Chapter 13. Integrating Storage Solutions and Kubernetes

In many cases decoupling state from applications and building your microservices to be as stateless as possible results in maximally reliable, manageable systems.

However, nearly every system that has any complexity has state in the system somewhere, from the records in a database to the index shards that serve results for a web search engine. At some point you have to have data stored somewhere.

Integrating this data with containers and container orchestration solutions is often the most complicated aspect of building a distributed system. This complexity largely stems from the fact that the move to containerized architectures is also a move toward decoupled, immutable, and declarative application development. These patterns are relatively easy to apply to stateless web applications, but even “cloud-native” storage solutions like Cassandra or MongoDB involve some sort of manual or imperative steps to set up a reliable, replicated solution.

As an example of this, consider setting up a ReplicaSet in MongoDB, which involves deploying the Mongo daemon and then running an imperative command to identify the leader, as well as the participants in the Mongo cluster. Of course, these steps can be scripted, but in a containerized world it is difficult to see how to integrate such commands into a deployment. Likewise, even getting DNS-resolvable names for individual containers in a replicated set of containers is challenging.

Additional complexity comes from the fact that there is data gravity. Most containerized systems aren’t built in a vacuum; they are usually adapted from existing systems deployed onto VMs, and these systems likely include data that has to be imported or migrated.

Finally, evolution to the cloud means that many times storage is actually an externalized cloud service, and in that context it can never really exist inside of the Kubernetes cluster.

This chapter covers a variety of approaches for integrating storage into containerized microservices in Kubernetes. First, we cover how to import existing external storage solutions (either cloud services or running on VMs) into Kubernetes. Next, we explore how to run reliable singletons inside of Kubernetes that enable you to have an environment that largely matches the VMs where you previously deployed storage solutions. Finally we cover StatefulSets, which are still under development but represent the future of stateful workloads in Kubernetes.

Importing External Services

In many cases, you have an existing machine running in your network that has some sort of database running on it. In this situation you may not want to immediately move that database into containers and Kubernetes. Perhaps it is run by a different team, or you are doing a gradual move, or the task of migrating the data is simply more trouble than it's worth.

Regardless of the reasons for staying put, this legacy server and service are not going to move into Kubernetes, but nonetheless it is still worthwhile to represent this server in Kubernetes. When you do this, you get to take advantage of all of the built-in naming and service discovery primitives provided by Kubernetes. Additionally, this enables you to configure all your applications so that it looks like the database that is running on a machine somewhere is actually a Kubernetes service. This means that it is trivial to replace it with a database that is a Kubernetes service. For example, in production, you may rely on your legacy database that is running on a machine, but for continuous testing you may deploy a test database as a transient container. Since it is created and destroyed for each test run, data persistence isn't important in the continuous testing case. Representing both databases as Kubernetes services enables you to maintain identical configurations in both testing and production. High fidelity between test and production ensures that passing tests will lead to successful deployment in production.

To see concretely how you maintain high fidelity between development and production, remember that all Kubernetes objects are deployed into *namespaces*. Imagine that we have `test` and `product` namespaces defined. The test service is imported using an object like:

```
kind: Service
metadata:
  name: my-database
  # note 'test' namespace here
  namespace: test
...
```

The production service looks the same, except it uses a different namespace:

```
kind: Service
```

```
metadata:
  name: my-database
  # note 'prod' namespace here
  namespace: prod
...
```

When you deploy a Pod into the `test` namespace and it looks up the service named `my-database`, it will receive a pointer to `my-database.test.svc.cluster.internal`, which in turn points to the test database. In contrast, when a Pod deployed in the `prod` namespace looks up the same name (`my-database`) it will receive a pointer to `my-database.prod.svc.cluster.internal`, which is the production database. Thus, the same service name, in two different namespaces, resolves to two different services. For more details on how this works, see Chapter 7.

> **NOTE**
>
> The following techniques all use database or other storage services, but these approaches can be used equally well with other services that aren't running inside your Kubernetes cluster.

Services Without Selectors

When we first introduced services, we talked at length about label queries and how they were used to identify the dynamic set of Pods that were the backends for a particular service. With external services, however, there is no such label query. Instead, you generally have a DNS name that points to the specific server running the database. For our example, let's assume that this server is named `database.company.com`. To import this external database service into Kubernetes, we start by creating a service without a Pod selector that references the DNS name of the database server (Example 13-1).

