CHAPTER 7

Docker Swarms

In the previous chapter, I demonstrated how to manage a complex application on a single server using Docker Compose. In this chapter, I explain how to scale up applications so they run on multiple servers, using a *Docker swarm*.

A Docker swarm is a cluster of servers that run the Docker engine. Each server in a swarm is known as a node, of which there are two types. Manager nodes are used to orchestrate *services*, which are the desired state for containerized applications, such as the example ASP.NET Core MVC application or MySQL. A service describes how many containers should be created and which nodes in the cluster should run them. This is known as the service's desired state.

Manager nodes perform their orchestration by managing the other type of node: worker nodes. A worker is responsible for running the containers delivering the functionality specified by the service. When a container or a worker node fails, the manager node will automatically detect the change and try to return to the desired state by starting additional containers on other nodes. To make this process simpler, a Docker swarm supports software-defined networks that can span the nodes in a cluster and routes requests between the containers connected to the network. Table 7-1 puts Docker swarms in context.

SWARMS REQUIRE LINUX

The clustering of nodes in a swarm is achieved using features that are specific to Linux. At the time of writing, only Linux servers can be clustered together to create a swarm. Neither Windows nor macOS supports the cluster features and cannot be used for the examples in this chapter, although Microsoft has plans to introduce the required features for Windows.

If you don't have your own Linux servers, you can create Linux virtual machines to see how swarms work, either locally or on public clouds such as Microsoft Azure or Amazon Web Services. Many public cloud services also have direct support for swarms, such as the Azure Container Service.

Alternatively, you can use Docker Cloud, which is a service provided by Docker that orchestrates the deployment and management of Docker services on public cloud services, including Azure and AWS. The Docker Cloud service is a paid-for service and requires an account on at least one of the cloud providers. There are free tiers available, but they place limits on the number of nodes you can include in your swarm.

A. Freeman, *Essential Docker for ASP.NET Core MVC*, DOI 10.1007/978-1-4842-2778-7_7

Table 7-1. *Putting Swarms in Context*

Question	Answer
What is it?	A Docker swarm is a cluster of servers that run the containers in an application in a coordinated manner.
Why is it useful?	A swarm allows an application to scale beyond a single server, making an application more tolerant to the failure of individual containers.
How is it used?	A swarm consists of manager and worker nodes. The manager nodes are responsible for managing the containers in the application, deciding which workers will run them, and ensuring that they are running.
Are there any pitfalls or limitations?	It is important to exclude components that receive external network connections (such as load balancers) from the swarm; otherwise, they will be subject to the ingress load balancing feature, which is described in this chapter.
Are there any alternatives?	There are several alternatives to Docker swarms. The most successful are Kubernetes (`https://kubernetes.io`), which was originally developed by Google, and Mesos (`http://mesos.apache.org`), which is an Apache Software Foundation Project. Both Kubernetes and Mesos provide support for working with Docker containers. Kubernetes, in particular, is well-established and well-supported. Docker swarms are relatively new but have the advantage of being integrated into the Docker runtime and have the backing of Microsoft, which is important for ASP.NET Core MVC projects.

Table 7-2 summarizes this chapter.

Table 7-2. *Chapter Summary*

Problem	Solution	Listing
Create a swarm	Run `docker swarm init` on the manager node and then run the command shown in the output on each of the workers	1–3
Differentiate worker nodes	Assign the node a label	4
Create a network in the swarm	Use the `docker network create` command with the `-d` argument set to `overlay`	5
Deploy a service to a swarm	Use the `docker service create` command	6, 11–14
Examine a service	Use the `docker service ps` or `docker service ls` command	7, 15, 16, 22, 38
Modify a service	Use the `docker service update` command	8–10, 23, 30, 33
Remove a service	Use the `docker service rm` command	19
Deploy services using a compose file	Use the `deploy` keyword to describe the service in the compose file and then use the `docker stack deploy` command	20, 21, 37

(*continued*)

Table 7-2. (*continued*)

Problem	Solution	Listing
Change the number of containers in a service	Use the `docker service scale` command	26, 27, 34
Change the status of a node in the cluster	Use the `docker node update` command	28, 29
Remove the services described in a compose file	Use the `docker stack rm` command	35
Deploy one container in a service on each node in a swarm	Set `mode` to `global` in the compose file	36

Preparing for This Chapter

This chapter depends on the `ExampleApp` MVC project created in Chapter 3 and modified in the chapters since. If you don't want to work through the process of creating the example, you can get the project as part of the free source code download that accompanies this book. See the `apress.com` page for this book.

To ensure that there is no conflict with examples from previous chapters, run the commands shown in Listing 7-1 to remove the Docker containers, networks, and volumes. Ignore any errors or warnings these commands produce.

Listing 7-1. Removing the Containers, Networks, and Volumes

```
docker rm -f $(docker ps -aq)
docker network rm $(docker network ls -q)
docker volume rm $(docker volume ls -q)
```

Preparing the Swarm

In this chapter, I create a swarm that contains five nodes or servers. Working with a swarm means referring to the servers in the cluster by name, and to make the examples easier to follow, the servers are described in Table 7-3. The configuration that I have chosen for my swarm is typical for a small cluster. As the number of worker nodes increases, a swarm should include additional manager nodes.

■ **Note** The `manager` node in my swarm runs Ubuntu. The worker nodes all run CoreOS Container Linux, which is a lightweight Linux distribution specifically intended for running application containers and that includes Docker as part of its standard build. CoreOS is supported by the main cloud platforms, including Microsoft Azure, Amazon Web Services, and Google Cloud, and it makes a good target operating system for working with Docker swarms. See `http://coreos.com` for details.

Table 7-3. *The Hosts in the Example Swarm*

Name	Description
`manager`	This is the manager node, which will be responsible for orchestrating the services in the swarm. It is also where I copied the `ExampleApp` folder that contains the example application and the configuration files and is where almost all the commands in this chapter are run. This server will also run the load balancer that will receive HTTP requests and distribute them to the containers running the MVC application.
`dbhost`	This is a worker node that will be dedicated to running the MySQL container.
`worker1`	This is a worker node that will run MVC application containers.
`worker2`	This is a worker node that will run MVC application containers.
`worker3`	This is a worker node that will run MVC application containers.

Creating the Swarm

A swarm is created on the manager node, which generates a key that can then be used by the workers to join the swarm. Run the command shown in Listing 7-2 on the manager node to create a swarm.

Listing 7-2. Creating a Swarm

```
docker swarm init
```

The output gives you the instructions for associating the other nodes with the swarm, with the command you must run on each node in the swarm marked in bold.

```
Swarm initialized: current node (u9np4ffl3aqao9c3c2bnuzgqm) is now a manager.

To add a worker to this swarm, run the following command:

docker swarm join \
    --token SWMTKN-1-61tskndg374fkvoa7fl0c8d57w2zvku9hzvqpfmojqhw51dlj9-
c1cwiu06s4bfjcj0ekf9lh8uv \
    172.16.0.5:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the
instructions.
```

You will see a different token and a different network value or format of network address based on the configuration of your manager node. Run the command indicated in the output on each worker in your swarm. As the command runs successfully, you will see the following message displayed on each worker:

```
...
This node joined a swarm as a worker
...
```

This is the only command that is run on the worker nodes. Once a node is part of a swarm, it is controlled using the manager node. All the remaining examples in this chapter are carried out on the manager node.

