Chapter 14. Deploying Real-World Applications

The previous chapters described a variety of API objects that are available in a Kubernetes cluster and ways in which those objects can best be used to construct reliable distributed systems. However, none of the preceding chapters really discussed how you might use the objects in practice to deploy a complete, real-world application. That is the focus of this chapter.

We'll take a look at three real-world applications:

- Parse, an open source API server for mobile applications
- Ghost, a blogging and content management platform
- Redis, a lightweight, performant key/value store

These complete examples should give you a better idea of how to structure your own deployments using Kubernetes.

Parse

The Parse server is a cloud API dedicated to providing easy-to-use storage for mobile applications. It provides a variety of different client libraries that make it easy to integrate with Android, iOS, and other mobile platforms. Parse was purchased by Facebook in 2013 and subsequently shut down. Fortunately for us, a compatible server was open sourced by the core Parse team and is available for us to use. This section describes how to set up Parse in Kubernetes.

Prerequisites

Parse uses MongoDB cluster for its storage. Chapter 13 described how to set up a replicated MongoDB using Kubernetes `StatefulSets`. This section assumes you have a three-replica Mongo cluster running in Kubernetes with the names `mongo-0.mongo`, `mongo-1.mongo`, and `mongo-2.mongo`.

These instructions also assume that you have a Docker login; if you don't have one, you can get one for free at *https://docker.com*.

Finally, we assume you have a Kubernetes cluster deployed and the `kubectl` tool properly configured.

Building the parse-server

The open source `parse-server` comes with a *Dockerfile* by default, for easy containerization. First, clone the Parse repository:

```
$ git clone https://github.com/ParsePlatform/parse-server
```

Then move into that directory and build the image:

```
$ cd parse-server
$ docker build -t ${DOCKER_USER}/parse-server .
```

Finally, push that image up to the Docker hub:

```
$ docker push ${DOCKER_USER}/parse-server
```

Deploying the parse-server

Once you have the container image built, deploying the `parse-server` into your cluster is fairly straightforward. Parse looks for three environment variables when being configured:

`APPLICATION_ID`

: An identifier for authorizing your application

`MASTER_KEY`

: An identifier that authorizes the master (root) user

`DATABASE_URI`

: The URI for your MongoDB cluster

Putting this all together, you can deploy Parse as a Kubernetes Deployment using the YAML file in Example 14-1.

Example 14-1. parse.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: parse-server
  namespace: default
spec:
  replicas: 1
  template:
    metadata:
      labels:
        run: parse-server
    spec:
      containers:
      - name: parse-server
        image: ${DOCKER_USER}/parse-server
        env:
        - name: DATABASE_URI
          value: "mongodb://mongo-0.mongo:27017,\
            mongo-1.mongo:27017,mongo-2.mongo\
            :27017/dev?replicaSet=rs0"
        - name: APP_ID
          value: my-app-id
        - name: MASTER_KEY
          value: my-master-key
```

Testing Parse

To test your deployment, you need to expose it as a Kubernetes service. You can do that using the service definition in Example 14-2.

Example 14-2. parse-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: parse-server
  namespace: default
spec:
  ports:
  - port: 1337
    protocol: TCP
    targetPort: 1337
  selector:
    run: parse-server
```

Now your Parse server is up and running and ready to receive requests from your mobile applications. Of course, in any real application you are likely going to want to secure the connection with HTTPS. You can see the `parse-server` GitHub page for more details on such a configuration.

Ghost

Ghost is a popular blogging engine with a clean interface written in JavaScript. It can either use a file-based SQLite database or MySQL for storage.

Configuring Ghost

Ghost is configured with a simple JavaScript file that describes the server. We will store this file as a configuration map. A simple development configuration for Ghost looks like Example 14-3.

Example 14-3. ghost-config.js

```
var path = require('path'),
    config;

config = {
    development: {
        url: 'http://localhost:2368',
        database: {
            client: 'sqlite3',
            connection: {
                filename: path.join(process.env.GHOST_CONTENT,
                                    '/data/ghost-dev.db')
            },
            debug: false
        },
        server: {
            host: '0.0.0.0',
            port: '2368'
        },
        paths: {
            contentPath: path.join(process.env.GHOST_CONTENT, '/')
        }
    }
};

module.exports = config;
```

Once you have this configuration file saved to *config.js*, you can create a Kubernetes ConfigMap object using:

```
$ kubectl apply cm --from-file ghost-config.js ghost-config
```

This creates a ConfigMap that is named `ghost-config`. As with the Parse example, we will mount this configuration file as a volume inside of our container. We will deploy Ghost as a `Deployment` object, which defines this volume mount as part of the Pod template (Example 14-4).

