Orchestration Basics

We've been using the term 'orchestration' throughout the course. In this lesson, you'll learn what orchestration tools are and when to use them.

In this chapter, we saw some tools (the `--restart-mode` switch, the *docker stats* command) that help keep your containers running and keep an eye on them. This is good for starters. However, if you rely on containers and/or have high workloads, using such tools and automating their use will become tedious.

Imagine coding calls to create or update containers, and to remove them when necessary. How will you update your containers when you publish new versions? Since running containers is cheap, you may want to perform rolling updates so that your application isn't down during updates. In a rolling update, a new container is started with the newest image, and once it is ready, users are routed to it, then the old container is removed. Imagine coding this; it's going to be tedious.

Now factor in the possibility that several Docker hosts may need to run your containers either for large workloads, scaling out, or simply good reliability. In such cases, you will need to set up a reverse proxy and have it route users to the appropriate containers. You really don't want to maintain script files that do this.

Good news; you don't need to worry about that. Those problems are solved by orchestration tools. When such needs arise, it will be time to use an orchestration tool like Docker Swarm or Kubernetes.

Kubernetes or Docker Swarm receive your orders and apply them. You create a cluster of servers (a single server is fine also), then you feed your Kubernetes or Docker Swarm with a file that states which containers you want, how to expose them to the outside world, and how many containers should be run for each image. Your orchestrator will make sure that happens.

Here is an example Kubernetes file:

```
apiVersion: apps/v1beta1
kind: Deployment
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: core-server
    spec:
      containers:
      - image: learnbook/aspnetcore-server:1.1
        ports:
        - containerPort: 80
        env:
        - name: ASPNETCORE_ENVIRONMENT
          value: "Release"
---
apiVersion: v1
kind: Service
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: core-server
```

Although it looks complex, it's not that hard. Sending it to the cluster by using the *kubectl* command makes sure that I get my application running as two containers exposed over the internet with a load balancer. Nice, isn't it?

Better yet, need to upgrade or run more or fewer containers? Update the file stating your new needs, feed it to the orchestrator, and it will take care of things like rolling upgrades. Look at my modified file:

```
apiVersion: apps/v1beta1
kind: Deployment
spec:
  replicas: 10
  template:
    metadata:
      labels:
        app: core-server
    spec:
      containers:
      - image: learnbook/aspnetcore-server:1.2
        ports:
        - containerPort: 80
        env:
        - name: ASPNETCORE_ENVIRONMENT
          value: "Release"
---
apiVersion: v1
```

```
kind: Service
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: core-servereturn 0;
}
```

I only changed the *replicas* and *image* values in order to ask for **10** containers and have them run a more recent **1.2** image. I simply send it to the cluster, again using the *kubectl* command, and Kubernetes proceeds with the operations necessary to reach that state considering the existing state. It will:

1. Start **10** containers running the 1.2 image.
2. Wait for **2** containers to be ready.
3. Route the users to the **2** containers that are ready.
4. Stop and remove the previous two containers that run the **1.1** image.
5. As the **8** more containers become available, route users to them.
6. Load balance the incoming traffic to the **10** containers.

This is probably something you wouldn't dream of if you were managing actual servers, even if they were virtualized. You took a huge step forward moving your application into containers and moving those containers to an orchestrator is another giant step forward.

Orchestration is out of the scope for this course, so I won't continue here.

With that, our course comes to an end. We hope that you enjoyed the course and quickly become proficient enough with containerizing your applications and Docker. See you in the next course!