When you have added all the nodes to the swarm, run the command in Listing 7-3 on the manager to examine the swarm.

Listing 7-3. Examining the Nodes in the Swarm

```
docker node ls
```

The output from this command will show all the nodes in the swarm (the asterisk indicates the node you are working on).

```
ID                            HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
136uyxxgnldhurrgtj2o9cr0h *   manager   Ready   Active        Leader
e7wwq0oghzx16g9jkz9642nws     dbhost    Ready   Active
gmpyip1ms88kiu5inlwytz2qd     worker2   Ready   Active
o479m45qwi6vlq8pmgas21e96     worker3   Ready   Active
yl9ajlaku5du3hpmsbaf33dyh     worker1   Ready   Active
```

■ **Tip** If you want to leave a swarm, then run the `docker swarm leave` command on the node. For manager nodes, use the `docker swarm leave --force` command.

Labeling Swarm Nodes

Nodes in a swarm can be assigned labels that can be used to control the types of container they run. For this example, I am going to use labels to differentiate between the node that will be responsible for running the database and the nodes that will be responsible for the MVC application. Run the commands shown in Listing 7-4 on the manager node to assign labels to the worker nodes in the swarm, assigning them the `type` label and setting its value to `mvc`. Labels are arbitrary and can be assigned in any way that helps make sense of your swarm and the application running on it: the `type` label I have used here and the value `mvc` have no special meaning to Docker.

Listing 7-4. Assigning Labels to the Nodes in the Swarm

```
docker node update --label-add type=mvc worker1
docker node update --label-add type=mvc worker2
docker node update --label-add type=mvc worker3
```

Manually Deploying Services on a Swarm

The simplest way to use a swarm is to describe the application in a compose file, as demonstrated in the "*Deploying to a Swarm Using a Compose File*" section. But to understand what happens behind the scenes, it helps to go through the process of deploying an application manually, even though this is a process that requires careful attention to detail to type complex commands correctly. (For this reason, it is something that you should do only to understand how swarms work and not in production.)

Creating the Software-Defined Networks

Docker supports software-defined networks that connect the nodes in a swarm, allowing containers to seamlessly communicate across the swarm. To create the network for the example application, run the command shown in Listing 7-5 on the manager node.

Listing 7-5. Creating the Software-Defined Networks

```
docker network create -d overlay swarm_backend
```

The `-d` argument specifies the type of software-defined network, which is set to `overlay` when creating a network that will span the servers in the swarm. I have prefixed the network name with `swarm_` to differentiate it from earlier examples; this is a not a requirement for real projects.

■ **Tip** There is only one network in this example because the communication between the load balancer and the MVC application will occur outside the swarm, as explained in the "*Creating the Load Balancer*" section.

The effect of the command in Listing 7-5 is to create a software-defined network that connects together all the nodes in the swarm, as illustrated by Figure 7-1.

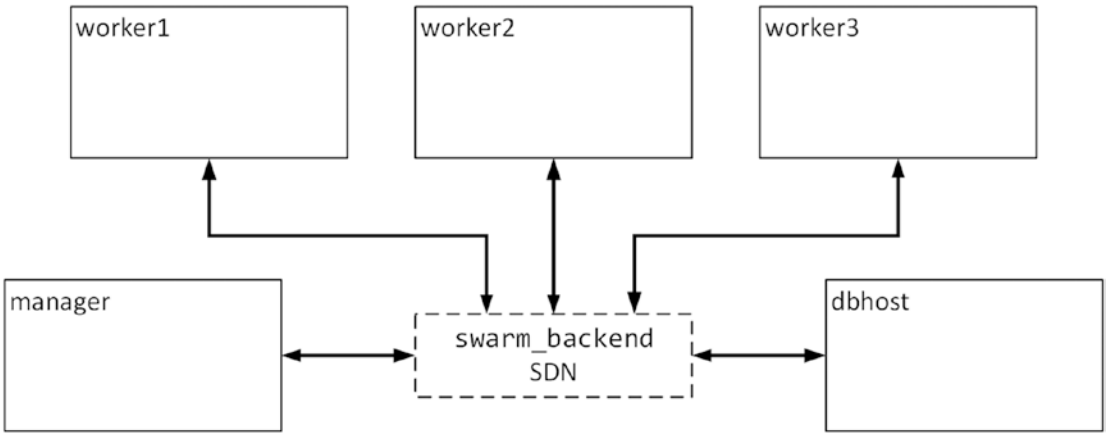

Figure 7-1. *Creating a swarm network*

Creating the Database Service

Deploying and configuring a database in a swarm can be awkward because the nature of databases is somewhat opposed to the nature of swarms. The services that can take most advantage of the swarm features are the ones that can easily be replicated by starting new containers and that can be easily moved between nodes in the cluster.

Databases don't fit well into this model: there tends to be a small number of database services in an application (often just one instance), they rely on data files that cannot be easily transported to other nodes in the cluster, and there can be substantial initialization and shutdown processes that preserve data integrity.

For these reasons, you may prefer to set up your database server outside of the swarm and configure your MVC application containers to make queries over the physical network that connects your servers. Since this is a book about Docker, I am going to demonstrate how to deploy the database within the swarm cluster, even though there will be only one container in the service and it will be deployed to a specific node. This will let me demonstrate some useful Docker features, even though it is not a suitable approach for all projects.

Run the command shown in Listing 7-6 on the manager node to create the MySQL service. (Enter the command on a single line.)

Listing 7-6. Creating the Database Service

```
docker service create --name mysql
    --mount type=volume,source=productdata,destination=/var/lib/mysql
    --constraint "node.hostname == dbhost" --replicas 1 --network swarm_backend
    -e MYSQL_ROOT_PASSWORD=mysecret -e bind-address=0.0.0.0 mysql:8.0.0
```

Services are created using the `docker service create` command, with the arguments telling Docker how the service should be deployed. For MySQL, I want to create a single container with a persistent data volume on the `dbhost` swarm node, using the arguments described in Table 7-4.

Table 7-4. *The Arguments Used to Create the MySQL Service*

Name	Description
`--name`	This argument sets the name for the service, which is `mysql` in this example.
`--mount`	This argument is used to specify the volumes that will be used by the containers created by the service. In this example, a volume called `productdata` will be used to provide content for the `/var/lib/sql` directory.
`--constraint`	This argument is used to restrict the nodes in the swarm where containers for the services can run, as described in the "Using Swarm Constraints" sidebar. In this example, containers for this service can run only on the node whose host name is `dbhost`.
`--replicas`	This argument is used to specify the desired number of containers that should be run for this service. In this example, one container is specified, which is typical for databases.
`--network`	This argument is used to specify the software-defined networks to which containers created for this service will be attached. In this example, the containers will be connected to the `swarm_backend` network.
`-e`	This argument is used to specify environment variables that will be used to configure containers created for this service. The example uses the same MySQL variables introduced in Chapter 5.

