svc-europe]
ports: [port: 27017, targetPort: 27017]]
selector: [name: mongo-node]

ReplicationControler
name: mongo-rc-europe
labels: [name: mongo-europe]
replicas: 1
selector: [name: mongo-node]

Pod
labels: [name: mongo-node]

Container
name: mongo-node
image: mongo
command: mongod —replSet my_replica_set
containerPort: 27017
volumeMounts:
  name: mongo-persistent-storage
  mountPath: /data/db

Volume
name:
 mongo-persistent-storage
pdName: mongodb-disk-europe
```

```
Service: LoadBalancer
name: mongo-svc-asia
labels: [name: mongo-svc-asia]
ports: [port: 27017, targetPort: 27017]]
selector: [name: mongo-node]

ReplicationControler
name: mongo-rc-asia
labels: [name: mongo-asia]
replicas: 1
selector: [name: mongo-node]

Pod
labels: [name: mongo-node]

Container
name: mongo-node
image: mongo
command: mongod —replSet my_replica_set
containerPort: 27017
volumeMounts:
  name: mongo-persistent-storage
  mountPath: /data/db

Volume
name:
 mongo-persistent-storage
pdName: mongodb-disk-asia
```

Figure 5: Replica set running over multiple availability zones

MongoDB and Containers in the Real World

fuboTV provide a soccer streaming service in North America and they run their full stack (including MongoDB) on Docker and Kubernetes; find out the benefits they see from this and how it's achieved in this case study.

Square Enix is one of the world's leading providers of gaming experiences, publishing such iconic titles as *Tomb Raider* and *Final Fantasy*. They have produced an internal multi-tenant database-as-a-service using MongoDB and Docker – find out more in this case study.

Comparethemarket.com is a one of the UK's leading providers for price comparison services and uses MongoDB as the operational database behind its large microservice environment. Service uptime is critical, and MongoDB's distributed design is key to ensure that SLA's are always met. Comparethemarket.com's deployment consists of microservices deployed in AWS. Each microservice, or logical grouping of related microservices, is provisioned with its own MongoDB replica set running in Docker containers, and deployed across multiple AWS Availability Zones to provide resiliency and high availability. MongoDB Ops Manager is used to provide the operational automation that is essential to launch new features quickly: deploying replica sets, providing continuous backups, and performing zero downtime upgrades.

We Can Help

We are the MongoDB experts. Over 2,000 organizations rely on our commercial products, including startups and more than a third of the Fortune 100. We offer software and services to make your life easier:

MongoDB Enterprise Advanced is the best way to run MongoDB in your data center. It's a finely-tuned package of advanced software, support, certifications, and other services designed for the way you do business.

MongoDB Cloud Manager is the easiest way to run MongoDB in the cloud. It makes MongoDB the system you worry about the least and like managing the most.

MongoDB Professional helps you manage your deployment and keep it running smoothly. It includes support from MongoDB engineers, as well as access to MongoDB Cloud Manager.

Development Support helps you get up and running quickly. It gives you a complete package of software and services for the early stages of your project.

MongoDB Consulting packages get you to production faster, help you tune performance in production, help you scale, and free you up to focus on your next release.

MongoDB Training helps you become a MongoDB expert, from design to operating mission-critical systems at scale. Whether you're a developer, DBA, or architect, we can make you better at MongoDB.

Resources

For more information, please visit mongodb.com or contact us at sales@mongodb.com.

Case Studies (mongodb.com/customers)
Presentations (mongodb.com/presentations)
Free Online Training (university.mongodb.com)
Webinars and Events (mongodb.com/events)
Documentation (docs.mongodb.org)
MongoDB Enterprise Download (mongodb.com/download)

mongoDB

New York • Palo Alto • Washington, D.C. • London • Dublin • Barcelona • Sydney • Tel Aviv
US 866-237-8815 • INTL +1-650-440-4474 • info@mongodb.com