The following diagram depicts all three probes graphically:

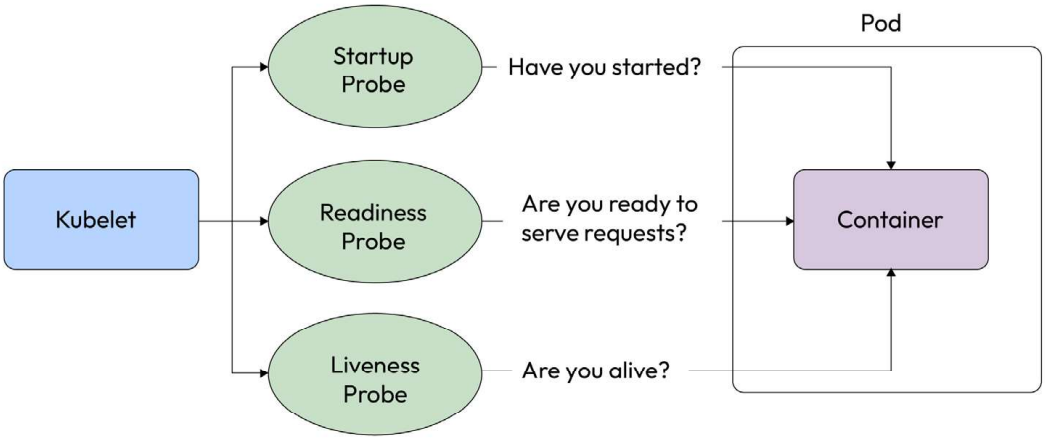

Figure 5.3 – Kubernetes probes

Let's look at each one in turn and understand how and when to use them.

Startup probe

Kubernetes uses **startup probes** to check whether the application has started. You can use startup probes on applications that start slow or those you don't know how long it might take to start. While the startup probe is active, it disables other probes so that they don't interfere with its operation. As the application has not started until the startup probe reports it, there is no point in having any other probes active.

Readiness probe

Readiness probes ascertain whether a container is ready to serve requests. They differ from startup probes because, unlike the startup probe, which only checks whether the application has started, the readiness probe ensures that the container can begin to process requests. A pod is ready when all the containers of the pod are ready. Readiness probes ensure that no traffic is sent to a pod if the pod is not ready. Therefore, it allows for a better user experience.

Liveness probe

Liveness probes are used to check whether a container is running and healthy. The probe checks the health of the containers periodically. If a container is found to be unhealthy, the liveness probe will kill the container. If you've set the `restartPolicy` field of your pod to `Always` or `OnFailure`, Kubernetes will restart the container. Therefore, it improves the service's reliability by detecting deadlocks and ensuring the containers are running instead of just reporting as running.

Now, let's look at an example to understand probes better.

Probes in action

Let's improve the last manifest and add some probes to create the following `nginx-probe.yaml` manifest file:

```
    ...
    startupProbe:
      exec:
        command:
        - cat
        - /usr/share/nginx/html/index.html
      failureThreshold: 30
      periodSeconds: 10
    readinessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 5
    livenessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 3
  restartPolicy: Always
```

The manifest file contains all three probes:

- The startup probe checks whether the `/usr/share/nginx/html/index.html` file exists. It will continue checking it 30 times at an interval of 10 seconds until one of them succeeds. Once it detects the file, the startup probe will stop probing further.
- The readiness probe checks whether there is a listener on port `80` and responds with `HTTP 2xx – 3xx on path /`. It waits for 5 seconds initially and then checks the pod every 5 seconds. If it gets a `2xx – 3xx` response, it will report the container as ready and accept requests.
- The liveness probe checks whether the pod responds with `HTTP 2xx – 3xx` on `port 80` and `path /`. It waits for 5 seconds initially and probes the container every 3 seconds. Suppose, during a check, that it finds the pod not responding for `failureThreshold` times (this defaults to `3`). In that case, it will kill the container, and the kubelet will take appropriate action based on the pod's `restartPolicy` field.
- Let's apply the YAML file and watch the pods come to life by using the following command:

```
$ kubectl delete pod nginx && kubectl apply -f nginx-probe.yaml && \
 kubectl get pod -w
NAME    READY   STATUS              RESTARTS   AGE
nginx   0/1     Running             0          4s
```

```
nginx   0/1     Running             0          11s
nginx   1/1     Running             0          12s
```

As we can see, the pod is quickly ready from the running state. It takes approximately 10 seconds for that to happen as the readiness probe kicks in 10 seconds after the pod starts. Then, the liveness probe keeps monitoring the health of the pod.

Now, let's do something that will break the liveness check. Imagine someone getting a shell to the container and deleting some important files. How do you think the liveness probe will react? Let's have a look.

Let's delete the `/usr/share/nginx/html/index.html` file from the container and then check how the container behaves using the following command:

```
$ kubectl exec -it nginx -- rm -rf /usr/share/nginx/html/index.html && \
 kubectl get pod nginx -w
NAME    READY   STATUS    RESTARTS     AGE
nginx   1/1     Running   0            2m5s
nginx   0/1     Running   1 (2s ago)   2m17s
nginx   1/1     Running   1 (8s ago)   2m22s
```

So, while we watch the pod, the initial delete is only detected after 9 seconds. That's because of the liveness probe. It tries for 9 seconds, three times `periodSeconds`, since `failureThreshold` defaults to `3`, before declaring the pod as unhealthy and killing the container. No sooner does it kill the container than the kubelet restarts it as the pod's `restartPolicy` field is set to `Always`. Then, we see the startup and readiness probes kicking in, and soon, the pod gets ready. Therefore, no matter what, your pods are reliable and will work even if a part of your application is faulty.

> **Tip**
>
> Using readiness and liveness probes will help provide a better user experience, as no requests go to pods that are not ready to process any request. If your application does not respond appropriately, it will replace the container. If multiple pods are running to serve the request, your service is exceptionally resilient.

As we discussed previously, a pod can contain one or more containers. Let's look at some use cases where you might want multiple containers instead of one.

Pod multi-container design patterns

You can run multiple containers in pods in two ways – running a container as an init container or running a container as a helper container to the main container. We'll explore both approaches in the following subsections.

Init containers

Init containers are run before the main container is bootstrapped, so you can use them to initialize your container environment before the main container takes over. Here are some examples:

- A directory might require a particular set of ownership or permissions before you want to start your container using the non-root user
- You might want to clone a Git repository before starting the web server
- You can add a startup delay
- You can generate configuration dynamically, such as for containers that want to dynamically connect to some other pod that it is not aware of during build time but should be during runtime

> **Tip**
> Use init containers only as a last resort, as they hamper the startup time of your containers. Try to bake the configuration within your container image or customize it.

Now, let's look at an example to see init containers in action.

To access the resources for this section, `cd` into the following:

```
$ cd ~/modern-devops/ch5/multi-container-pod/init/
```

Let's serve the `example.com` website from our `nginx` web server. We will get the `example.com` web page and save it as `index.html` in the `nginx` default HTML directory before starting `nginx`.

Access the manifest file, `nginx-init.yaml`, which should contain the following:

```
…
spec:
  containers:
  - name: nginx-container
    image: nginx
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: html-volume
  initContainers:
  - name: init-nginx
    image: busybox:1.28
    command: ['sh', '-c', 'mkdir -p /usr/share/nginx/html && wget -O /usr/share/nginx/
html/index.html http://example.com']
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: html-volume
  volumes:
  - name: html-volume
    emptyDir: {}
```

If we look at the `spec` section of the manifest file, we'll see the following:

- `containers`: This section defines one or more containers that form the pod.
- `containers.name`: This is the container's name, which is `nginx-container` in this case.
- `containers.image`: This is the container image, which is `nginx` in this case.
- `containers.volumeMounts`: This defines a list of volumes that should be mounted to the container. It is similar to the volumes we read about in *Chapter 4*, *Creating and Managing Container Images*.
- `containers.volumeMounts.mountPath`: This defines the path to mount the volume on, which is `/usr/share/nginx/html` in this case. We will share this volume with the init container so that when the init container downloads the `index.html` file from `example.com`, this directory will contain the same file.
- `containers.volumeMounts.name`: This is the name of the volume, which is `html-volume` in this case.
- `initContainers`: This section defines one or more init containers that run before the main containers.
- `initContainers.name`: This is the init container's name, which is `init-nginx` in this case.
- `initContainers.image`: This is the init container image, which is `busybox:1.28` in this case.
- `initContainers.command`: This is the command that the busybox should execute. In this case, `'mkdir -p /usr/share/nginx/html && wget -O /usr/share/nginx/html/index.html http://example.com'` will download the content of `example.com` to the `/usr/share/nginx/html` directory.
- `initContainers.volumeMounts`: We will mount the same volume we defined in `nginx-container` on this container. So, anything we save in this volume will automatically appear in `nginx-container`.
- `initContainers.volumeMounts.mountPath`: This defines the path to mount the volume on, which is `/usr/share/nginx/html` in this case.
- `initContainers.volumeMounts.name`: This is the name of the volume, which is `html-volume` in this case.
- `volumes`: This section defines one or more volumes associated with the pod's containers.
- `volumes.name`: This is the volume's name, which is `html-volume` in this case.
- `volumes.emptyDir`: This defines an `emptyDir` volume. It is similar to a `tmpfs` volume in Docker. Therefore, it is not persistent and lasts just for the container's lifetime.