Volumes are handled differently in swarms, and care must be taken. The `--mount` argument used in Listing 7-6 will create a new volume called `productdata` and use it for the container specified in the service. But the volume is specific to the swarm node where the service starts the container and will not migrate if the container is stopped and moved to another node in the swarm.

The constraint in the command in Listing 7-6 means that the manager can achieve the desired state for the `mysql` service only by creating a container on the `dbhost` node. Only one replica has been specified, which means that a single database container is started on `dbhost`, connected to the `swarm_backend` network, as shown in Figure 7-2.

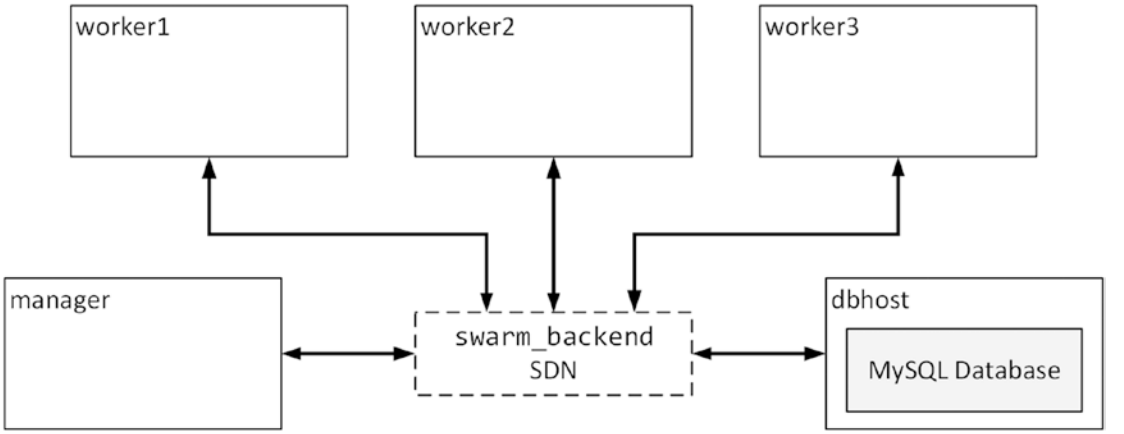

Figure 7-2. *Starting the database service*

It can take a while for the database service to start, as the image is downloaded from the Docker Hub to the worker node. Keep track of the progress of the service by running the command shown in Listing 7-7 on the manager node.

Listing 7-7. Monitoring a Docker Service

```
docker service ps mysql
```

The `docker service ps` command lists the containers that are running in the swarm for a service. As the service is being prepared, you will see output like this:

```
...
ID            NAME     IMAGE        NODE    DESIRED STATE  CURRENT STATE
wzly7bxjxaqg  mysql.1  mysql:8.0.0  dbhost  Running        Preparing
...
```

The `DESIRED STATE` column shows the state that Docker is working toward for the container, while the `CURRENT STATE` column shows how it is progressing. Once the container has started, the output of the `docker service ps` command will show that the container is in the target state, like this:

```
...
ID            NAME     IMAGE        NODE    DESIRED STATE  CURRENT STATE
wzly7bxjxaqg  mysql.1  mysql:8.0.0  dbhost  Running        Running
...
```

USING SWARM CONSTRAINTS

Most real projects will be deployed into clusters where some nodes have a specific role, such as running the database, typically because they have more capable or specialized hardware.

When you deploy a service into a swarm, you can control which nodes will be assigned containers by specifying constraints. The manager will only assign containers to nodes that meet the constraints specified for the service.

There are several different ways that a service can be constrained. The first is they can be deployed to a specific node, identified using its host name. This is the approach I took with the database in Listing 7-6:

```
...
--constraint "node.hostname == dbhost"
...
```

If you want to deploy a service so that it runs only on nodes of a specific type, then you can use a constraint like this:

```
...
--constraint "node.role == manager"
...
```

You can also restrict deployment to nodes that have been assigned a specific label. I assigned labels to three of the worker nodes in Listing 7-4, and I use this label when deploying the service for the MVC application in Listing 7-14, using a constraint like this:

```
...
--constraint "node.labels.type==mvc"
...
```

Labels are the most flexible way to constrain a service because you can assign any label you need to any label in the swarm.

Preparing the Database

When I described the application using a Docker Compose file in Chapter 6, I was able to include a container that was dedicated to initializing the database, after which it would exit. It is possible to arrange this kind of setup in a swarm, but there is a more interesting feature available that provides useful insight into how swarms function. To prepare the database, I am going to temporarily map the database's port 3306 so that it can be accessed through the host operating system and use this to perform the initialization. Bear with me; this is more interesting than it may sound.

Run the command shown in Listing 7-8 on the manager node to update the database service. (The command is run on the manager node, even though the container is running on a worker. Remember that the manager node is always used to configure the services running on a cluster.)

Listing 7-8. Updating the Database Service

```
docker service update --publish-add 3306:3306 mysql
```

The `docker service update` command is used to change the configuration of services after they have been created. You can change almost any aspect of the service including defining new mapped ports with the `--publish-add` argument. This command makes port 3306 available outside of the swarm for the `mysql` service. Docker applies the change across the swarm, ensuring consistency across all the containers that are part of the service.

When you expose a port from a service, Docker maps it to the host operating systems on all the nodes in the swarm, using a feature called *ingress load balancing*. This means that *any* request to port 3306 received on *any* node in the swarm will be received by Docker and sent to one of the containers in the service whose port was mapped.

For the `mysql` service, there is only a single container, which is running on the `dbhost` node, and so any request sent to port 3306 on the `manager`, `worker1`, `worker2`, `worker3`, and `dbhost` nodes will be received by MySQL on port 3306 inside the container running on the `dbhost` node. When there are multiple containers in a service, Docker will load balance the requests so that they are distributed between the containers.

The ingress load balancing feature simplifies the process of preparing the database. Run the commands shown in Listing 7-9 in the `ExampleApp` folder on the manager node.

Listing 7-9. Preparing the Database in the Swarm

```
dotnet restore
dotnet run --INITDB=true
```

When the .NET application runs, the `INITDB` argument selects the database initialization mode, rather than starting ASP.NET Core. Since all the nodes in the swarm map port 3306 to the database service, it doesn't matter which host name is used to configure the database. The application defaults to `localhost` when no other host name is supplied, which means that the connection from Entity Framework Core is made to port 3306 on the manager node, and the swarm ensures that it is channeled to port 3306 on the `dbhost` node so that it can be received by MySQL, as illustrated by Figure 7-3.