Example 13-1. dns-service.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: external-database
spec:
  type: ExternalName
  externalName: "database.company.com
```

When a typical Kubernetes service is created, an IP address is also created and the Kubernetes DNS service is populated with an A record that points to that IP address. When you create a service of type `ExternalName`, the Kubernetes DNS service is instead populated with a CNAME record that points to the external name you specified (`database.company.com` in this case). When an application in the cluster does a DNS lookup for the hostname `external-database.svc.default.cluster`, the DNS protocol aliases that name to "database.company.com." This then resolves to the IP address of your external database server. In this way, all containers in Kubernetes believe that they are talking to a service that is backed with other containers, when in fact they are being redirected to the external database.

Note that this is not restricted to databases you are running on your own infrastructure. Many cloud databases and other services provide you with a DNS name to use when accessing the database (e.g., `my-database.databases.cloudprovider.com`). You can use this DNS name as the `externalName`. This imports the cloud-provided database into the namespace of your Kubernetes cluster.

Sometimes, however, you don't have a DNS address for an external database service, just an IP address. In such cases, it is still possible to import this server as a Kubernetes service, but the operation is a little different. First, you create a `Service` without a label selector, but also without the `ExternalName` type we used before (Example 13-2).

Example 13-2. external-ip-service.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: external-ip-database
```

At this point, Kubernetes will allocate a virtual IP address for this service and populate an A record for it. However, because there is no selector for the service, there will be no endpoints populated for the load balancer to redirect traffic to.

Given that this is an external service, the user is responsible for populating the endpoints manually with an `Endpoints` resource (Example 13-3).

Example 13-3. external-ip-endpoints.yaml

```
kind: Endpoints
apiVersion: v1
metadata:
  name: external-ip-database
subsets:
  - addresses:
    - ip: 192.168.0.1
    ports:
    - port: 3306
```

If you have more than one IP address for redundancy, you can repeat them in the `addresses` array. Once the endpoints are populated, the load balancer will start redirecting traffic from your Kubernetes service to the IP address endpoint(s).

> **NOTE**
>
> Because the user has assumed responsibility for keeping the IP address of the server up to date, you need to either ensure that it never changes or make sure that some automated process updates the `Endpoints` record.

Limitations of External Services: Health Checking

External services in Kubernetes have one significant restriction: they do not perform any health checking. The user is responsible for ensuring that the endpoint or DNS name supplied to Kubernetes is as reliable as necessary for the application.

Running Reliable Singletons

The challenge of running storage solutions in Kubernetes is often that primitives like ReplicaSet expect that every container is identical and replaceable, but for most storage solutions this isn't the case. One option to address this is to use Kubernetes primitives, but not attempt to replicate the storage. Instead, simply run a single Pod that runs the database or other storage solution. In this way the challenges of running replicated storage in Kubernetes don't occur, since there is no replication.

At first blush, this might seem to run counter to the principles of building reliable distributed systems, but in general, it is no less reliable than running your database or storage infrastructure on a single virtual or physical machine, which is how many people currently have built their systems. Indeed, in reality, if you structure the system properly the only thing you are sacrificing is potential downtime for upgrades or in case of machine failure. While for large-scale or mission-critical systems this may not be acceptable, for many smaller-scale applications this kind of limited downtime is a reasonable trade-off for the reduced complexity. If this is not true for you, feel free to skip this section and either import existing services as described in the previous section, or move on to Kubernetes-native `StatefulSets`, described in the following section. For everyone else, we'll review how to build reliable singletons for data storage.

Running a MySQL Singleton

In this section, we'll describe how to run a reliable singleton instance of the MySQL database as a Pod in Kubernetes, and how to expose that singleton to other applications in the cluster.

To do this, we are going to create three basic objects:

- A persistent volume to manage the lifespan of the on-disk storage independently from the lifespan of the running MySQL application
- A MySQL Pod that will run the MySQL application
- A service that will expose this Pod to other containers in the cluster

In Chapter 5 we described persistent volumes, but a quick review makes sense. A persistent volume is a storage location that has a lifetime independent of any Pod or container. This is very useful in the case of persistent storage solutions where the on-disk representation of a database should survive even if the containers running the database application crash, or move to different machines. If the application moves to a different machine, the volume should move with it, and data should be preserved. Separating the data storage out as a persistent volume makes this possible. To begin, we'll create a persistent volume for our MySQL database to use.