So, let's go ahead and apply the manifest and watch the pod come to life using the following commands:

```
$ kubectl delete pod nginx && kubectl apply -f nginx-init.yaml && \
 kubectl get pod nginx -w
NAME    READY   STATUS            RESTARTS   AGE
nginx   0/1     Init:0/1          0          0s
nginx   0/1     PodInitializing   0          1s
nginx   1/1     Running           0          3s
```

Initially, we can see that the `nginx` pod shows a status of `Init:0/1`. This means that `0` out of `1` init containers have started initializing. After some time, we can see that the pod reports its status, `PodInitializing`, which means that the init containers have started running. The pod reports a running status once the init containers have run successfully.

Now, once the pod starts to run, we can port-forward the container from port `80` to host port `8080` using the following command:

```
$ kubectl port-forward nginx 8080:80
```

Open a duplicate Terminal and try to `curl` the localhost on port `8080` by using the following command:

```
$ curl localhost:8080
    <title>Example Domain</title>
```

Here, we can see the example domain response from our web server. This means that the init container worked perfectly fine.

As you may have understood by now, the life cycle of init containers ends before the primary containers start, and a pod can contain one or more main containers. So, let's look at a few design patterns we can use in the main container.

The ambassador pattern

The **ambassador pattern** derives its name from an ambassador, an envoy representing a country overseas. You can also think of an ambassador as a proxy of a particular application. Let's say, for example, that you have migrated one of your existing Python Flask applications to containers, and one of your containers needs to communicate with a Redis database. The database always existed in the local host. Therefore, the database connection details within your application contain `localhost` everywhere.

Now, there are two approaches you can take:

- You can change the application code and use config maps and secrets (more on these later) to inject the database connection details into the environment variable.

- You can keep using the existing code and use a second container as a TCP proxy to the Redis database. The TCP proxy will link with the config map and secrets and contain the Redis database's connection details.

> **Tip**
> The ambassador pattern helps developers focus on the application without worrying about the configuration details. Consider using it if you want to decouple application development from config management.

The second approach solves our problem if we wish to do a like-for-like migration. We can use config maps to define the environment-specific configuration without changing the application code. The following diagram shows this approach:

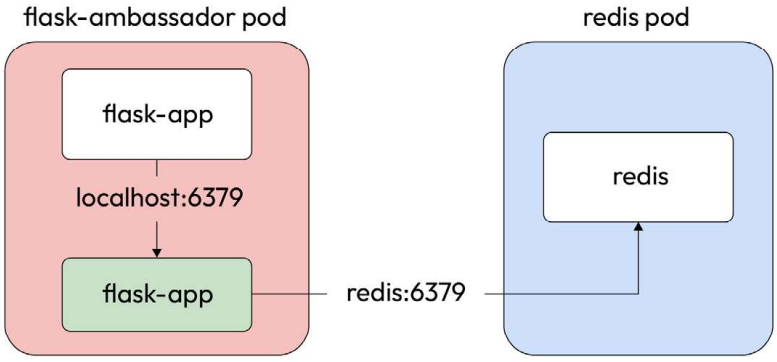

Figure 5.4 – The ambassador pattern

Before we delve into the technicalities, let's understand a config map.

Config map

A **config map** contains key-value pairs that we can use for various purposes, such as defining environment-specific properties or injecting an external variable at container startup or during runtime.

The idea of the config map is to decouple the application with configuration and to externalize configuration at a Kubernetes level. It is similar to using a properties file, for example, to define the environment-specific configuration.

The following diagram explains this beautifully:

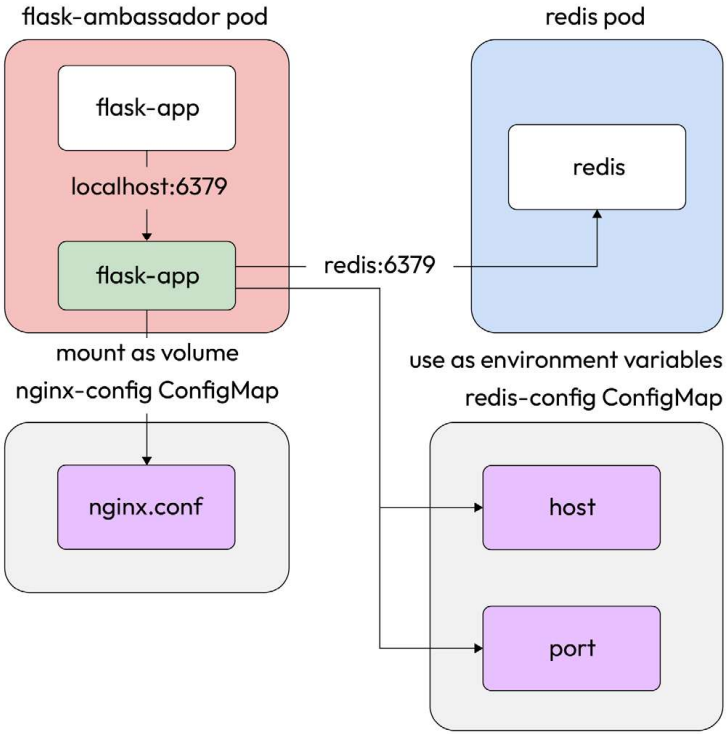

Figure 5.5 – Config maps

We will use `ConfigMap` to define the connection properties of the external Redis database within the ambassador container.

Example application

We will use the example application we used in *Chapter 3*, *Containerization with Docker*, in the *Deploying a sample application with Docker Compose* section. The source code has been replicated into the following directory:

```
$ cd ~/modern-devops/ch5/multi-container-pod/ambassador
```

You can visualize the `app.py` file of the Flask application, the `requirements.txt` file, and the Dockerfile to understand what the application does.

Now, let's build the container using the following command:

```
$ docker build -t <your_dockerhub_user>/flask-redis .
```

Let's push it to our container registry using the following command:

```
$ docker push <your_dockerhub_user>/flask-redis
```

As you may have noticed, the `app.py` code defines the cache as `localhost:6379`. We will run an ambassador container on `localhost:6379`. The proxy will tunnel the connection to the `redis` pod running elsewhere.

First, let's create the `redis` pod using the following command:

```
$ kubectl run redis --image=redis
```

Now, let's expose the `redis` pod to the cluster resources via a `Service` resource. This will allow any pod within the cluster to communicate with the `redis` pod using the `redis` hostname. We will discuss Kubernetes `Service` resources in the next chapter in detail:

```
$ kubectl expose pod redis --port 6379
```

Cool! Now that the pod and the `Service` resource are up and running, let's work on the ambassador pattern.

We need to define two config maps first. The first describes the `redis` host and port details, while the second defines the template `nginx.conf` file to work as a reverse proxy.

The `redis-config-map.yaml` file looks like this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: redis-config
data:
  host: "redis"
  port: "6379"
```

The preceding YAML file defines a config map called `redis-config` that contains `host` and `port` properties. You can have multiple config maps, one for each environment.

The `nginx-config-map.yaml` file looks as follows:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
data:
  nginx.conf: |
    ...
    stream {
        server {
            listen     6379;
            proxy_pass stream_redis_backend;
        }
        upstream stream_redis_backend {
            server REDIS_HOST:REDIS_PORT;
        }
    }
```

This config map injects the `nginx.conf` template as a config map value. This template defines the configuration of our ambassador pod to listen on `localhost:6379` and tunnel the connection to `REDIS_HOST:REDIS_PORT`. As the `REDIS_HOST` and `REDIS_PORT` values are placeholders, we must fill these up with the correct values that we obtained from the `redis-config` config map. To do that, we can mount this file to a volume and then manipulate it. We can use `initContainer` to initialize the proxy with the correct configuration.

Now, let's look at the pod configuration manifest, `flask-ambassador.yaml`. There are multiple parts of this YAML file. Let's look at the `containers` section first:

```
...
spec:
  containers:
  - name: flask-app
    image: <your_dockerhub_user>/flask-redis
  - name: nginx-ambassador
    image: nginx
    volumeMounts:
    - mountPath: /etc/nginx
      name: nginx-volume
  ...
```

This section contains a container called `flask-app` that uses the `<your_dockerhub_user>/flask-redis` image that we built in the previous section. The second container is the `nginx-ambassador` container that will act as the proxy to `redis`. Therefore, we have mounted the `/etc/nginx` directory on a volume. This volume is also mounted on the init container to generate the required configuration before `nginx` boots up.