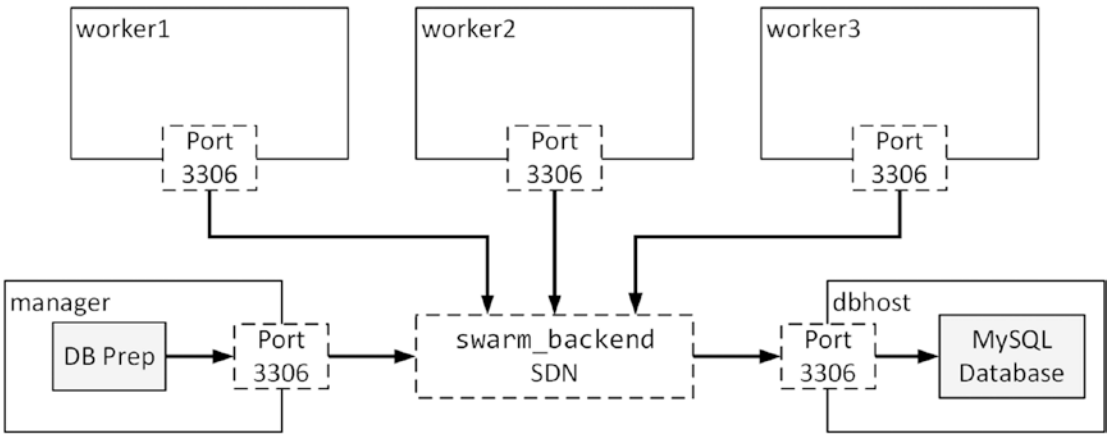

Figure 7-3. *Using ingress load balancing to prepare the database*

It doesn't matter which host in the cluster the database preparation application connects to because any request to port 3306 on any node will be channeled to the container running on `dbhost`, ensuring that the database is correctly prepared for the application. Put another way, a connection sent to port 3306 on the `manager` node is effectively the same as a connection to port 3306 on the `dbhost` node.

The application will produce the following results, showing that the database migrations have been applied to create the schema and that the seed data has been added:

```
...
Preparing Database...
Applying Migrations...
Creating Seed Data...
Database Preparation Complete
...
```

■ **Tip** If you see an error when you use the `dotnet run` command, then wait a couple of minutes to give Docker time to update the service so that port 3306 is available and try again. If you see an error telling you that the `Products` table already exists, then you may have already initialized the database: Entity Framework Core migrations have to be applied only once. If you want to repeat the initialization, delete the `productdata` volume on the `dbhost` worker node, restart the `mysql` service, expose port 3306, and run the commands in Listing 7-9 again.

Once you have prepared the database, run the command shown in Listing 7-10 on the manager node to disable ingress load balancing for port 3306 so that the database returns to being accessible only through the `swarm_backend` software-defined network.

Listing 7-10. Disabling the Port Mapping

```
docker service update --publish-rm 3306:3306 mysql
```

Creating the MVC Application Service

The MVC application is better suited to deployment in a swarm than the database: the application is self-contained, it doesn't matter which nodes run the application, and multiple instances of the MVC application can be created without conflicting with one another.

The manager node in a swarm doesn't distribute the images required for a service to the worker nodes, which means that the image used to create a service must be published to a repository, such as the Docker Hub, so that the nodes in the swarm can get the data they require to create containers.

Run the commands shown in Listing 7-11 in the manager node's `ExampleApp` folder to prepare the application for deployment.

Listing 7-11. Preparing the MVC Application for Deployment as a Service

```
dotnet publish --framework netcoreapp1.1 --configuration Release --output dist
docker build . -t apress/exampleapp:swarm-1.0 -f Dockerfile
```

You will need to change the name of the image created by the `docker build` command to replace `apress` with your Docker Hub account. So, if your account name is `bobsmith`, then your image tag should be `bobsmith/exampleapp:swarm-1.0`.

The `dotnet publish` command ensures that recent changes made to the MVC application will be included in the Docker image that is created by the `docker build` command.

The tag for the MVC application image includes a variation, which is `swarm-1.0`. (You don't have to include `swarm` in the tags for your images. I have done this so that the image for this example can coexist with those from earlier chapters.)

■ **Tip** It is always a good idea to use specific versions of images when creating services rather than relying on the `:latest` version, which may change unexpectedly during the life of the application. Later in the chapter, I explain how to upgrade an application that is used in a service, which also requires an image tag with a version.

Run the commands shown in Listing 7-12 in the `ExampleApp` folder of the `manager` node to authenticate with the Docker Hub using the credentials you created in Chapter 4.

Listing 7-12. Authenticating with Docker Hub

```
docker login -u <yourUsername> -p <yourPassword>
```

Run the command shown in Listing 7-13 on the manager node to push the image to the repository. Don't forget to change the name of the image for the `docker push` command to replace `apress` with your Docker Hub account.

Listing 7-13. Pushing the MVC Image to the Repository

```
docker push apress/exampleapp:swarm-1.0
```

Once the image has been pushed to the repository, run the command shown in Listing 7-14 on the manager node to create the service for the MVC application. Enter the command on a single line.

Listing 7-14. Creating the MVC Application Service

```
docker service create --name mvcapp --constraint "node.labels.type==mvc"
    --replicas 5 --network swarm_backend -p 3000:80
    -e DBHOST=mysql apress/exampleapp:swarm-1.0
```

This command tells Docker to create a service for the MVC application that consists of five containers, which can be run only on nodes that have been assigned the `type` label with a value of `mvc`. Access to the containers in the service will be through the port published using the `-p` argument, which configures the ingress load balancing feature described in the previous section.

Notice that the `DBHOST` environment variable, which specifies the name of the database server for the Entity Framework Core connection string, is set to `mysql`. Within a swarm, Docker creates host names that provide access to the ingress load balancing feature for each service that is created. This means Entity Framework Core can open a connection to `mysql` and rely on Docker to channel that connection to one of the containers that provides the `mysql` service. There is only one container in the `mysql` service, but Docker will load balance between containers if there is more than one available.

The complete set of arguments used to create the MVC application service is described in Table 7-5.

Table 7-5. *The Arguments Used to Create the MVC Service*

Name	Description
`--name`	This argument is used to set the name for the service, which is `mvcapp` in this example.
`--constraint`	This argument is used to restrict the nodes where containers for this service will run, as described in the "*Using Swarm Constraints*" sidebar. For this example, the containers will run only on the nodes assigned the `mvc` label in Listing 7-4.
`--replicas`	This argument specifies the desired number of containers that will run for this service. For this example, five containers are specified.
`--network`	This argument specifies the software-defined networks that the containers for this service will be connected to. For this example, the containers will be connected to the `swarm_backend` service.
`-p`	This argument exposes a port inside the containers created for this service to ingress load balancing features. Requests sent to the specified port will be directed to one of the service containers. In this example, requests sent to port 3000 on any node in the cluster will be directed to port 80 inside one of the `mvc` containers.
`-e`	This argument sets an environment variable when containers are created for the service. In this example, the `DBHOST` variable is set in order to provide the host name that Entity Framework Core needs to connect to the database. The host name is set to the name of the database service, which takes advantage of the ingress load balancing feature.

It will take Docker a moment to set up the containers. You can check the progress by running the command shown in Listing 7-15 on the manager node.

Listing 7-15. Checking Service Status

```
docker service ls
```

The results from the `docker service ls` command show a summary of the services that have been created on the swarm and the status of each of them. When Docker has created all the containers required for the MVC application service, the REPLICAS column in the `docker service ls` output will show `5/5`, indicating that Docker has created all five containers and has reached the desired state for the service, as follows:

```
ID            NAME    MODE        REPLICAS  IMAGE
4kf7xsym2vc5  mvcapp  replicated  5/5       apress/exampleapp:swarm-1.0
j2v5gdqzu1di  mysql   replicated  1/1       mysql:8.0.0
```

Docker is responsible for allocating the containers required for the service to worker nodes. Run the command shown in Listing 7-16 on the manager to see which nodes in the containers within the service have been deployed to.

