Defining a Zero-Downtime Deployment

In this lesson, we will look into the definition of a zero-downtime deployment.

WE'LL COVER THE FOLLOWING

Updating a single-replica MongoDB cannot demonstrate true power behind Deployments. We need a scalable service. It's not that MongoDB cannot be scaled (it can), but it is not as straight-forward as an application that was designed to be scalable. We'll jump to the second application in the stack and create a Deployment of the ReplicaSet that will create Pods based on the `vfarcic/go-demo-2` image.

> Zero-downtime deployment is a prerequisite for higher frequency releases.

Looking into the Definition

Let's take a look at the Deployment definition of the API.

```
cat deploy/go-demo-2-api.yml
```

The **output** is as follows.

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: go-demo-2-api
spec:
  replicas: 3
  selector:
    matchLabels:
      type: api
      service: go-demo-2
  minReadySeconds: 1
  progressDeadlineSeconds: 60
  revisionHistoryLimit: 5
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
  template:
    metadata:
      labels:
        type: api
        service: go-demo-2
        language: go
    spec:
      containers:
      - name: api
        image: vfarcic/go-demo-2
        env:
        - name: DB
          value: go-demo-2-db
        readinessProbe:
          httpGet:
            path: /demo/hello
            port: 8080
          periodSeconds: 1
        livenessProbe:
          httpGet:
            path: /demo/hello
            port: 8080
```

We'll skip explaining `apiVersion`, `kind`, and `metadata`, since they always follow the same pattern.

- **Line 5-7:** The `spec` section has a few of the fields we haven't seen before, and a few of those we are familiar with. The `replicas` and the `selector` are the same as what we used in the ReplicaSet from the previous chapter.

- **Line 11:** `minReadySeconds` defines the minimum number of seconds before Kubernetes starts considering the Pods healthy. We put the value of this field to `1` second. The default value is `0`, meaning that the Pods will be considered available as soon as they are ready and, when specified, `livenessProbe` returns OK. If in doubt, omit this field and leave it to the default value of `0`. We defined it mostly for demonstration

purposes.

- **Line 13:** The next field is `revisionHistoryLimit`. It defines the number of old ReplicaSets we can rollback. Like most of the fields, it is set to the sensible default value of `10`. We changed it to `5` and, as a result, we will be able to rollback to any of the previous five ReplicaSets.

- **Line 14:** The `strategy` can be either the `RollingUpdate` or the `Recreate` type. The latter will kill all the existing Pods before an update. `Recreate` resembles the processes we used in the past when the typical strategy for deploying a new release was first to stop the existing one and then put a new one in its place. This approach inevitably leads to downtime. The only case when this strategy is useful is when applications are not designed for two releases to coexist. Unfortunately, that is still more common than it should be. If you're in doubt whether your application is like that, ask yourself the following question. Would there be an adverse effect if two different versions of my application are running in parallel? If that's the case, a `Recreate` strategy might be a good choice and *you must be aware that you cannot accomplish zero-downtime deployments*.

Deployment Strategies

Let's look into a bit of detail of both `Recreate` and `RollingUpdate` strategies.

Recreate Strategy

The `Recreate` strategy is much better suited for our single-replica database. We should have set up the native database replication (not the same as Kubernetes ReplicaSet object), but, that is out of the scope of this chapter.

If we're running the database as a single replica, we must have mounted a network drive volume. That would allow us to avoid data loss when updating it or in case of a failure. Since most databases (MongoDB included) cannot have multiple instances writing to the same data files, killing the old release before creating a new one is a good strategy when replication is absent. We'll apply it later.

RollingUpdate Strategy

The `RollingUpdate` strategy is the default type, for a good reason. It allows us

to deploy new releases without downtime. It creates a new ReplicaSet with zero replicas and, depending on other parameters, increases the replicas of the new one, and decreases those from the old one. The process is finished when the replicas of the new ReplicaSet entirely replace those from the old one.

When `RollingUpdate` is the strategy of choice, it can be fine-tuned with the `maxSurge` and `maxUnavailable` fields. The former defines the maximum number of Pods that can exceed the desired number (set using `replicas`). It can be set to an absolute number (e.g., `2`) or a percentage (e.g., `35%`). The total number of Pods will never exceed the desired number (set using `replicas`) and the `maxSurge` combined. The default value is `25%`.

`maxUnavailable` defines the maximum number of Pods that are not operational. If, for example, the number of replicas is set to 15 and this field is set to 4, the minimum number of Pods that would run at any given moment would be 11. Just as the `maxSurge` field, this one also defaults to **25%**. If this field is not specified, there will always be at least **75%** of the desired Pods.

In most cases, the default values of the Deployment specific fields are a good option. We changed the default settings only as a way to demonstrate better all the options we can use. We'll remove them from most of the Deployment definitions that follow.

The Template

The `template` is the same `PodTemplate` we used before. A best practice is to be explicit with image tags like we did when we set `mongo:3.3`. However, that might not always be the best strategy with the images we're building. Given we employ right practices, we can rely on `latest` tags being stable. Even if we discover they're not, we can remedy that quickly by creating a new `latest` tag. However, We cannot expect the same from third-party images. They must always be tagged to a specific version.

> ⚠ Never deploy third-party images based on `latest` tags. By being explicit with the release, we have more control over what is running in production, as well as what should be the next upgrade.

We won't always use `latest` for our services, but only for the initial Deployments. Assuming that we are doing our best to maintain the `latest` tag stable and production-ready, it is handy when setting up the cluster for the first time. After that, each new release will be with a specific tag. Our automated continuous deployment pipeline will do that for us in one of the next chapters.

In the next lesson, we will create the Deployment defined in this lesson.