This example uses NFS for maximum portability, but Kubernetes supports many different persistent volume drive types. For example, there are persistent volume drivers for all major public cloud providers, as well as many private cloud providers. To use these solutions, simply replace `nfs` with the appropriate cloud provider volume type (e.g., `azure`, `awsElasticBlockStore`, or `gcePersistentDisk`). In all cases, this change is all you need. Kubernetes knows how to create the appropriate storage disk in the respective cloud provider. This is a great example of how Kubernetes simplifies the development of reliable distributed systems.

Here's the example persistent volume object (Example 13-4).

Example 13-4. nfs-volume.yaml

```
apiVersion: v1
kind: PersistentVolume
```

```
metadata:
  name: database
  labels:
    volume: my-volume
spec:
  capacity:
    storage: 1Gi
  nfs:
    server: 192.168.0.1
    path: "/exports"
```

This defines an NFS persistent volume object with 1 GB of storage space.

We can create this persistent volume as usual with:

```
$ kubectl apply -f nfs-volume.yaml
```

Now that we have a persistent volume created, we need to claim that persistent volume for our Pod. We do this with a `PersistentVolumeClaim` object (Example 13-5).

Example 13-5. nfs-volume-claim.yaml

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: database
spec:
  resources:
    requests:
      storage: 1Gi
  selector:
    matchLabels:
      volume: my-volume
```

The `selector` field uses labels to find the matching volume we defined previously.

This kind of indirection may seem overly complicated, but it has a purpose — it serves to isolate our Pod definition from our storage definition. You can declare volumes directly inside a Pod specification, but this locks that Pod specification to a particular volume provider (e.g., a specific public or private cloud). By using volume claims, you can keep your Pod specifications cloud-agnostic; simply create different volumes, specific to the cloud, and use a `PersistentVolumeClaim` to bind them together.

Now that we've claimed our volume, we can use a ReplicaSet to construct our singleton Pod. It might seem odd that we are using a ReplicaSet to manage a single Pod, but it is necessary for reliability. Remember that once scheduled to a

machine, a bare Pod is bound to that machine forever. If the machine fails, then any Pods that are on that machine that are not being managed by a higher-level controller like a ReplicaSet vanish along with the machine and are not rescheduled elsewhere. Consequently, to ensure that our database Pod is rescheduled in the presence of machine failures, we use the higher-level ReplicaSet controller, with a replica size of one, to manage our database (Example 13-6).

Example 13-6. mysql-replicaset.yaml

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: mysql
  # labels so that we can bind a Service to this Pod
  labels:
    app: mysql
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: database
        image: mysql
        resources:
          requests:
            cpu: 1
            memory: 2Gi
        env:
        # Environment variables are not a best practice for security,
        # but we're using them here for brevity in the example.
        # See Chapter 11 for better options.
        - name: MYSQL_ROOT_PASSWORD
          value: some-password-here
        livenessProbe:
          tcpSocket:
            port: 3306
        ports:
        - containerPort: 3306
        volumeMounts:
          - name: database
            # /var/lib/mysql is where MySQL stores its databases
            mountPath: "/var/lib/mysql"
      volumes:
      - name: database
        persistentVolumeClaim:
          claimName: database
```

Once we create the `ReplicaSet` it will in turn create a Pod running MySQL using the persistent disk we originally created. The final step is to expose this as

a Kubernetes service (Example 13-7).

Example 13-7. mysql-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: mysql
spec:
  ports:
  - port: 3306
    protocol: TCP
  selector:
    app: mysql
```

Now we have a reliable singleton MySQL instance running in our cluster and exposed as a service named `mysql`, which we can access at the full domain name `mysql.svc.default.cluster`.

Similar instructions can be used for a variety of data stores, and if your needs are simple and you can survive limited downtime in the face of a machine failure or a need to upgrade the database software, a reliable singleton may be the right approach to storage for your application.

Dynamic Volume Provisioning

Many clusters also include *dynamic volume provisioning*. With dynamic volume provisioning, the cluster operator creates one or more `StorageClass` objects. Here's a default storage class that automatically provisions disk objects on the Microsoft Azure platform (Example 13-8).

Example 13-8. storageclass.yaml