Listing 7-16. Examining the Containers in a Service

```
docker service ps mvcapp
```

The `docker service ps` command produces a list of all the containers running in a service. Here is the output from my setup, formatted to fit on the page and with only the most useful columns shown:

```
ID             NAME      NODE     DESIRED STATE  CURRENT STATE
7jd1yop74mxfy  mvcapp.1  worker1  Running        Running
cse2cyutpvxh9  mvcapp.2  worker2  Running        Running
5r9y8OOjugey8  mvcapp.3  worker1  Running        Running
cktdzfOypzcpt  mvcapp.4  worker3  Running        Running
5esdl2d3jd1ee  mvcapp.5  worker2  Running        Running
```

In this example, there are five containers running on three workers, which means that some worker nodes are running multiple containers, as illustrated in Figure 7-4.

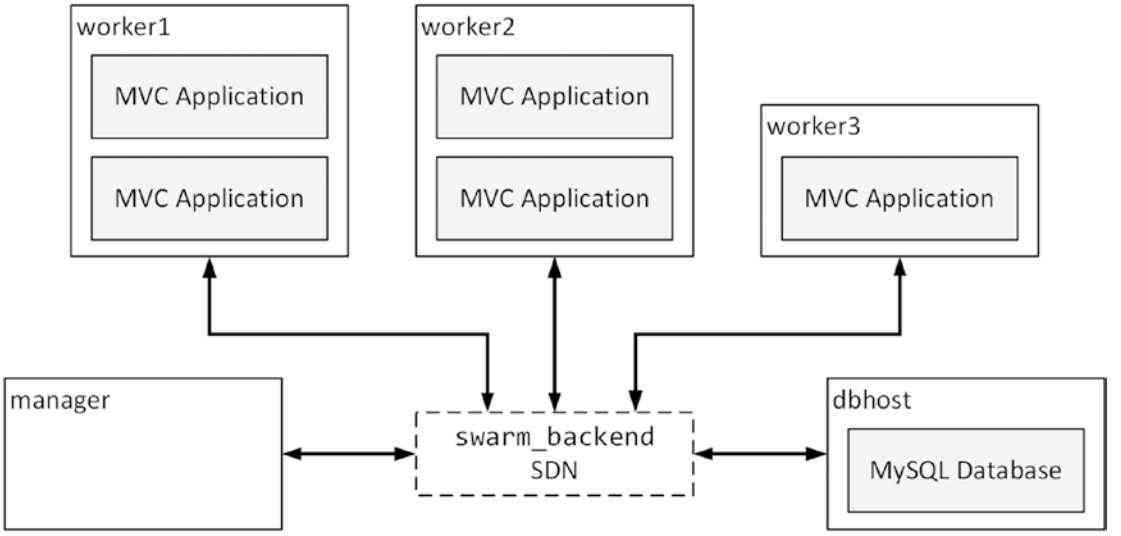

Figure 7-4. *Deploying the ASP.NET Core MVC application service*

If you reboot a worker node, disconnect it from the network, or just stop a container on a worker, the swarm manager will return the service to its desired state by creating additional containers on the other nodes.

Testing the MVC Application Service

The configuration of the MVC application service included a port mapping so that HTTP requests received on port 3000 of any node in the swarm will be sent to port 80 inside one of the MVC containers. Since there are multiple MVC containers, the ingress load balancing feature will distribute the requests between them.

You can see how the requests are distributed by opening a new browser window and requesting the URL for port 3000 on any of the swarm nodes, including the manager node. The URL in my setup is `http://manager:3000`. Keep reloading the page and you will see that the container name included in the response will change, as shown in Figure 7-5. Docker doesn't make any promises about how requests are distributed, and you may have to reload the page a few times before you see a different container name. If you don't see any change in the name, then wait a few minutes and reload the page, try starting a second browser, or make a request from a different IP address.

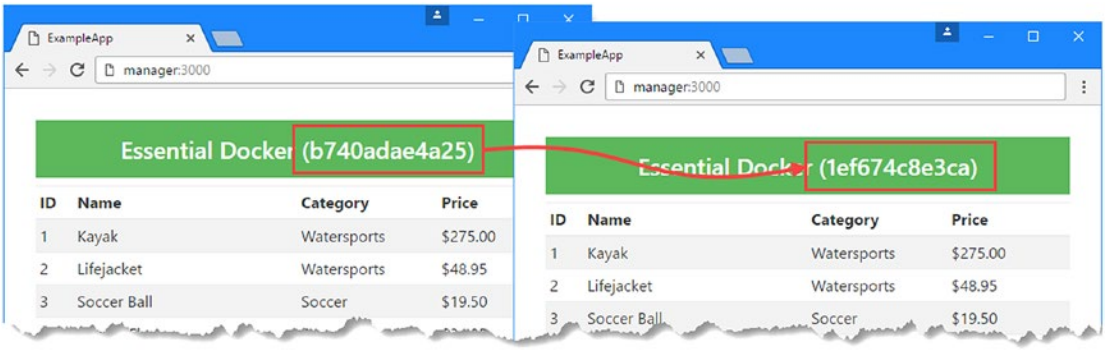

Figure 7-5. *Testing the MVC service through the mesh routing feature*

Creating the Load Balancer

Although you could rely on the ingress load balancing feature to handle HTTP requests from users, the convention is to deploy a load balancer so that there is a single point of entry to the application from the outside world.

It is important not to deploy the load balancer as a service in the swarm since that would mean the Docker ingress load balancing system would receive the HTTP requests from the outside world and then distribute them over the software-defined network to one or more containers, which would use ingress load balancing to target one of the MVC containers.

Instead, I am going to run the HAProxy application I used in earlier examples in a separate container on the manager node. This container will use a regular port mapping to the host operating system to receive HTTP requests and forward them to the MVC application containers through the swarm mesh network. Listing 7-17 shows the change required to the `haproxy.cfg` file to configure the load balancer.

Listing 7-17. Configuring the Load Balancer in the haproxy.cfg File

```
defaults
    timeout connect 5000
    timeout client  50000
    timeout server  50000

frontend localnodes
    bind *:80
    mode http
    default_backend mvc

backend mvc
    mode http
    balance roundrobin
    server mvc1 manager:3000
```

There are lots of ways to configure the load balancer, but the approach I use is to forward requests to the ingress load-balanced port on the local machine so that HAProxy doesn't have to be configured with the details of the worker nodes. Run the command shown in Listing 7-18 on the manager node to create a new load balancer container that uses the configuration file from Listing 7-17. (Enter the command on a single line.)

Listing 7-18. Creating and Starting the Load Balancer Container

```
docker run -d --name loadbalancer
   -v "$(pwd)/haproxy.cfg:/usr/local/etc/haproxy/haproxy.cfg"
   -p 80:80 haproxy:1.7.0
```

Adding the load balancer completes the application by creating a container outside of the swarm that uses the ingress load balancing feature to distribute between the MVC containers, as illustrated in Figure 7-6.