Example 14-4. ghost.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: ghost
spec:
  replicas: 1
```

```
  selector:
    matchLabels:
      run: ghost
  template:
    metadata:
      labels:
        run: ghost
    spec:
      containers:
      - image: ghost
        name: ghost
        command:
        - sh
        - -c
        - cp /ghost-config/config.js /var/lib/ghost/config.js
          && /entrypoint.sh npm start
        volumeMounts:
        - mountPath: /ghost-config
          name: config
      volumes:
      - name: config
        configMap:
          defaultMode: 420
          name: ghost-config
```

One thing to note here is that we are copying the *config.js* file from a different location into the location where Ghost expects to find it, since the ConfigMap can only mount directories, not individual files. Ghost expects other files that are not in that ConfigMap to be present in its directory, and thus we cannot simply mount the entire ConfigMap into */var/lib/ghost*.

You can run this with:

```
$ kubectl apply -f ghost.yaml
```

Once the pod is up and running, you can expose it as a service with:

```
$ kubectl expose deployments ghost --port=2368
```

Once the service is exposed, you can use the `kubectl proxy` command to access the Ghost server:

```
$ kubectl proxy
```

Then visit *http://localhost:8001/api/v1/namespaces/default/services/ghost/proxy/* in your web browser to begin interacting with Ghost.

Ghost + MySQL

Of course, this example isn't very scalable, or even reliable, since the contents of the blog are stored in a local file inside the container. A more scalable approach is to store the blog's data in a MySQL database.

To do this, first modify *config.js* to include:

```
...
database: {
   client: 'mysql',
   connection: {
     host     : 'mysql',
     user     : 'root',
     password : 'root',
     database : 'ghost_db',
     charset  : 'utf8'
   }
 },
...
```

Next, create a new `ghost-config` ConfigMap object:

```
$ kubectl create configmap ghost-config-mysql --from-file config.js
```

Then update the Ghost deployment to change the name of the ConfigMap mounted from `config-map` to `config-map-mysql`:

```
...
      - configMap:
          name: ghost-config-mysql
...
```

Using the instructions from "Kubernetes-Native Storage with StatefulSets", deploy a MySQL server in your Kubernetes cluster. Make sure that it has a service named `mysql` defined as well.

You will need to create the database in the MySQL database:

```
$ kubectl exec -it mysql-zzmlw -- mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
...

mysql> create database ghost_db;
...
```

Finally, perform a rollout to deploy this new configuration.

```
$ kubectl apply -f ghost.yaml
```

Because your Ghost server is now decoupled from its database, you can scale up your Ghost server and it will continue to share the data across all replicas.

Edit *ghost.yaml* to set `spec.replicas` to `3`, then run:

```
$ kubectl apply -f ghost.yaml
```

Your ghost installation is now scaled up to three replicas.

Redis

Redis is a popular in-memory key/value store, with numerous additional features. It's an interesting application to deploy because it is a good example of the value of the Kubernetes Pod abstraction. This is because a reliable Redis installation actually is two programs working together. The first is `redis-server`, which implements the key/value store, and the other is `redis-sentinel`, which implements health checking and failover for a replicated Redis cluster.

When Redis is deployed in a replicated manner, there is a single master server that can be used for both read and write operations. Additionally, there are other replica servers that duplicate the data written to the master and can be used for load-balancing read operations. Any of these replicas can fail over to become the master if the original master fails. This failover is performed by the Redis sentinel. In our deployment, both a Redis server and a Redis sentinel are colocated in the same file.

Configuring Redis

As before, we're going to use Kubernetes ConfigMaps to configure our Redis installation. Redis needs separate configurations for the master and slave replicas. To configure the master, create a file named *master.conf* that contains the code in Example 14-5.

Example 14-5. master.conf

```
bind 0.0.0.0
port 6379

dir /redis-data
```

This directs Redis to bind to all network interfaces on port 6379 (the default Redis port) and store its files in the */redis-data* directory.

The slave configuration is identical, but it adds a single `slaveof` directive. Create a file named *slave.conf* that contains what's in Example 14-6.

Example 14-6. slave.conf

```
bind 0.0.0.0
port 6379

dir .

slaveof redis-0.redis 6379
```

Notice that we are using `redis-0.redis` for the name of the master. We will set up this name using a service and a StatefulSet.

We also need a configuration for the Redis sentinel. Create a file named *sentinel.conf* with the contents of Example 14-7.

Example 14-7. sentinel.conf

```
bind 0.0.0.0
port 26379

sentinel monitor redis redis-0.redis 6379 2
sentinel parallel-syncs redis 1
sentinel down-after-milliseconds redis 10000
sentinel failover-timeout redis 20000
```

Now that we have all of our configuration files, we need to create a couple of simple wrapper scripts to use in our StatefulSet deployment.

The first script simply looks at the hostname for the Pod and determines whether this is the master or a slave, and launches Redis with the appropriate

configuration. Create a file named *init.sh* containing the code in Example 14-8.

Example 14-8. init.sh

```
#!/bin/bash
if [[ ${HOSTNAME} == 'redis-0' ]]; then
  redis-server /redis-config/master.conf
else
  redis-server /redis-config/slave.conf
fi
```

The other script is for the sentinel. In this case it is necessary because we need to wait for the `redis-0.redis` DNS name to become available. Create a script named *sentinel.sh* containing the code in Example 14-9.

Example 14-9. sentinel.sh