```
apiVersion: storage.k8s.io/v1beta1
kind: StorageClass
metadata:
  name: default
  annotations:
    storageclass.beta.kubernetes.io/is-default-class: "true"
  labels:
    kubernetes.io/cluster-service: "true"
provisioner: kubernetes.io/azure-disk
```

Once a storage class has been created for a cluster, you can refer to this storage class in your persistent volume claim, rather than referring to any specific persistent volume. When the dynamic provisioner sees this storage claim, it uses the appropriate volume driver to create the volume and bind it to your persistent volume claim.

Here's an example of a `PersistentVolumeClaim` that uses the `default` storage class we just defined to claim a newly created persistent volume (Example 13-9).

Example 13-9. dynamic-volume-claim.yaml

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: my-claim
  annotations:
    volume.beta.kubernetes.io/storage-class: default
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

The `volume.beta.kubernetes.io/storage-class` annotation is what links this claim back up to the storage class we created.

Persistent volumes are great for traditional applications that require storage, but if you need to develop high-availability, scalable storage in a Kubernetes-native fashion, the newly released StatefulSet object can be used. With this in mind,

we’ll describe how to deploy MongoDB using StatefulSets in the next section.

Kubernetes-Native Storage with StatefulSets

When Kubernetes was first developed, there was a heavy emphasis on homogeneity for all replicas in a replicated set. In this design, no replica had an individual identity or configuration. It was up to the individual application developer to determine a design that could establish this identity for the application.

While this approach provides a great deal of isolation for the orchestration system, it also makes it quite difficult to develop stateful applications. After significant input from the community and a great deal of experimentation with various existing stateful applications, StatefulSets were introduced into Kubernetes in version 1.5.

> **NOTE**
>
> Because StatefulSets are a beta feature, it’s possible that the API will change before it becomes an official Kubernetes API. The StatefulSet API has had a lot of input and is generally considered fairly stable, but the beta status should be considered before taking on StatefulSets. In many cases the previously outlined patterns for stateful applications may serve you better in the near term.

Properties of StatefulSets

StatefulSets are replicated groups of Pods similar to ReplicaSets, but unlike a ReplicaSet, they have certain unique properties:

- Each replica gets a persistent hostname with a unique index (e.g., `database-0`, `database-1`, etc.).
- Each replica is created in order from lowest to highest index, and creation will block until the Pod at the previous index is healthy and available. This also applies to scaling up.
- When deleted, each replica will be deleted in order from highest to lowest. This also applies to scaling down the number of replicas.

Manually Replicated MongoDB with StatefulSets

In this section, we'll deploy a replicated MongoDB cluster. For now, the replication setup itself will be done manually to give you a feel for how StatefulSets work. Eventually we will automate this setup as well.

To start, we'll create a replicated set of three MongoDB Pods using a StatefulSet object (Example 13-10).

Example 13-10. mongo-simple.yaml

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: mongo
spec:
  serviceName: "mongo"
  replicas: 3
  template:
    metadata:
      labels:
        app: mongo
    spec:
      containers:
      - name: mongodb
        image: mongo:3.4.1
        command:
        - mongod
        - --replSet
        - rs0
        ports:
        - containerPort: 27017
          name: peer
```

As you can see, the definition is similar to the ReplicaSet definition from previous sections. The only changes are the `apiVersion` and `kind` fields. Create the StatefulSet:

```
$ kubectl apply -f mongo-simple.yaml
```

Once created, the differences between a ReplicaSet and a StatefulSet become apparent. Run `kubectl get pods` and you will likely see:

```
NAME      READY     STATUS              RESTARTS   AGE
mongo-0   1/1       Running             0          1m
mongo-1   0/1       ContainerCreating   0          10s
```

There are two important differences between this and what you would see with a ReplicaSet. The first is that each replicated Pod has a numeric index (`0`, `1`, …),

instead of the random suffix that is added by the ReplicaSet controller. The second is that the Pods are being slowly created in order, not all at once as they would be with a ReplicaSet.