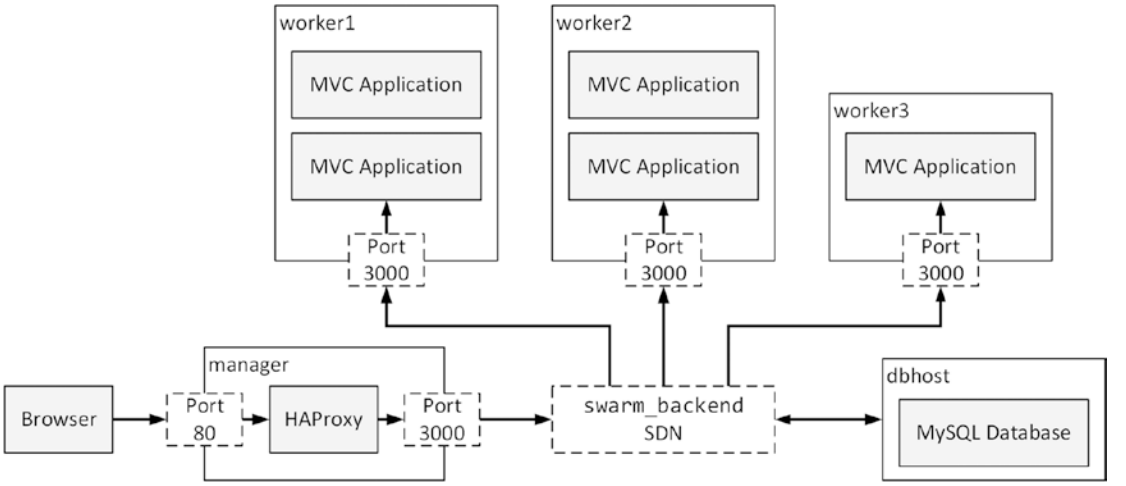

Figure 7-6. *Completing the application*

You can test the application by using the browser to request port 80 on the manager node, which means using the URL `http://manager` in my setup. The HTTP request is fed to the HAProxy load balancer through the port 80 mapping on the manager node, which then directs the request to port 3000, relying on the ingress load balancing feature to distribute the request to the MVC application containers.

Reloading the web page will show that different MVC application containers are being used to handle the requests, although you may have to wait for a moment before reloading (or use a different browser) to see the change illustrated in Figure 7-7.

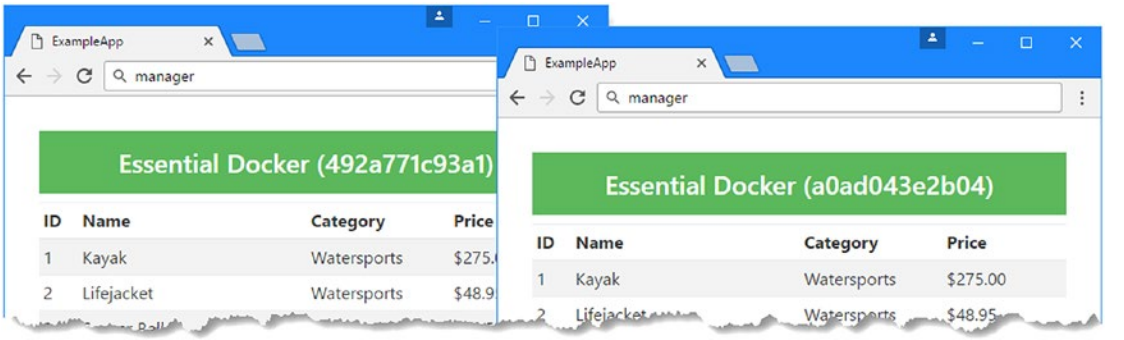

Figure 7-7. *Making requests through a load balancer*

■ **Tip** If you don't get the expected results, use the `docker logs loadbalancer` command to inspect the output from the load balancer container for errors. Resolve the problem, use `docker rm -f loadbalancer` to remove the container, and run the command in Listing 7-18 again.

Removing the Service from the Swarm

To remove the service from the cluster, run the commands shown in Listing 7-19 on the manager node.

Listing 7-19. Removing Docker Services

```
docker service rm mvcapp
docker service rm mysql
docker rm -f loadbalancer
docker network rm swarm_backend
```

Deploying to a Swarm Using a Compose File

Although you can deploy an application to a swarm using the commands described in the previous section, it is easier to include the information in a compose file and let Docker take care of the deployment. Add a file called `docker-compose-swarm.yml` in the `ExampleApp` folder of the manager node and add the configuration shown in Listing 7-20.

Listing 7-20. The Contents of docker-compose-swarm.yml in the ExampleApp Folder

```
version: "3"

volumes:
  productdata_swarm:

networks:
  backend:

services:

  mysql:
    image: "mysql:8.0.0"
    volumes:
      - productdata_swarm:/var/lib/mysql
    networks:
      - backend
    environment:
      - MYSQL_ROOT_PASSWORD=mysecret
      - bind-address=0.0.0.0
    deploy:
        replicas: 1
        placement:
            constraints:
            - node.hostname == dbhost
```

```
  mvc:
    image: "apress/exampleapp:swarm-1.0"
    networks:
      - backend
    environment:
      - DBHOST=mysql
    ports:
      - 3000:80
    deploy:
        replicas: 5
        placement:
            constraints:
            - node.labels.type == mvc
```

A compose file that targets a swarm describes the desired state for a service using the `deploy` configuration section, which is described in Table 7-6.

Table 7-6. *The Deployment Configuration Options in the Compose File*

Name	Description
`deploy`	This configuration section is used to configure the desired state for the service when it is deployed on the cluster.
`replicas`	This setting is used to specify how many instances of a container are required for a service. In this example, there will be one container for the database and five containers for the ASP.NET Core MVC application.
`placement`	This configuration section is used to configure the placement of the containers for the service.
`constraints`	This setting specifies constrains for locating the containers in the swarm. In this example, the database container will run only on the `dbhost` node, and the MVC application containers will run on the nodes that were labeled in Listing 7-4.

Some Docker Compose features are not supported when deploying services to a swarm, such as the `build` configuration section (services must use images) and the `depend_on` setting. The compose file should contain only those services that you want to deploy to the cluster. The file in Listing 7-20, for example, omits details of the database initialization and load balancer containers, which will be created on the manager node, outside of the swarm.

Performing the Deployment

Run the command shown in Listing 7-21 in the `ExampleApp` folder on the manager node to deploy the services described in the new compose file.

Listing 7-21. Deploying Services from a Compose File

```
docker stack deploy --compose-file docker-compose-swarm.yml exampleapp
```

A stack is the term that Docker uses to describe a collection of services that make up an application. The `docker stack deploy` command creates a stack using the information in the compose file, using the arguments described in Table 7-7.

Table 7-7. *The Arguments Used for the Docker Stack Deploy Command*

Name	Description
`--compose-file`	The `--compose-file` argument specifies the compose file that contains the details of the services in the stack. An alternative description file, called a bundle, can also be used.
`exampleapp`	The final argument to the `docker stack deploy` command is the name that will be assigned to the stack and that is used to prefix the names of the containers, volumes, and software-defined networks that are created for the stack.