The following is the `initContainers` section:

```
  initContainers:
  - name: init-nginx
    image: busybox:1.28
    command: ['sh', '-c', 'cp -L /config/nginx.conf /etc/nginx/nginx.conf && sed -i "s/
REDIS_HOST/${REDIS_HOST}/g" /etc/nginx/nginx.conf']
    env:
      - name: REDIS_HOST
        valueFrom:
          configMapKeyRef:
            name: redis-config
            key: host
      - name: REDIS_PORT
        valueFrom:
          configMapKeyRef:
            name: redis-config
            key: port
    volumeMounts:
    - mountPath: /etc/nginx
      name: nginx-volume
    - mountPath: /config
      name: config
```

This section defines a `busybox` container – `init-nginx`. The container needs to generate the `nginx-ambassador` proxy configuration to communicate with Redis; therefore, two environment variables are present. Both environment variables are sourced from the `redis-config` config map. Apart from that, we have also mounted the `nginx.conf` file from the `nginx-config` config map. The `command` section within the init container uses the environment variables to replace placeholders within the `nginx.conf` file, after which we get a TCP proxy to the Redis backend.

The `volumes` section defines `nginx-volume` as an `emptyDir` volume, and the `config` volume is mounted from the `nginx.conf` file present in the `nginx-config` config map:

```
  volumes:
  - name: nginx-volume
    emptyDir: {}
  - name: config
    configMap:
      name: nginx-config
      items:
      - key: "nginx.conf"
        path: "nginx.conf"
```

Now, let's start applying the YAML files in steps.

Apply both of the config maps using the following commands:

```
$ kubectl apply -f redis-config-map.yaml
$ kubectl apply -f nginx-config-map.yaml
```

Let's apply the pod configuration using the following command:

```
$ kubectl apply -f flask-ambassador.yaml
```

Get the pod to see whether the configuration is correct by using the following command:

```
$ kubectl get pod/flask-ambassador
NAME               READY   STATUS    RESTARTS   AGE
flask-ambassador   2/2     Running   0          10s
```

As the pod is running successfully now, let's port-forward `5000` to the localhost for some tests by using the following command:

```
$ kubectl port-forward flask-ambassador 5000:5000
```

Now, open a duplicate Terminal and try to `curl` on `localhost:5000` using the following command:

```
$ curl localhost:5000
Hi there! This page was last visited on 2023-06-18, 16:52:28.
$ curl localhost:5000
Hi there! This page was last visited on 2023-06-18, 16:52:28.
$ curl localhost:5000
Hi there! This page was last visited on 2023-16-28, 16:52:32.
```

As we can see, every time we `curl` the application, we get the last visited time on our screen. The ambassador pattern is working.

This was a simple example of the ambassador pattern. There are advanced configurations you can do to add fine-grained control on how your application should interact with the outside world. You can use the ambassador pattern to secure traffic that moves from your containers. It also simplifies application development for your development team as they need not worry about these nuances. In contrast, the operations team can use these containers to manage your environment in a better way without stepping on each other's toes.

> **Tip**
>
> As the ambassador pattern adds some overhead as you tunnel connections via a proxy, you should only use it if the management benefits outweigh the extra cost you incur because of the ambassador container.

Now, let's look at another multi-container pod pattern – sidecars.

The sidecar pattern

Sidecars derive their names from motorcycle sidecars. The sidecar does not change the bike's core functionality and can work perfectly without it. Instead, it adds an extra seat, a functionality that helps you give an additional person a ride. Similarly, sidecars in a pod are helper containers that provide functionalities unrelated to the main container's core functionality and enhance it instead. Examples include logging and monitoring containers. Keeping a separate container for logging will help decouple the logging responsibilities from your main container, which will help you monitor your application even when the main container goes down for some reason.

It also helps if there is some issue with the logging code, and instead of the entire application going down, only the logging container is impacted. You can also use sidecars to keep helper or related containers together with the main container since we know containers within the pod share the same machine.

> **Tip**
>
> Only use multi-container pods if two containers are functionally related and work as a unit.

You can also use sidecars to segregate your application with secrets. For example, if you are running a web application that needs access to specific passwords to operate, it would be best to mount the secrets to a sidecar and let the sidecar provide the passwords to the web application via a link. This is because if someone gains access to your application container's filesystem, they cannot get hold of your passwords as another container is responsible for sourcing it, as shown in the following diagram:

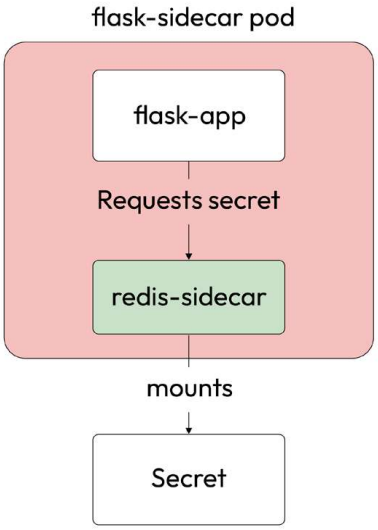

Figure 5.6 – The sidecar pattern

Let's implement the preceding pattern to understand a sidecar better. We have a Flask application that interacts with a Redis sidecar. We will pre-populate the Redis sidecar with a secret `foobar`, and we will do that by using the Kubernetes secret resource.

Secrets

Secrets are very similar to config maps, with the difference that the secret values are `base64`-encoded instead of `plaintext`. While `base64` encoding does not make any difference, and it is as bad as `plaintext` from a security standpoint, you should use secrets for sensitive information such as passwords. That is because the Kubernetes community will develop a solution to tighten the security around secrets in future releases. If you use secrets, you will directly benefit from it.

> **Tip**
> As a rule of thumb, always use secrets for confidential data, such as API keys and passwords, and config maps for non-sensitive configuration data.

To access the files for this section, go to the following directory:

```
$ cd ~/modern-devops/ch5/multi-container-pod/sidecar
```

Now, let's move on to the example Flask application.

Example application

The Flask application queries a Redis sidecar for the secret and sends that as a response. That is not ideal, as you won't send secrets back as a response, but for this demo, let's go ahead with that.

So, first, let's design our sidecar so that it pre-populates data within the container after it starts.

We need to create a secret named `secret` with a value of `foobar`. Now, `base64`-encode the Redis command to set the secret into the cache by running the following command:

```
$ echo 'SET secret foobar' | base64
U0VUIHNlY3JldCBmb29iYXIK
```

Now that we have the `base64`-encoded secret, we can create a `redis-secret.yaml` manifest with the string as follows:

```
apiVersion: v1
kind: Secret
metadata:
  name: redis-secret
data:
  redis-secret: U0VUIHNlY3JldCBmb29iYXIK
```

Then, we need to build the Redis container so that this secret is created at startup. To access the files for this section, go to the following directory:

```
$ cd ~/modern-devops/ch5/multi-container-pod/sidecar/redis/
```

Create an `entrypoint.sh` file, as follows:

```
redis-server --daemonize yes && sleep 5
redis-cli < /redis-master/init.redis
redis-cli save
redis-cli shutdown
redis-server
```

The shell script looks for a file, `init.redis`, within the `/redis-master` directory and runs the `redis-cli` command on it. This means the cache will be pre-populated with the values defined in our secret, provided we mount the secret as `/redis-master/init.redis`.

Then, we must create a Dockerfile that will use this `entrypoint.sh` script, as follows:

```
FROM redis
COPY entrypoint.sh /tmp/
CMD ["sh", "/tmp/entrypoint.sh"]
```

Now that we are ready, we can build and push the code to Docker Hub:

```
$ docker build -t <your_dockerhub_user>/redis-secret .
$ docker push <your_dockerhub_user>/redis-secret
```

Now that we are ready with the Redis image, we must build the Flask application image. To access the files for this section, `cd` into the following directory:

```
$ cd ~/modern-devops/ch5/multi-container-pod/sidecar/flask
```

Let's look at the `app.py` file first:

```
...
cache = redis.Redis(host='localhost', port=6379)
def get_secret():
    try:
        secret = cache.get('secret')
        return secret
...
def index():
    secret = str(get_secret().decode('utf-8'))
    return 'Hi there! The secret is {}.\n'.format(secret)
```

The code is simple – it gets the secret from the cache and returns that in the response.

We also created the same Dockerfile that we did in the previous section.

So, let's build and push the container image to Docker Hub:

```
$ docker build -t <your_dockerhub_user>/flask-redis-secret .
$ docker push <your_dockerhub_user>/flask-redis-secret
```

Now that our images are ready, let's look at the pod manifest, `flask-sidecar.yaml`, which is present in the `~/modern-devops/ch5/multi-container-pod/sidecar/` directory:

```
...
spec:
  containers:
  - name: flask-app
    image: <your_dockerhub_user>/flask-redis-secret
  - name: redis-sidecar
    image: <your_dockerhub_user>/redis-secret
    volumeMounts:
    - mountPath: /redis-master
      name: secret
  volumes:
  - name: secret
    secret:
      secretName: redis-secret
      items:
      - key: redis-secret
        path: init.redis
```

The pod defines two containers – `flask-app` and `redis-sidecar`. The `flask-app` container runs the Flask application that will interact with `redis-sidecar` for the secret. The `redis-sidecar` container has mounted the `secret` volume on `/redis-master`. The pod definition also contains a single volume called `secret`, and the volume points to the `redis-secret` secret and mounts that as a file, `init.redis`.