Once the StatefulSet is created, we also need to create a “headless” service to manage the DNS entries for the StatefulSet. In Kubernetes a service is called “headless” if it doesn’t have a cluster virtual IP address. Since with StatefulSets each Pod has a unique identity, it doesn’t really make sense to have a load-balancing IP address for the replicated service. You can create a headless service using `clusterIP: None` in the service specification (Example 13-11).

Example 13-11. mongo-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: mongo
spec:
  ports:
  - port: 27017
    name: peer
  clusterIP: None
  selector:
    app: mongo
```

Once you create that service, there are usually four DNS entries that are populated. As usual, `mongo.default.svc.cluster.local` is created, but unlike with a standard service, doing a DNS lookup on this hostname provides all the addresses in the StatefulSet. In addition, entries are created for `mongo-0 .mongo .default .svc .cluster.local` as well as `mongo-1.mongo` and `mongo-2.mongo`. Each of these resolves to the specific IP address of the replica index in the StatefulSet. Thus, with StatefulSets you get well-defined, persistent names for each replica in the set. This is often very useful when you are configuring a replicated storage solution. You can see these DNS entries in action by running commands in one of the Mongo replicas:

```
$ kubectl exec mongo-0 bash ping mongo-1.mongo
```

Next, we’re going to manually set up Mongo replication using these per-Pod hostnames.

We’ll choose `mongo-0.mongo` to be our initial primary. Run the `mongo` tool in that Pod:

```
$ kubectl exec -it mongo-0 mongo
> rs.initiate( {
  _id: "rs0",
  members:[ { _id: 0, host: "mongo-0.mongo:27017" } ]
 });
 OK
```

This command tells `mongodb` to initiate the ReplicaSet `rs0` with `mongo-0.mongo` as the primary replica.

> **NOTE**
>
> The `rs0` name is arbitrary. You can use whatever you'd like, but you'll need to change it in the *mongo.yaml* StatefulSet definition as well.

Once you have initiated the Mongo ReplicaSet, you can add the remaining replicas by running the following commands in the `mongo` tool on the `mongo-0.mongo` Pod:

```
$ kubectl exec -it mongo-0 mongo
> rs.add("mongo-1.mongo:27017");
> rs.add("mongo-2.mongo:27017");
```

As you can see, we are using the replica-specific DNS names to add them as replicas in our Mongo cluster. At this point, we're done. Our replicated MongoDB is up and running. But it's really not as automated as we'd like it to be. In the next section, we'll see how to use scripts to automate the setup.

Automating MongoDB Cluster Creation

To automate the deployment of our StatefulSet-based MongoDB cluster, we're going to add an additional container to our Pods to perform the initialization.

To configure this Pod without having to build a new Docker image, we're going to use a ConfigMap to add a script into the existing MongoDB image. Here's the container we're adding:

```
...
      - name: init-mongo
        image: mongo:3.4.1
        command:
        - bash
        - /config/init.sh
        volumeMounts:
        - name: config
          mountPath: /config
      volumes:
      - name: config
        configMap:
          name: "mongo-init"
```

Note that it is mounting a ConfigMap volume whose name is `mongo-init`. This ConfigMap holds a script that performs our initialization. First, the script determines whether it is running on `mongo-0` or not. If it is on `mongo-0`, it creates the ReplicaSet using the same command we ran imperatively previously. If it is on a different Mongo replica, it waits until the ReplicaSet exists, and then it registers itself as a member of that ReplicaSet.

Example 13-12 has the complete ConfigMap object.

Example 13-12. mongo-configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mongo-init
data:
  init.sh: |
    #!/bin/bash

    # Need to wait for the readiness health check to pass so that the
    # mongo names resolve. This is kind of wonky.
    until ping -c 1 ${HOSTNAME}.mongo; do
      echo "waiting for DNS (${HOSTNAME}.mongo)..."
      sleep 2
    done

    until /usr/bin/mongo --eval 'printjson(db.serverStatus())'; do
      echo "connecting to local mongo..."
```

```
  sleep 2
done
echo "connected to local."

HOST=mongo-0.mongo:27017

until /usr/bin/mongo --host=${HOST} --eval 'printjson(db.serverStatus())'; do
  echo "connecting to remote mongo..."
  sleep 2
done
echo "connected to remote."

if [[ "${HOSTNAME}" != 'mongo-0' ]]; then
  until /usr/bin/mongo --host=${HOST} --eval="printjson(rs.status())" \
        | grep -v "no replset config has been received"; do
    echo "waiting for replication set initialization"
    sleep 2
  done
  echo "adding self to mongo-0"
  /usr/bin/mongo --host=${HOST} \
     --eval="printjson(rs.add('${HOSTNAME}.mongo'))"
fi

if [[ "${HOSTNAME}" == 'mongo-0' ]]; then
  echo "initializing replica set"
  /usr/bin/mongo --eval="printjson(rs.initiate(\
      {'_id': 'rs0', 'members': [{'_id': 0, \
       'host': 'mongo-0.mongo:27017'}]}))"
fi
echo "initialized"

while true; do
  sleep 3600
done
```