As Docker processes the compose file, it will create the components that application requires, producing output like this:

```
...
Creating network exampleapp_backend
Creating service exampleapp_mvc
Creating service exampleapp_mysql
...
```

The software-defined network and the containers for the database and the MVC application will be started. The `docker service` commands described earlier in the chapter can be used to inspect the services that have been created. Run the commands shown in Listing 7-22 on the manager node.

Listing 7-22. Inspecting the Services

```
docker service ls
docker service ps exampleapp_mvc
```

The `docker service ls` command lists the services that have been deployed to the swarm. The second command details the containers that have been created for the `exampleapp_mvc` service. The name of the service combines the final argument given to the `docker stack deploy` command in Listing 7-21 and the name of the service defined in the compose file in Listing 7-20.

Preparing the Database

The process for preparing the database hasn't changed. Run the command in Listing 7-23 to expose port 3306 via the ingress load balancer so that the database can be reached outside of the software-defined network defined in the compose file.

Listing 7-23. Exposing the Database Port

```
docker service update --publish-add 3306:3306 exampleapp_mysql
```

Give Docker a minute to perform the update and then run the command shown in Listing 7-24 in the ExampleApp folder of the manager node to apply the Entity Framework Core migration and create the seed data. (If you get a database connection error, then wait a little longer and try again.)

Listing 7-24. Initializing the Database

```
dotnet run --INITDB=true
```

Creating the Deployment Load Balancer

The final step is to create the load balancer, which is outside of the swarm for the reasons described earlier in the chapter. Run the command shown in Listing 7-25 in the ExampleApp folder of the manager node to create the load balancer container. This container isn't part of the cluster but relies on the ingress load balancing feature to distribute HTTP requests to the MVC application containers. (Enter the command on a single line.)

Listing 7-25. Creating the Load Balancer

```
docker run -d --name stack_loadbalancer
  -v "$(pwd)/haproxy.cfg:/usr/local/etc/haproxy/haproxy.cfg" -p 80:80 haproxy:1.7.0
```

Once the load balancer has started, you can test the application by opening a new browser window and requesting port 80 on the manager node. For me, this means the URL is http://manager. The configuration of the application is just the same as when each component was created manually, but using the compose file simplifies the commands used for deployment.

Managing the Swarm

To a certain extent, a swarm is self-managing because when a container or a node crashes, the manager will try to create additional containers to return to the desired state of the services. Even so, there are times when you will want to change the way that a swarm behaves, which you can do using the features described in the sections that follow.

Scaling Services

One of the most common changes to a service is to alter the number of containers to better suit the application's workload. Run the command shown in Listing 7-26 on the manager node to alter the number of containers in the MVC application service.

Listing 7-26. Changing the Number of Containers

```
docker service scale exampleapp_mvc=3
```

The docker service scale command is used to change the desired state of the service. This command tells Docker to reduce the number of containers in the mvc service to 3. It will take a moment for the change to take effect, but once Docker has adjusted the service, the new configuration can be seen by running the docker service ls command, which will produce a response like this:

```
ID            NAME              MODE        REPLICAS  IMAGE
41sdv2rpppyr  exampleapp_mysql  replicated  1/1       mysql:8.0.0
mchxy13rn37z  exampleapp_mvc    replicated  3/3       apress/exampleapp:swarm-1.0
```

Run the command shown in Listing 7-27 on the manager node to return the service to its original number of containers.

Listing 7-27. Returning to the Original Scale

```
docker service scale exampleapp_mvc=5
```

Taking Nodes Out of Service

There will be times when you need to take a node out of the swarm, perhaps to perform an upgrade or some other kind of maintenance. You could just disconnect the node and let the manager detect the change and rebalance the service, but a more orderly approach is to drain a node in the swarm. Draining a node takes it out of service and causes the manager to redistribute its containers to other nodes, at which point it can be shut down without interrupting service delivery. Run the command shown in Listing 7-28 on the manager node to drain the `worker2` node.

Listing 7-28. Draining a Node

```
docker node update --availability drain worker2
```

In addition to labeling nodes, as shown in Listing 7-4, the `docker node update` command can be used with the `--availability` argument to change the status of nodes in a swarm. There are three types of availability, as described in Table 7-8.

Table 7-8. *The Swarm Node Availability Types*

Name	Description
`active`	This is the default availability. The manager is free to assign new containers to the node, and any existing containers continue running indefinitely.
`pause`	In this mode, the existing containers on the node will continue to run, but the manager won't assign any new containers.
`drain`	In this mode, existing containers will be stopped, and the manager will not assign any new containers.

The `drain` mode takes a node out of service, which can be useful for performing maintenance or, as in this case, testing the ability of a swarm to redistribute its workload. Run the `docker service ps exampleapp_mvc` command, and you will see how the status of the service has changed.

```
ID             NAME                      NODE     DESIRED STATE  CURRENT STATE
7jd1yop74mxfy  exampleapp_mvcapp.1       worker1  Running        Running
2ejbexg6yezd9  exampleapp_mvcapp.2       worker1  Running        Running
cse2cyutpvxh9   \_ exampleapp_mvcapp.2   worker2  Shutdown       Shutdown
5r9y8OOjugey8  exampleapp_mvcapp.3       worker1  Running        Running
cktdzf0ypzcpt  exampleapp_mvcapp.4       worker3  Running        Running
6j8h32cikboe2  exampleapp_mvcapp.5       worker3  Running        Running
5esdl2d3jd1ee   \_ exampleapp_mvcapp.5   worker2  Shutdown       Shutdown
```

The manager node responded to the change in the swarm configuration by instructing the `worker1` and `worker3` nodes to start additional containers. The containers on the drained node are not removed, which means they can be used again once the server is returned to the `active` state. Run the command shown in Listing 7-29 on the manager node to reactivate the node.

Listing 7-29. Activating a Node

```
docker node update --availability active worker2
```

The manager will not automatically redistribute the containers in a service when you reactivate a node as long as it has been able to achieve the desired state using the remaining nodes in the swarm. Run the command shown in Listing 7-30 on the manager node to force a redistribution of the containers.

Listing 7-30. Forcing Redistribution of Containers

```
docker service update --force exampleapp_mvc
```

Updating a Service

You can update services to deploy new releases of your applications into the swarm. To create a visible change, alter the banner in the Razor view used by the MVC application, as shown in Listing 7-31.

Listing 7-31. Making a Visible Change in the Index.cshtml File in the ExampleApp/Views/Home Folder

```
@model IEnumerable<ExampleApp.Models.Product>
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>ExampleApp</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
</head>
<body>
    <div class="m-1 p-1">
        <h4 class="bg-success text-xs-center p-1 text-white">
            Swarm: @ViewBag.Message
        </h4>
        <table class="table table-sm table-striped">
            <thead>
                <tr><th>ID</th><th>Name</th><th>Category</th><th>Price</th></tr>
            </thead>
            <tbody>
                @foreach (var p in Model) {
                    <tr>
                        <td>@p.ProductID</td>
                        <td>@p.Name</td>
                        <td>@p.Category</td>
```

```
                    <td>$@p.Price.ToString("F2")</td>
                </tr>

            }
        </tbody>
    </table>
</div>
</body>
</html>
```

Save the change to the view and run the commands shown in Listing 7-32 in the ExampleApp folder of the manager node to publish the application, update the MVC application image, and push it to the Docker Hub. You may need to authenticate yourself with the Docker Hub before you push the new image. Don't forget to change the name of the image to replace apress with your Docker Hub account.