```
#!/bin/bash
while ! ping -c 1 redis-0.redis; do
  echo 'Waiting for server'
  sleep 1
done

redis-sentinel /redis-config/sentinel.conf
```

Now we need to package all of these files up into a ConfigMap object. You can do this with a single command line:

```
$ kubectl create configmap \
  --from-file=slave.conf=./slave.conf \
  --from-file=master.conf=./master.conf \
  --from-file=sentinel.conf=./sentinel.conf \
  --from-file=init.sh=./init.sh \
  --from-file=sentinel.sh=./sentinel.sh \
  redis-config
```

Creating a Redis Service

The next step in deploying Redis is to create a Kubernetes service that will provide naming and discovery for the Redis replicas (e.g., `redis-0.redis`). To do this, we create a service without a cluster IP address (Example 14-10).

Example 14-10. redis-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: redis
spec:
  ports:
  - port: 6379
    name: peer
  clusterIP: None
  selector:
    app: redis
```

You can create this service with `kubectl apply -f redis-service.yaml`. Don’t worry that the Pods for the service don’t exist yet. Kubernetes doesn’t care; it will add the right names when the Pods are created.

Deploying Redis

We're ready to deploy our Redis cluster. To do this we're going to use a StatefulSet. We introduced StatefulSets in "Manually Replicated MongoDB with StatefulSets", when we discussed our MongoDB installation. StatefulSets provide indexing (e.g., `redis-0.redis`) as well as ordered creation and deletion semantics (`redis-0` will always be created before `redis-1`, and so on). They're quite useful for stateful applications like Redis, but honestly, they basically look like Kubernetes `Deployments`. For our Redis cluster, here's what the StatefulSet looks like Example 14-11.

Example 14-11. redis.yaml

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: redis
spec:
  replicas: 3
  serviceName: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - command: [sh, -c, source /redis-config/init.sh ]
        image: redis:3.2.7-alpine
        name: redis
        ports:
        - containerPort: 6379
          name: redis
        volumeMounts:
        - mountPath: /redis-config
          name: config
        - mountPath: /redis-data
          name: data
      - command: [sh, -c, source /redis-config/sentinel.sh]
        image: redis:3.2.7-alpine
        name: sentinel
        volumeMounts:
        - mountPath: /redis-config
          name: config
      volumes:
      - configMap:
          defaultMode: 420
          name: redis-config
        name: config
      - emptyDir:
        name: data
```

You can see that there are two containers in this Pod. One runs the *init.sh* script that we created and the main Redis server, and the other is the sentinel that

monitors the servers.

You can also note that there are two volumes defined in the Pod. One is the volume that uses our ConfigMap to configure the two Redis applications, and the other is a simple `emptyDir` volume that is mapped into the Redis server container to hold the application data so that it survives a container restart. For a more reliable Redis installation this could be a network-attached disk, as discussed in Chapter 13.

Now that we've defined our Redis cluster, we can create it using:

```
$ kubectl apply -f redis.yaml
```

Playing with Our Redis Cluster

To demonstrate that we've actually successfully created a Redis cluster, we can perform some tests.

First, we can determine which server the Redis sentinel believes is the master. To do this, we can run the `redis-cli` command in one of the pods:

```
$ kubectl exec redis-2 -c redis \
  -- redis-cli -p 26379 sentinel get-master-addr-by-name redis
```

This should print out the IP address of the `redis-0` pod. You can confirm this using `kubectl get pods -o wide`.

Next, we'll confirm that the replication is actually working.

To do this, first try to read the value `foo` from one of the replicas:

```
$ kubectl exec redis-2 -c redis -- redis-cli -p 6379 get foo
```

You should see no data in the response.

Next, try to write that data to a replica:

```
$ kubectl exec redis-2 -c redis -- redis-cli -p 6379 set foo 10
READONLY You can't write against a read only slave.
```

You can't write to a replica, because it's read-only. Let's try the same command against `redis-0`, which is the master:

```
$ kubectl exec redis-0 -c redis -- redis-cli -p 6379 set foo 10
OK
```

Now try the original read from a replica:

```
$ kubectl exec redis-2 -c redis -- redis-cli -p 6379 get foo
10
```

This shows that our cluster is set up correctly, and data is replicating between masters and slaves.

Summary

In the preceding sections we described how to deploy a variety of applications using assorted Kubernetes concepts. We saw how to put together service-based naming and discovery to deploy web frontends like Ghost as well as API servers like Parse, and we saw how Pod abstraction makes it easy to deploy the components that make up a reliable Redis cluster. Regardless of whether you will actually deploy these applications to production, the examples demonstrated patterns that you can repeat to manage your applications using Kubernetes. We hope that seeing the concepts we described in previous chapters come to life in real-world examples helps you better understand how to make Kubernetes work for you.