So, in the end, we have a file, `/redis-master/init.redis`, and, as we know, the `entrypoint.sh` script looks for this file and runs the `redis-cli` command to pre-populate the Redis cache with the secret data.

Let's apply the secret first using the following command:

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

Then, we can apply the `flask-sidecar.yaml` file using the following command:

```
$ kubectl apply -f flask-sidecar.yaml
```

Now, let's get the pods using the following command:

```
$ kubectl get pod flask-sidecar
NAME            READY   STATUS              RESTARTS   AGE
flask-sidecar   2/2     Running             0          11s
```

As the pod is running, it's time to port-forward it to the host using the following command:

```
$ kubectl port-forward flask-sidecar 5000:5000
```

Now, let's open a duplicate Terminal, run the `curl localhost:5000` command, and see what we get:

```
$ curl localhost:5000
Hi there! The secret is foobar.
```

As we can see, we get the secret, `foobar`, in the response. The sidecar is working correctly!

Now, let's look at another popular multi-container pod pattern – the adapter pattern.

The adapter pattern

As its name suggests, the **adapter pattern** helps change something to fit a standard, such as cell phones and laptop adapters, which convert our main power supply into something our devices can digest. A great example of the adapter pattern is transforming log files so that they fit an enterprise standard and feed your log analytics solution:

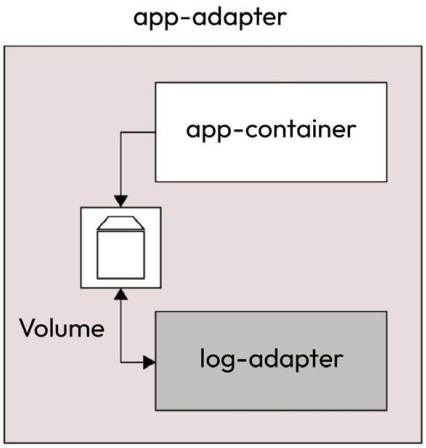

Figure 5.7 – The adapter pattern

It helps when you have a heterogeneous solution outputting log files in several formats but a single log analytics solution that only accepts messages in a particular format. There are two ways of doing this: changing the code for outputting log files in a standard format or using an adapter container to execute the transformation.

Let's look at the following scenario to understand it further.

We have an application that continuously outputs log files without a date at the beginning. Our adapter should read the stream of logs and append the timestamp as soon as a logline is generated.

For this, we will use the following pod manifest, `app-adapter.yaml`:

```
...
spec:
  volumes:
  - name: logs
    emptyDir: {}
  containers:
  - name: app-container
    image: ubuntu
    command: ["/bin/bash"]
    args: ["-c", "while true; do echo 'This is a log line' >> /var/log/app.log; sleep
2;done"]
    volumeMounts:
    - name: logs
      mountPath: /var/log
  - name: log-adapter
    image: ubuntu
    command: ["/bin/bash"]
    args: ["-c", "apt update -y && apt install -y moreutils && tail -f /var/log/app.log |
ts '[%Y-%m-%d %H:%M:%S]' > /var/log/out.log"]
    volumeMounts:
    - name: logs
      mountPath: /var/log
```

The pod contains two containers – the app container, which is a simple Ubuntu container that outputs `This is a log line` every 2 seconds, and the log adapter, which continuously tails the `app.log` file, adds a timestamp at the beginning of the line, and sends the resulting output to `/var/log/out.log`. Both containers share the `/var/log` volume, which is mounted as an `emptyDir` volume on both containers.

Now, let's apply this manifest using the following command:

```
$ kubectl apply -f app-adapter.yaml
```

Let's wait a while and check whether the pod is running by using the following command:

```
$ kubectl get pod app-adapter
NAME          READY   STATUS    RESTARTS   AGE
app-adapter   2/2     Running   0          8s
```

As the pod is running, we can now get a shell into the log adapter container by using the following command:

```
$ kubectl exec -it app-adapter -c log-adapter -- bash
```

When we get into the shell, we can `cd` into the `/var/log` directory and list its contents using the following command:

```
root@app-adapter:/# cd /var/log/ && ls
app.log apt/ dpkg.log out.log
```

As we can see, we get `app.log` and `out.log` as two files. Now, let's use the `cat` command to print both of them to see what we get.

First, `cat` the `app.log` file using the following command:

```
root@app-adapter:/var/log# cat app.log
This is a log line
This is a log line
This is a log line
```

Here, we can see that a series of log lines are being printed.

Now, `cat` the `out.log` file to see what we get using the following command:

```
root@app-adapter:/var/log# cat out.log
[2023-06-18 16:35:25] This is a log line
[2023-06-18 16:35:27] This is a log line
[2023-06-18 16:35:29] This is a log line
```

Here, we can see timestamps in front of the log line. This means that the adapter pattern is working correctly. You can then export this log file to your log analytics tool.

Summary

We have reached the end of this critical chapter. We've covered enough ground to get you started with Kubernetes and understand and appreciate the best practices surrounding it.

We started with Kubernetes and why we need it and then discussed bootstrapping a Kubernetes cluster using Minikube and KinD. Then, we looked at the pod resource and discussed creating and managing pods, troubleshooting them, ensuring your application's reliability using probes, and multi-container design patterns to appreciate why Kubernetes uses pods in the first place instead of containers.

In the next chapter, we will deep dive into the advanced aspects of Kubernetes by covering controllers, services, ingresses, managing a stateful application, and Kubernetes command-line best practices.

Questions

Answer the following questions to test your knowledge of this chapter:

1. All communication with Kubernetes happens via which of the following?

   A. Kubelet

   B. API server

   C. Etcd

   D. Controller manager

   E. Scheduler

2. Which of the following is responsible for ensuring that the cluster is in the desired state?

   A. Kubelet

   B. API server

   C. Etcd

   D. Controller manager

   E. Scheduler

3. Which of the following is responsible for storing the desired state of the cluster?

   A. Kubelet

   B. API server

   C. Etcd

   D. Controller manager

   E. Scheduler

4. A pod can contain more than one container. (True/False)

5. You can use port-forwarding for which of the following use cases? (Choose two)

   A. For troubleshooting a misbehaving pod

   B. For exposing a service to the internet

   C. For accessing a system service such as the Kubernetes dashboard

6. Using a combination of which two probes can help you ensure that your application is reliable even when your application has some intermittent issues? (Choose two.)

   A. Startup probe

   B. Liveness probe

   C. Readiness probe

7. We may use KinD in production. (True/False)
8. Which of the following multi-container patterns is used as a forward proxy?

   A. Ambassador

   B. Adapter

   C. Sidecar

   D. Init containers

Answers

Here are the answers to this chapter's questions:

1. B
2. D
3. C
4. True
5. A, C
6. B, C
7. False
8. A

6

Managing Advanced Kubernetes Resources

In the previous chapter, we covered Kubernetes and why we need it and then discussed bootstrapping a Kubernetes cluster using MiniKube and KinD. We then looked at the `Pod` resource and discussed how to create and manage pods, how to troubleshoot them, and how to ensure your application's reliability using probes, along with multi-container design patterns to appreciate why Kubernetes uses pods in the first place instead of containers. We also looked at `Secrets` and `ConfigMaps`.

Now, we will dive deep into the advanced aspects of Kubernetes and Kubernetes command-line best practices.

In this chapter, we're going to cover the following main topics:

- The need for advanced Kubernetes resources
- Kubernetes Deployments
- Kubernetes Services and Ingresses
- Horizontal pod autoscaling
- Managing stateful applications
- Kubernetes command-line best practices, tips, and tricks

So, let's dive in!

Technical requirements

For this chapter, we will spin up a cloud-based Kubernetes cluster, **Google Kubernetes Engine** (**GKE**), for the exercises. That is because you will not be able to spin up load balancers and PersistentVolumes within your local system, and therefore, we cannot use KinD and MiniKube in this chapter.

Currently, **Google Cloud Platform** (**GCP**) provides a free $300 trial for 90 days, so you can go ahead and sign up for one at `https://cloud.google.com/free`.

Spinning up GKE

Once you've signed up and logged in to your console, you can open the Google Cloud Shell CLI to run the commands.

You need to enable the GKE API first using the following command:

```
$ gcloud services enable container.googleapis.com
```

To create a three-node GKE cluster, run the following command:

```
$ gcloud container clusters create cluster-1 --zone us-central1-a
```

And that's it! The cluster is up and running.

You will also need to clone the following GitHub repository for some exercises: `https://github.com/PacktPublishing/Modern-DevOps-Practices-2e`.