> **NOTE**
>
> This script currently sleeps forever after initializing the cluster. Every container in a Pod has to have the same `RestartPolicy`. Since we want our main Mongo container to be restarted, we need to have our initialization container run forever too, or else Kubernetes might think our Mongo Pod is unhealthy.

Putting it all together, here is the complete StatefulSet that uses the ConfigMap in Example 13-13.

Example 13-13. mongo.yaml

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: mongo
spec:
  serviceName: "mongo"
  replicas: 3
  template:
    metadata:
```

```
    labels:
      app: mongo
  spec:
    containers:
    - name: mongodb
      image: mongo:3.4.1
      command:
      - mongod
      - --replSet
      - rs0
      ports:
      - containerPort: 27017
        name: web
    # This container initializes the mongodb server, then sleeps.
    - name: init-mongo
      image: mongo:3.4.1
      command:
      - bash
      - /config/init.sh
      volumeMounts:
      - name: config
        mountPath: /config
    volumes:
    - name: config
      configMap:
        name: "mongo-init"
```

Given all of these files, you can create a Mongo cluster with:

```
$ kubectl apply -f mongo-config-map.yaml
$ kubectl apply -f mongo-service.yaml
$ kubectl apply -f mongo.yaml
```

Or if you want, you can combine them all into a single YAML file where the individual objects are separated by `---`. Ensure that you keep the same ordering, since the StatefulSet definition relies on the ConfigMap definition existing.

Persistent Volumes and StatefulSets

For persistent storage, you need to mount a persistent volume into the */data/db* directory. In the Pod template, you need to update it to mount a persistent volume claim to that directory:

```
...
        volumeMounts:
        - name: database
          mountPath: /data/db
```

While this approach is similar to the one we saw with reliable singletons, because the StatefulSet replicates more than one Pod you cannot simply reference a persistent volume claim. Instead, you need to add a *persistent volume claim template*. You can think of the claim template as being identical to the Pod template, but instead of creating Pods, it creates volume claims. You need to add the following onto the bottom of your StatefulSet definition:

```
volumeClaimTemplates:
- metadata:
    name: database
    annotations:
      volume.alpha.kubernetes.io/storage-class: anything
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 100Gi
```

When you add a volume claim template to a StatefulSet definition, each time the StatefulSet controller creates a Pod that is part of the StatefulSet it will create a persistent volume claim based on this template as part of that Pod.

> **NOTE**
>
> In order for these replicated persistent volumes to work correctly, you either need to have autoprovisioning set up for persistent volumes, or you need to prepopulate a collection of persistent volume objects for the StatefulSet controller to draw from. If there are no claims that can be created, the StatefulSet controller will not be able to create the corresponding Pods.

One Final Thing: Readiness Probes

The final piece in productionizing our MongoDB cluster is to add liveness checks to our Mongo-serving containers. As we learned in “Health Checks”, the liveness probe is used to determine if a container is operating correctly. For the liveness checks, we can use the `mongo` tool itself by adding the following to the Pod template in the StatefulSet object:

```
...
 livenessProbe:
   exec:
     command:
       - /usr/bin/mongo
       - --eval
       - db.serverStatus()
     initialDelaySeconds: 10
     timeoutSeconds: 10
 ...
```

Summary

Once we have combined StatefulSets, persistent volume claims, and liveness probing, we have a hardened, scalable cloud-native MongoDB installation running on Kubernetes. While this example dealt with MongoDB, the steps for creating StatefulSets to manage other storage solutions are quite similar and similar patterns can be followed.