Listing 7-32. Publishing the MVC Application and Updating the Image

```
dotnet publish --framework netcoreapp1.1 --configuration Release --output dist
docker build . -t apress/exampleapp:swarm-1.1 -f Dockerfile
docker push apress/exampleapp:swarm-1.1
```

I have tagged the image to indicate that this is a later version that the image originally used to create the service. Run the command shown in Listing 7-33 on the manager node to upgrade the containers in the MVC application service so they use the new image.

Listing 7-33. Updating a Service

```
docker service update --image apress/exampleapp:swarm-1.1 exampleapp_mvc
```

The --image argument for the docker service update command is used to change the image for a service. The update is performed by stopping each container in turn, applying the update, and then starting it again before moving to the next container. It will take a moment for Docker to update all the containers. You can keep track of the progress by running the docker service ps exampleapp_mvc command, whose output includes the image that each container in the service is using.

■ **Tip** If you have a lot of containers to update, you can speed up the process by using the --update-parallelism argument to the docker service update command, specifying the number of containers that should be taken out of service and updated at the same time. Bear in mind, however, that increasing the number of concurrent updates reduces the number of containers that are available to handle user requests.

Once the changes have been rolled out, use a browser to send an HTTP request to the load balancer, which is listening to port 80 on the manager node. You will see the response shown in Figure 7-8, which shows that the updated MVC application has been used to handle the request.

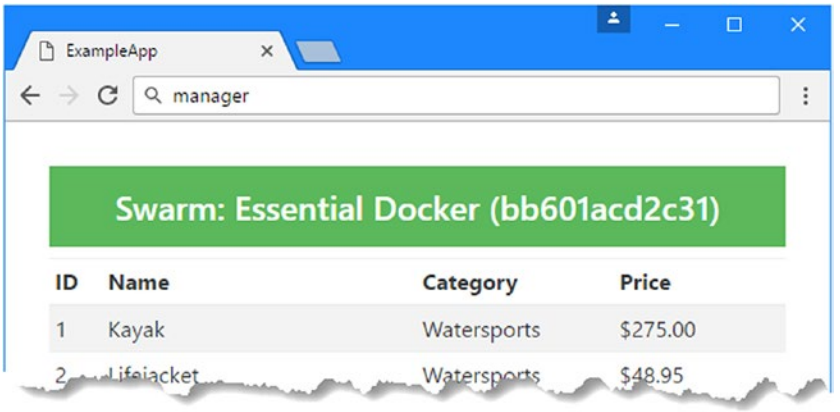

Figure 7-8. *Upgrading a service*

Shutting Down Services

When using a swarm, you stop a service by telling Docker to reduce the number of replicas to zero. Run the command shown in Listing 7-34 on the manager node to scale down the MVC application service so there are no containers running.

Listing 7-34. Scaling Down Docker Services

```
docker service scale exampleapp_mvc=0
```

The Docker workers will stop and remove the containers for the `example_mvcapp` service. It can take a while for all the containers to stop, and you can monitor the process using the `docker service ls` command. Scaling down a service doesn't remove the containers, which Docker keeps in order to speed up the process of scaling up again in the future.

To remove all the components described by the compose file, run the command shown in Listing 7-35 from the `ExampleApp` folder of the manager node.

Listing 7-35. Removing a Docker Stack

```
docker stack rm exampleapp
```

Creating Global Services

The services that I created earlier were replicated, meaning that I specified the number of containers that were required for each service and left the manager node to figure out how to deploy the containers in the swarm.

You can also create global services, in which the service is made up of one container running on each node in the swarm. This can be more predictable than a replicated service because you always know how many containers there are and where they are running, but it provides less flexibility because the number of containers can be increased only by adding additional nodes to the swarm.

■ **Caution** Don't make the mistake of assuming that global services will let you work around the limitations of stateful applications, such that you can use the load balancer to send all the requests from a single client to the same container. Containers in a global service are still part of the swarm, which means that HTTP requests sent to the ports exposed by the containers are handled using ingress load balancing.

To create a global service, set the `--mode` argument to `global` when using the `docker service create` command or use the `mode` setting in the compose file, as shown in Listing 7-36.

Listing 7-36. Creating a Global Service in the docker-compose-swarm.yml File in the ExampleApp Folder

```
version: "3"

volumes:
  productdata_swarm:

networks:
  backend:

services:

  mysql:
    image: "mysql:8.0.0"
    volumes:
      - productdata_swarm:/var/lib/mysql
    networks:
      - backend
    environment:
      - MYSQL_ROOT_PASSWORD=mysecret
      - bind-address=0.0.0.0
    deploy:
        replicas: 1
        placement:
            constraints:
            - node.hostname == dbhost

  mvc:
    image: "apress/exampleapp:swarm-1.0"
    networks:
      - backend
    environment:
      - DBHOST=mysql
    ports:
      - 3000:80
    deploy:
        mode: global
        placement:
            constraints:
            - node.labels.type == mvc
```

The replicas setting has no effect on a global service: there will be exactly one MVC container running on every node in the swarm that meets the constraints. Run the command in Listing 7-37 from the ExampleApp folder on the manager node to deploy the services described in the compose file.

Listing 7-37. Deploying the Application

```
docker stack deploy --compose-file docker-compose-swarm.yml exampleapp
```

You can see the list of containers by running the command shown in Listing 7-38 on the manager node.

Listing 7-38. Inspecting the Global Service

```
docker service ps exampleapp_mvc
```

The output shows that the containers are running on all three of the nodes that meet the constraint.

```
ID            NAME                  IMAGE                          NODE     STATE
rwq13zt1pmOp  exampleapp_mvc.p4ar   apress/exampleapp:swarm-1.0    worker3  Running
ojfwoezOnyvy  exampleapp_mvc.wn4u   apress/exampleapp:swarm-1.0    worker2  Running
qllo5aOhzy4o  exampleapp_mvc.wked   apress/exampleapp:swarm-1.0    worker1  Running
```

Run the docker service ls command on the manager node to inspect the services and you will see that the mvcapp service is marked as global, like this:

```
ID            NAME              MODE        REPLICAS  IMAGE
7vdsOgsxjxv9  exampleapp_mysql  replicated  1/1       mysql:8.0.0
wd8ouiyu537q  exampleapp_mvc    global      3/3       apress/exampleapp:swarm-1.0
```

Summary

In this chapter, I showed you how to cluster together Linux servers running Docker to form a swarm. I demonstrated how to deploy the example application into the swarm, how swarms load balance requests between the containers they contain, and how to manage the swarm once the application has been deployed. In the next chapter, I explain how to use Docker to create a containerized development environment.