Run the following command to clone the repository into your home directory and `cd` into the `ch6` directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
$ cd modern-devops/ch6
```

Now, let's understand why we need advanced Kubernetes resources.

The need for advanced Kubernetes resources

In the last chapter, we looked at pods, the basic building blocks of Kubernetes that provide everything for your containers to run within a Kubernetes environment. However, pods on their own are not that effective. The reason is that while they define a container application and its specification, they do not replicate, auto-heal, or maintain a particular state. When you delete a pod, the pod is gone. You cannot maintain multiple versions of your code or roll out and roll back releases using a pod. You also cannot autoscale your application with traffic with pods alone. Pods do not allow you to expose your containers to the outside world, and they do not provide traffic management capabilities such as load balancing, content and path-based routing, storing persistent data to externally attached storage,

and so on. To solve these problems, Kubernetes provides us with specific advanced resources, such as Deployments, Services, Ingresses, PersistentVolumes and claims, and StatefulSets. Let's start with Kubernetes Deployments in the next section.

Kubernetes Deployments

Let's understand Kubernetes Deployments using a simple analogy.

Imagine you're a chef preparing a particular dish in your kitchen. You want to make sure that it is consistently perfect every time you serve it, and you want to be able to change the recipe without causing a chaotic mess.

In the world of Kubernetes, a "Deployment" is like your sous chef. It helps you create and manage copies of your pods effortlessly.

Here's how it works:

- **Creating consistency**: You want to serve your dish to many guests. Therefore, instead of cooking each plate separately, you prepare a bunch of them at once. All of them should taste the same and strictly as intended. A Kubernetes Deployment does the same for your pod. It creates multiple identical copies of your pod, ensuring they all have the same setup.
- **Updating safely**: Now, imagine you have a new twist for your dish. You want to try it out, but you want your guests only to eat something if it turns out right. Similarly, when you want to update your app in Kubernetes, the `Deployment` resource slowly and carefully replaces old copies with new ones individually, so your app is always available, and your guests (or users) don't notice any hiccups.
- **Rolling back gracefully**: Sometimes, experiments don't go as planned, and you must revert to the original recipe. Just as in your kitchen, Kubernetes lets you roll back to the previous version of your pod if things don't work out with the new one.
- **Scaling easily**: Imagine your restaurant suddenly gets a rush of customers, and you need more plates for your special dish. A Kubernetes Deployment helps with that, too. It can quickly create more copies of your pod to handle the increased demand and remove them when things quieten down.
- **Managing multiple kitchens**: If you have multiple restaurants, you'd want your signature dish to taste the same in all of them. Similarly, if you're using Kubernetes across different environments such as testing, development, and production, Deployments help keep things consistent.

In essence, Kubernetes Deployments help manage your pod, like a sous chef manages the dishes served from a kitchen. They ensure consistency, safety, and flexibility, ensuring your application runs smoothly and can be updated without causing a mess in your *software kitchen*.

Container application Deployments within Kubernetes are done through `Deployment` resources. `Deployment` resources employ `ReplicaSet` resources behind the scenes, so it would be good to look at `ReplicaSet` resources before we move on to understand `Deployment` resources.

ReplicaSet resources

`ReplicaSet` resources are Kubernetes controllers that help you run multiple pod replicas at a given time. They provide horizontal scaling for your container workloads, forming the basic building block of a horizontal scale set for your containers, a group of similar containers tied together to run as a unit.

`ReplicaSet` resources define the number of pod replicas to run at a given time. The Kubernetes controller then tries to maintain the replicas and recreates a pod if it goes down.

You should never use `ReplicaSet` resources on their own, but instead, they should act as a backend to a `Deployment` resource.

For understanding, however, let's look at an example. To access the resources for this section, `cd` into the following:

```
$ cd ~/modern-devops/ch6/deployments/
```

The `ReplicaSet` resource manifest, `nginx-replica-set.yaml`, looks like this:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
```

The resource manifest includes `apiVersion` and `kind`, as with any other resource. It also contains a `metadata` section that defines the resource's `name` and `labels` attributes, similar to any other Kubernetes resource.

The `spec` section contains the following attributes:

- `replicas`: This defines the number of pod replicas matched by the selector to run at a given time.
- `selector`: This defines the basis on which the `ReplicaSet` resource will include pods.
- `selector.matchLabels`: This defines labels and their values to select pods. Therefore, the `ReplicaSet` resource will select any pod with the `app: nginx` label.
- `template`: This is an optional section that you can use to define the pod template. This section's contents are very similar to defining a pod, except it lacks the `name` attribute, as the `ReplicaSet` resource will generate dynamic names for pods. If you don't include this section, the `ReplicaSet` resource will still try to acquire existing pods with matching labels. However, it cannot create new pods because of the missing template. Therefore, it is best practice to specify a template for a `ReplicaSet` resource.

Let's go ahead and apply this manifest to see what we get:

```
$ kubectl apply -f nginx-replica-set.yaml
```

Now, let's run the following command to list the `ReplicaSet` resources:

```
$ kubectl get replicaset
NAME    DESIRED   CURRENT   READY   AGE
nginx   3         3         0       9s
```

Right—so, we see that there are three desired replicas. Currently, `3` replicas are running, but `0` are ready. Let's wait for a while and then rerun the following command:

```
$ kubectl get replicaset
NAME    DESIRED   CURRENT   READY   AGE
nginx   3         3         3       1m10s
```

And now, we see `3` ready pods that are awaiting a connection. Now, let's list the pods and see what the `ReplicaSet` resource has done behind the scenes using the following command:

```
$ kubectl get pod
NAME          READY   STATUS    RESTARTS   AGE
nginx-6qr9j   1/1     Running   0          1m32s
nginx-7hkqv   1/1     Running   0          1m32s
nginx-9kvkj   1/1     Running   0          1m32s
```

There are three `nginx` pods, each with a name that starts with `nginx` but ends with a random hash. The `ReplicaSet` resource has appended a random hash to generate unique pods at the end of the `ReplicaSet` resource name. Yes—the name of every resource of a particular kind in Kubernetes should be unique.

Let's go ahead and use the following command to delete one of the pods from the `ReplicaSet` resource and see what we get:

```
$ kubectl delete pod nginx-9kvkj && kubectl get pod
pod "nginx-9kvkj" deleted
NAME          READY   STATUS    RESTARTS   AGE
nginx-6qr9j   1/1     Running   0          8m34s
nginx-7hkqv   1/1     Running   0          8m34s
nginx-9xbdf   1/1     Running   0          5s
```

We see that even though we deleted the `nginx-9kvkj` pod, the `ReplicaSet` resource has replaced it with a new pod, `nginx-9xbdf`. That is how `ReplicaSet` resources work.

You can delete a `ReplicaSet` resource just like any other Kubernetes resource. You can run the `kubectl delete replicaset <ReplicaSet name>` command for an imperative approach or use `kubectl delete -f <manifest_file>` for a declarative approach.

Let's use the former approach and delete the `ReplicaSet` resource by using the following command:

```
$ kubectl delete replicaset nginx
```

Let's check whether the `ReplicaSet` resource has been deleted by running the following command:

```
$ kubectl get replicaset
No resources found in default namespace.
```

We don't have anything in the `default` namespace. This means that the `ReplicaSet` resource is deleted.

As we discussed, `ReplicaSet` resources should not be used on their own but should instead be the backend of `Deployment` resources. Let's now look at Kubernetes `Deployment` resources.

Deployment resources

Kubernetes `Deployment` resources help to manage deployments for container applications. They are typically used for managing stateless workloads. You can still use them to manage stateful applications, but the recommended approach for stateful applications is to use `StatefulSet` resources.

Kubernetes Deployments use `ReplicaSet` resources as a backend, and the chain of resources looks like what's shown in the following diagram:

Figure 6.1 – Deployment chain

Let's take the preceding example and create an nginx `Deployment` resource manifest—`nginx-deployment.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
```

The manifest is very similar to the `ReplicaSet` resource, except for the `kind` attribute—`Deployment`, in this case.

Let's apply the manifest by using the following command:

```
$ kubectl apply -f nginx-deployment.yaml
```

So, as the `Deployment` resource has been created, let's look at the chain of resources it created. Let's run `kubectl get` to list the `Deployment` resources using the following command:

```
$ kubectl get deployment
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
nginx   3/3     3            3           6s
```

And we see there is one `Deployment` resource called `nginx`, with `3/3` ready pods and `3` up-to-date pods. As `Deployment` resources manage multiple versions, `UP-TO-DATE` signifies whether the latest `Deployment` resource has rolled out successfully. We will look into the details of this in the subsequent sections. It also shows `3` available pods at that time.

As we know `Deployment` resources create `ReplicaSet` resources in the background, let's get the `ReplicaSet` resources using the following command:

```
$ kubectl get replicaset
NAME               DESIRED   CURRENT   READY   AGE
nginx-6799fc88d8   3         3         3       11s
```

And we see that the `Deployment` resource has created a `ReplicaSet` resource, which starts with `nginx` and ends with a random hash. That is required as a `Deployment` resource might contain one or more `ReplicaSet` resources. We will look at how in the subsequent sections.

Next in the chain are pods, so let's get the pods using the following command to see for ourselves:

```
$ kubectl get pod
NAME                     READY   STATUS    RESTARTS   AGE
nginx-6799fc88d8-d52mj   1/1     Running   0          15s
nginx-6799fc88d8-dmpbn   1/1     Running   0          15s
nginx-6799fc88d8-msvxw   1/1     Running   0          15s
```

And, as expected, we have three pods. Each begins with the `ReplicaSet` resource name and ends with a random hash. That's why you see two hashes in the pod name.

Let's assume you have a new release and want to deploy a new version of your container image. So, let's update the `Deployment` resource with a new image using the following command:

```
$ kubectl set image deployment/nginx nginx=nginx:1.16.1
deployment.apps/nginx image updated
```

To check the deployment status, run the following command:

```
$ kubectl rollout status deployment nginx
deployment "nginx" successfully rolled out
```

Imperative commands such as those just shown are normally not used in production environments because they lack the audit trail you would get with declarative manifests using Git to version them. However, if you do choose to use imperative commands, you can always record the change cause of the previous rollout using the following command:

```
$ kubectl annotate deployment nginx kubernetes.io/change-cause\
="Updated nginx version to 1.16.1" --overwrite=true
deployment.apps/nginx annotated
```

To check the Deployment history, run the following command:

```
$ kubectl rollout history deployment nginx
deployment.apps/nginx
REVISION  CHANGE-CAUSE
1         <none>
2         Updated nginx version to 1.16.1
```

As we can see, there are two revisions in the Deployment history. Revision `1` was the initial Deployment, and revision `2` was because of the `kubectl set image` command we ran, as evident from the `CHANGE-CAUSE` column.

Let's say you find an issue after Deployment and want to roll it back to the previous version. To do so and also recheck the status of the Deployment, run the following command:

```
$ kubectl rollout undo deployment nginx && kubectl rollout status deployment nginx
deployment.apps/nginx rolled back
Waiting for deployment "nginx" rollout to finish: 2 out of 3 new replicas have been
updated...
Waiting for deployment "nginx" rollout to finish: 1 old replicas are pending
termination...
deployment "nginx" successfully rolled out
```

And finally, let's recheck the deployment history using the following command:

```
$ kubectl rollout history deployment nginx
deployment.apps/nginx
REVISION  CHANGE-CAUSE
2         Updated nginx version to 1.16.1
3         <none>
```

And we get revision `3` with a `CHANGE-CAUSE` value of `<none>`. In this case, we did not annotate the rollback as in the last command.

> **Tip**
> Always annotate Deployment updates as it becomes easier to peek into the history to see what got deployed.

Now, let's look at some common Kubernetes Deployment strategies to understand how to use Deployments effectively.

Kubernetes Deployment strategies

Updating an existing Deployment requires you to specify a new container image. That is why we version container images in the first place so that you can roll out, and roll back, application changes as required. As we run everything in containers—and containers, by definition, are ephemeral—this enables a host of different deployment strategies that we can implement. There are several deployment strategies, and some of these are set out as follows:

- **Recreate**: This is the simplest of all. Delete the old pod and deploy a new one.
- **Rolling update**: Slowly roll out the pods of the new version while still running the old version, and slowly remove the old pods as the new pods get ready.
- **Blue/green**: This is a derived deployment strategy where we keep both versions running simultaneously and switch the traffic to the newer version when we want.

- **Canary**: This applies to Blue/Green Deployments where we switch a percentage of traffic to the newer version of the application before fully rolling out the release.
- **A/B testing**: A/B testing is more of a technique to apply to Blue/Green Deployments. This is when you want to roll out the newer version to a subset of willing users and study the usage patterns before completely rolling out the newer version. You do not get A/B testing out of the box with Kubernetes but instead should rely on service mesh technologies that plug in well with Kubernetes, such as **Istio**, **Linkerd**, and **Traefik**.

Kubernetes provides two deployment strategies out of the box—`Recreate` and `RollingUpdate`.

Recreate

The `Recreate` strategy is the most straightforward deployment strategy. When you update the `Deployment` resource with the `Recreate` strategy, Kubernetes immediately spins down the old `ReplicaSet` resource and creates a new one with the required number of replicas along the lines of the following diagram:

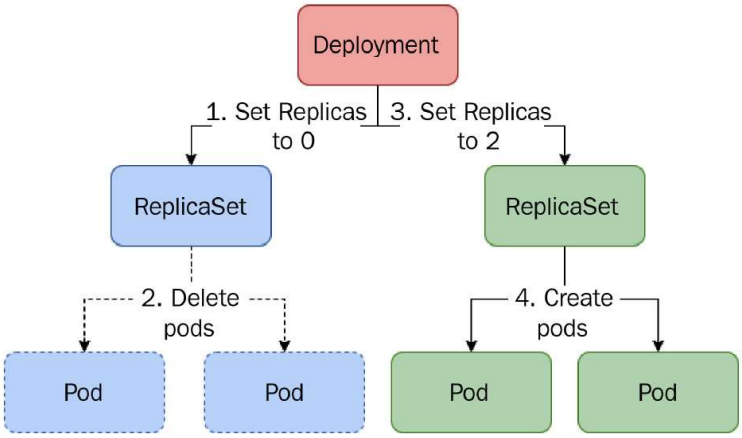

Figure 6.2 – Recreate strategy

Kubernetes does not delete the old `ReplicaSet` resource but instead sets `replicas` to `0`. That is required to roll back to the old version quickly. This approach results in downtime and is something you want to use only in case of a constraint. Thus, this strategy isn't the default deployment strategy in Kubernetes.

> **Tip**
>
> You can use the `Recreate` strategy if your application does not support multiple replicas, if it does not support more than a certain number of replicas (such as applications that need to maintain a quorum), or if it does not support multiple versions at once.

Let's update `nginx-deployment` with the `Recreate` strategy. Let's look at the `nginx-recreate.yaml` file:

```
...
spec:
  replicas: 3
  strategy:
    type: Recreate
...
```

The YAML file now contains a `strategy` section with a `Recreate` type. Now, let's apply the `nginx-recreate.yaml` file and watch the `ReplicaSet` resources using the following command:

```
$ kubectl apply -f nginx-recreate.yaml && kubectl get replicaset -w
deployment.apps/nginx configured
NAME               DESIRED   CURRENT   READY   AGE
nginx-6799fc88d8   0         0         0       0s
nginx-6889dfccd5   0         3         3       7m42s
nginx-6889dfccd5   0         0         0       7m42s
nginx-6799fc88d8   3         0         0       1s
nginx-6799fc88d8   3         3         0       2s
nginx-6799fc88d8   3         3         3       6s
```

The `Deployment` resource creates a new `ReplicaSet` resource—`nginx-6799fc88d8`—with `0` desired replicas. It then sets `0` desired replicas to the old `ReplicaSet` resource and waits for the old `ReplicaSet` resource to be completely evicted. It then starts automatically rolling out the new `ReplicaSet` resource to the desired images.

RollingUpdate

When you update the Deployment with a `RollingUpdate` strategy, Kubernetes creates a new `ReplicaSet` resource, and it simultaneously spins up the required number of pods on the new `ReplicaSet` resource while slowly spinning down the old `ReplicaSet` resource, as evident from the following diagram:

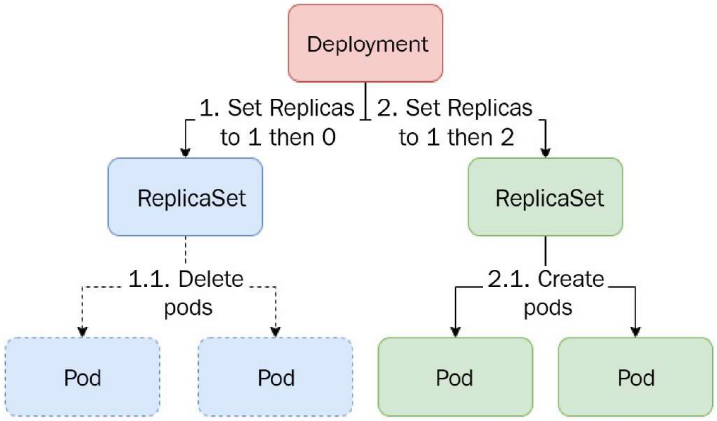

Figure 6.3 – RollingUpdate strategy

`RollingUpdate` is the default deployment strategy. You can use the `RollingUpdate` strategy in most applications, apart from ones that cannot tolerate more than one version of the application at a given time.

Let's update the `nginx` `Deployment` resource using the `RollingUpdate` strategy. We will reuse the standard `nginx-deployment.yaml` file that we used before. Use the following command and see what happens to the `ReplicaSet` resources:

```
$ kubectl apply -f nginx-deployment.yaml && kubectl get replicaset -w
deployment.apps/nginx configured
NAME               DESIRED   CURRENT   READY   AGE
nginx-6799fc88d8   3         3         3       49s
nginx-6889dfccd5   1         1         1       4s
nginx-6799fc88d8   2         2         2       53s
nginx-6889dfccd5   2         2         2       8s
nginx-6799fc88d8   1         1         1       57s
nginx-6889dfccd5   3         3         3       11s
nginx-6799fc88d8   0         0         0       60s
```

As we see, the old `ReplicaSet` resource—`nginx-6799fc88d8`—is being rolled down, and the new `ReplicaSet` resource—`nginx-6889dfccd5`—is being rolled out simultaneously.

The `RollingUpdate` strategy also has two options—`maxUnavailable` and `maxSurge`.

While `maxSurge` defines the maximum number of additional pods we can have at a given time, `maxUnavailable` defines the maximum number of unavailable pods we can have at a given time.

> **Tip**
>
> Set `maxSurge` to `0` if your application cannot tolerate more than a certain number of replicas. Set `maxUnavailable` to `0` if you want to maintain reliability and your application can tolerate more than the set replicas. You cannot specify `0` for both parameters, as that will make any rollout attempts impossible. While setting `maxSurge`, ensure your cluster has spare capacity to spin up additional pods, or the rollout will fail.

Using these settings, we can create different kinds of custom rollout strategies—some popular ones are discussed in the following sections.

Ramped slow rollout

If you have numerous replicas but want to roll out the release slowly, observe the application for any issues, and roll back your deployment if needed, you should use this strategy.

Let's create an `nginx` deployment, `nginx-ramped-slow-rollout.yaml`, using the **ramped slow rollout** strategy:

```
...
spec:
```

```
  replicas: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
...
```

The manifest is very similar to the generic Deployment, except that it contains `10` replicas and a `strategy` section.

The `strategy` section contains the following:

- `type`: `RollingUpdate`
- `rollingUpdate`: The section describing rolling update attributes –`maxSurge` and `maxUnavailable`

Now, let's apply the YAML file and wait for the deployment to completely roll out to `10` replicas using the following commands:

```
$ kubectl apply -f nginx-ramped-slow-rollout.yaml \
&& kubectl rollout status deployment nginx
deployment.apps/nginx configured
...
deployment "nginx" successfully rolled out
```

As we see, the pods have rolled out completely. Let's now update the `Deployment` resource with a different `nginx` image version and see what we get using the following command:

```
$ kubectl set image deployment nginx nginx=nginx:1.16.1 \
&& kubectl get replicaset -w
deployment.apps/nginx image updated
NAME               DESIRED   CURRENT   READY   AGE
nginx-6799fc88d8   10        10        10      3m51s
nginx-6889dfccd5   1         1         0       0s
nginx-6799fc88d8   9         10        10      4m
. . . . . .  . .  . .  .
nginx-6889dfccd5   8         8         8       47s
nginx-6799fc88d8   2         3         3       4m38s
nginx-6889dfccd5   9         9         8       47s
nginx-6799fc88d8   2         2         2       4m38s
nginx-6889dfccd5   9         9         9       51s
nginx-6889dfccd5   10        9         9       51s
nginx-6799fc88d8   1         2         2       4m42s
nginx-6889dfccd5   10        10        10      55s
nginx-6799fc88d8   0         1         1       4m46s
nginx-6799fc88d8   0         0         0       4m46s
```

So, we see two `ReplicaSet` resources here—`nginx-6799fc88d8` and `nginx-6889dfccd5`. While the `nginx-6799fc88d8` pod is slowly rolling down from `10` pods to `0`, one at a time, simultaneously, the `nginx-6889dfccd5` pod is slowly rolling up from `0` pods to `10`. At any given time, the number of pods never exceeds `11`. That is because `maxSurge` is set to `1`, and `maxUnavailable` is `0`. This is a slow rollout in action.

> **Tip**
>
> Ramped slow rollout is useful when we want to be cautious before we impact many users, but this strategy is extremely slow and may only suit some applications.

Let's look at the best-effort controlled rollout strategy for a faster rollout without compromising application availability.

Best-effort controlled rollout

Best-effort controlled rollout helps you roll out your deployment on a best-effort basis, and you can use it to roll out your release faster and ensure that your application is available. It can also help with applications that do not tolerate more than a certain number of replicas at a given point.

We will set `maxSurge` to `0` and `maxUnavailable` to any percentage we find suitable for remaining unavailable at a given time to implement this. It can be specified using the number of pods or as a percentage.

> **Tip**
>
> Using a percentage is a better option since, with this, you don't have to recalculate your `maxUnavailable` parameter if the replicas change.

Let's look at the manifest—`nginx-best-effort-controlled-rollout.yaml`:

```
...
spec:
  replicas: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 0
      maxUnavailable: 25%
...
```

Let's now apply the YAML file and see what we get:

```
$ kubectl apply -f nginx-best-effort-controlled-rollout.yaml \
&& kubectl get replicaset -w
deployment.apps/nginx configured
```

```
NAME                DESIRED   CURRENT   READY   AGE
nginx-6799fc88d8    2         0         0       20m
nginx-6889dfccd5    8         8         8       16m
nginx-6799fc88d8    2         2         1       20m
nginx-6889dfccd5    7         8         8       16m
. . . . . . . . . . . . . . . . .
nginx-6889dfccd5    1         1         1       16m
nginx-6799fc88d8    9         9         8       20m
nginx-6889dfccd5    0         1         1       16m
nginx-6799fc88d8    10        9         8       20m
nginx-6889dfccd5    0         0         0       16m
nginx-6799fc88d8    10        10        10      20m
```

So, we see the `ReplicaSet` resource rolling out such that the total pods are at most `10` at any point, and the total unavailable pods are never more than `25%`. You may also notice that instead of creating a new `ReplicaSet` resource, the `Deployment` resource uses an old `ReplicaSet` resource containing the `nginx:latest` image. Remember when I said the old `ReplicaSet` resource is not deleted when you update a `Deployment` resource?

`Deployment` resources on their own are great ways of scheduling and managing pods. However, we have overlooked an essential part of running containers in Kubernetes—exposing them to the internal or external world. Kubernetes provides several resources to help expose your workloads appropriately—primarily, `Service` and `Ingress` resources. Let's have a look at these in the next section.

Kubernetes Services and Ingresses

It's story time! Let's simplify Kubernetes Services.

Imagine you have a group of friends who love to order food from your restaurant. Instead of delivering each order to their houses separately, you set up a central delivery point in their neighborhood. This delivery point (or hub) is your "service."

In Kubernetes, a Service is like that central hub. It's a way for the different parts of your application (such as your website, database, or other things) to talk to each other, even if they're in separate containers or machines. It gives them easy-to-remember addresses to find each other without getting lost.

The `Service` resource helps expose Kubernetes workloads to the internal or external world. As we know, pods are ephemeral resources—they can come and go. Every pod is allocated a unique IP address and hostname, but once a pod is gone, the pod's IP address and hostname change. Consider a scenario where one of your pods wants to interact with another. However, because of its transient nature, you cannot configure a proper endpoint. If you use the IP address or the hostname as the endpoint of a pod and the pod is destroyed, you will no longer be able to connect to it. Therefore, exposing a pod on its own is not a great idea.

Kubernetes provides the `Service` resource to provide a static IP address to a group of pods. Apart from exposing the pods on a single static IP address, it also provides load balancing of traffic between

pods in a round-robin configuration. It helps distribute traffic equally between the pods and is the default method of exposing your workloads.

`Service` resources are also allocated a static **fully qualified domain name** (**FQDN**) (based on the Service name). Therefore, you can use the `Service` resource FQDN instead of the IP address within your cluster to make your endpoints fail-safe.

Now, coming back to `Service` resources, there are multiple `Service` resource types— `ClusterIP`, `NodePort`, and `LoadBalancer`, each having its own respective use case:

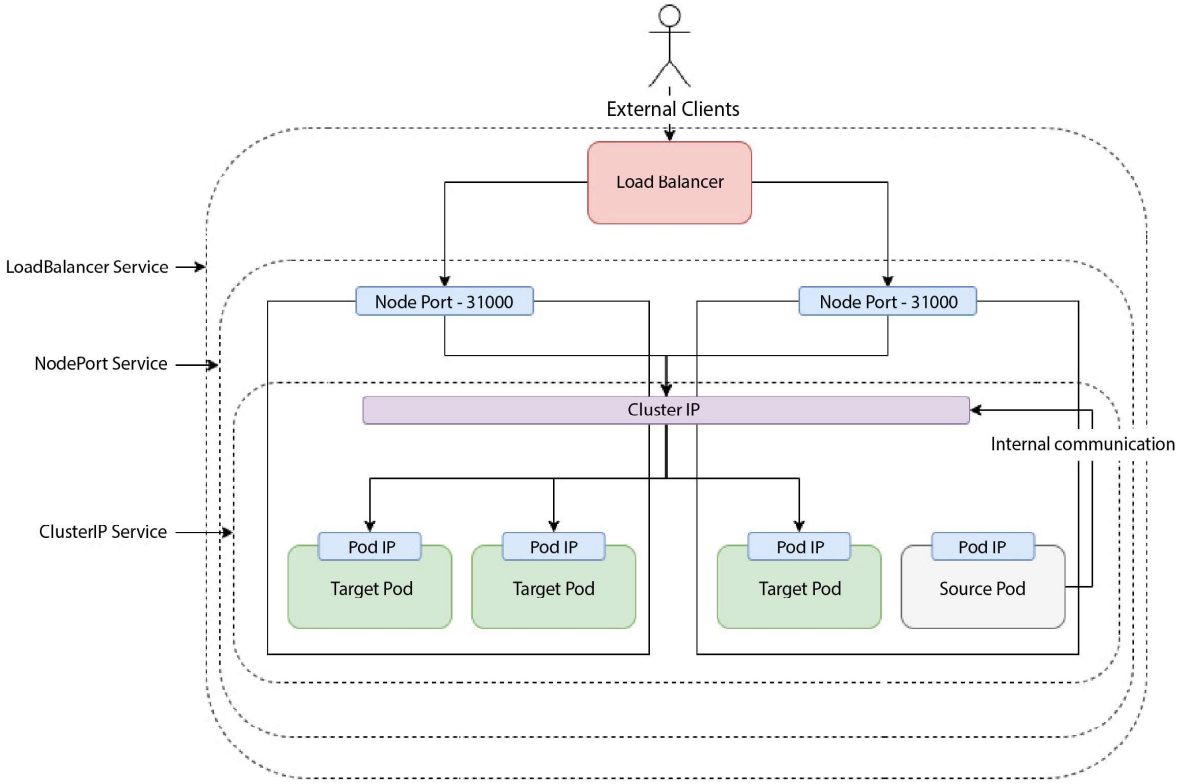

Figure 6.4 – Kubernetes Services

Let's understand each of these with the help of examples.

ClusterIP Service resources

`ClusterIP` `Service` resources are the default `Service` resource type that exposes pods within the Kubernetes cluster. It is not possible to access `ClusterIP` `Service` resources outside the cluster; therefore, they are never used to expose your pods to the external world. `ClusterIP` `Service` resources generally expose backend apps such as data stores and databases—the business and data layers—in a three-tier architecture.

> **Tip**
>
> When choosing between `Service` resource types, as a general rule of thumb, always start with the `ClusterIP` `Service` resource and change it if needed. This will ensure that only the required Services are exposed externally.

To understand `ClusterIP` `Service` resources better, let's create a `redis Deployment` resource first using the imperative method with the following command:

```
$ kubectl create deployment redis --image=redis
```

Let's try exposing the `redis` deployment pods using a `ClusterIP` `Service` resource. To access the resources for this section, `cd` into the following:

```
$ cd ~/modern-devops/ch6/services/
```

Let's look at the `Service` resource manifest, `redis-clusterip.yaml`, first:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: redis
  name: redis
spec:
  ports:
  - port: 6379
    protocol: TCP
    targetPort: 6379
  selector:
    app: redis
```

The `Service` resource manifest starts with `apiVersion` and `kind` as any other resource. It has a `metadata` section that contains `name` and `labels`.

The `spec` section contains the following:

- `ports`: This section includes a list of ports that we want to expose via the `Service` resource:

  A. `port`: The port we wish to expose.

  B. `protocol`: The protocol of the port we expose (TCP/UDP).

  C. `targetPort`: The target container port where the exposed port will forward the connection. This allows us to have a port mapping similar to Docker.

- `selector`: This section contains `labels` based on which pod group is selected.

Let's apply the `Service` resource manifest using the following command and see what we get:

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

Let's run `kubectl get` to list the `Service` resource and get the cluster IP:

```
$ kubectl get service redis
NAME   TYPE       CLUSTER-IP    EXTERNAL-IP  PORT(S)    AGE
redis  ClusterIP  10.12.6.109   <none>       6379/TCP   16s
```

We see a `redis` `Service` resource running with a `ClusterIP` type. But as this pod is not exposed externally, the only way to access it is through a second pod running within the cluster.

Let's create a `busybox` pod in interactive mode to inspect the `Service` resource and run some tests using the following command:

```
$ kubectl run busybox --rm --restart Never -it --image=busybox
/ #
```

And with this, we see a prompt. We have launched the `busybox` container and are currently within that. We will use the `telnet` application to check the connectivity between pods.

Let's telnet the cluster IP and port to see whether it's reachable using the following command:

```
/ # telnet 10.96.118.99 6379
Connected to 10.96.118.99
```

The IP/port pair is reachable from there. Kubernetes also provides an internal DNS to promote service discovery and connect to the `Service` resource. We can do a reverse `nslookup` on the cluster IP to get the `Service` resource's FQDN using the following command:

```
/ # nslookup 10.96.118.99
Server:         10.96.0.10
Address:        10.96.0.10:53
99.118.96.10.arpa name = redis.default.svc.cluster.local
```

As we can see, the IP address is accessible from the FQDN—`redis.default.svc.cluster.local`. We can use the entire domain or parts of it based on our location. The FQDN is formed of these parts: `<service_name>.<namespace>.svc.<cluster-domain>.local`.

Kubernetes uses **namespaces** to segregate resources. You can visualize namespaces as multiple virtual clusters within the same physical Kubernetes cluster. You can use them if many users work in multiple teams or projects. We have been working in the `default` namespace till now and will continue doing so. If your source pod is located in the same namespace as the `Service` resource, you can use `service_name` to connect to your `Service` resource—something like the following example:

```
/ # telnet redis 6379
Connected to redis
```

If you want to call a `Service` resource from a pod situated in a different namespace, you can use `<service_name>.<namespace>` instead—something like the following example:

```
/ # telnet redis.default 6379
Connected to redis.default
```

Some service meshes, such as Istio, allow multi-cluster communication. In that situation, you can also use the cluster name for connecting to the `Service` resource, but as this is an advanced topic, it is beyond the scope of this discussion.

> **Tip**
> Always use the shortest domain name possible for endpoints, as it allows for more flexibility in moving your Kubernetes resources across your environments.

`ClusterIP` Services work very well for exposing internal pods, but what if we want to expose our pods to the external world? Kubernetes offers various `Service` resource types for that; let's look at the `NodePort Service` resource type first.

NodePort Service resources

`NodePort Service` resources are used to expose your pods to the external world. Creating a `NodePort Service` resource spins up a `ClusterIP Service` resource and maps the `ClusterIP` port to a random high port number (default: `30000-32767`) on all cluster nodes. You can also specify a static `NodePort` number if you so desire. So, with a `NodePort Service` resource, you can access your pods using the IP address of any node within your cluster and the `NodePort` of the service.

> **Tip**
> Though it is possible to specify a static `NodePort` number, you should avoid using it. That is because you might end up in port conflicts with other `Service` resources and put a high dependency on config and change management. Instead, keep things simple and use dynamic ports.

Going by the Flask application example, let's create a `flask-app` pod with the `redis Service` resource we created before, acting as its backend, and then we will expose the pod on `NodePort`.

Use the following command to create a pod imperatively:

```
$ kubectl run flask-app --image=<your_dockerhub_user>/python-flask-redis
```

Now, as we've created the `flask-app` pod, let's check its status using the following command:

```
$ kubectl get pod flask-app
NAME        READY   STATUS    RESTARTS   AGE
flask-app   1/1     Running   0          19s
```

The `flask-app` pod is running successfully and is ready to accept requests. It's time to understand the resource manifest for the `NodePort Service` resource, `flask-nodeport.yaml`:

```
...
spec:
  ports:
  - port: 5000
    protocol: TCP
    targetPort: 5000
  selector:
    run: flask-app
  type: NodePort
```

The manifest is similar to the `ClusterIP` manifest but contains a `type` attribute specifying the `Service` resource type—`NodePort`.

Let's apply this manifest to see what we get using the following command:

```
$ kubectl apply -f flask-nodeport.yaml
```

Now, let's list the `Service` resource to get the `NodePort` Service using the following command:

```
$ kubectl get service flask-app
NAME       TYPE       CLUSTER-IP    EXTERNAL-IP  PORT(S)          AGE
flask-app  NodePort   10.3.240.246  <none>       5000:32618/TCP   9s
```

And we see that the type is now `NodePort`, and the container port `5000` is mapped to node port `32618`.

If you are logged in to any Kubernetes node, you can access the `Service` resource using `localhost:32618`. But as we are using Google Cloud Shell, we need to SSH into a node to access the `Service` resource.

Let's list the nodes first using the following command:

```
$ kubectl get nodes
NAME             STATUS   ROLES    AGE   VERSION
gke-node-1dhh    Ready    <none>   17m   v1.26.15-gke.4901
gke-node-71hl    Ready    <none>   17m   v1.26.15-gke.4901
gke-node-zwg1    Ready    <none>   17m   v1.26.15-gke.4901
```

And as we can see, we have three nodes. Let's SSH into the `gke-node-1dhh` node using the following command:

```
$ gcloud compute ssh gke-node-1dhh
```