We have seen some of the common challenges of managing distributed stateful applications, and a few less-than-ideal workarounds. Next, let's check out the Kubernetes native mechanism for addressing these requirements through the StatefulSet primitive.

Solution

To explain what StatefulSet provides for managing stateful applications, we occasionally compare its behavior to the already familiar ReplicaSet primitive that Kubernetes uses for running stateless workloads. In many ways, StatefulSet is for managing pets, and ReplicaSet is for managing cattle. Pets versus cattle is a famous (but also a controversial) analogy in the DevOps world: identical and replaceable servers are referred to as cattle, and nonfungible unique servers that require individual care are referred to as pets. Similarly, StatefulSet (initially inspired by the analogy and named PetSet) is designed for managing nonfungible Pods, as opposed to ReplicaSet, which is for managing identical replaceable Pods.

Let's explore how StatefulSets work and how they address the needs of stateful applications. Example 11-1 is our random-generator service as a StatefulSet. [1]

Example 11-1. Service for accessing StatefulSet

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: rg                                  ❶
spec:
  serviceName: random-generator             ❷
  replicas: 2                               ❸
  selector:
    matchLabels:
      app: random-generator
  template:
    metadata:
      labels:
        app: random-generator
    spec:
      containers:
      - image: k8spatterns/random-generator:1.0
        name: random-generator
        ports:
        - containerPort: 8080
```

1 Let's assume we have invented a highly sophisticated way of generating random numbers in a distributed RNG cluster with several instances of our service as nodes. Of course, that's not true, but for this example's sake, it's a good enough story.