14: Deploying apps with Docker Stacks

Deploying and managing multi-service apps at scale is hard.

Fortunately, Docker Stacks are here to help! They simplify application management by providing; *desired state, rolling updates, simple, scaling operations, health checks,* and more! All wrapped in a nice declarative model. Love it!

Now then, if these buzzwords are new to you or sound complicated, don't worry! You'll understand them all by the end of the chapter!

We'll split this chapter into the usual three parts:

- The TLDR
- The deep dive
- The commands

Deploying apps with Docker Stacks - The TLDR

Testing and deploying simple apps on your laptop is easy. But that's for amateurs. Deploying and managing multi-service apps, in real-world production environments... That's for pro's!

Fortunately, stacks are here to help!. They let you define complex multi-service apps in a single declarative file. They also provide a simple way deploy the app and manage its entire lifecycle — initial deployment > health checks > scaling > updates > rollbacks and more!

The process is simple. Define your app in a *Compose file*, then deploy and manage it with the `docker stack deploy` command. That's it!

The Compose file includes the entire stack of services that make up the app. It also includes all of the volumes, networks, secrets, and other infrastructure the app needs.

You then use the `docker stack deploy` command to deploy the app from the file. Simple.

To accomplish all of this, stacks build on top of Docker Swarm, meaning you get all of the security and advanced features that come with Swarm.

In a nutshell, Docker is great for development and testing. Docker Stacks are great for scale and production!

Deploying apps with Docker Stacks - The Deep Dive

If you know Docker Compose, you'll find Docker Stacks really easy. In fact, in many ways, stacks are what we always wished Compose was — fully integrated into Docker, and able to manage the entire lifecycle of applications.

Architecturally speaking, stacks are at the top of the Docker application hierarchy. They build on top of *services*, which in turn build on top of containers. See Figure 14.1.

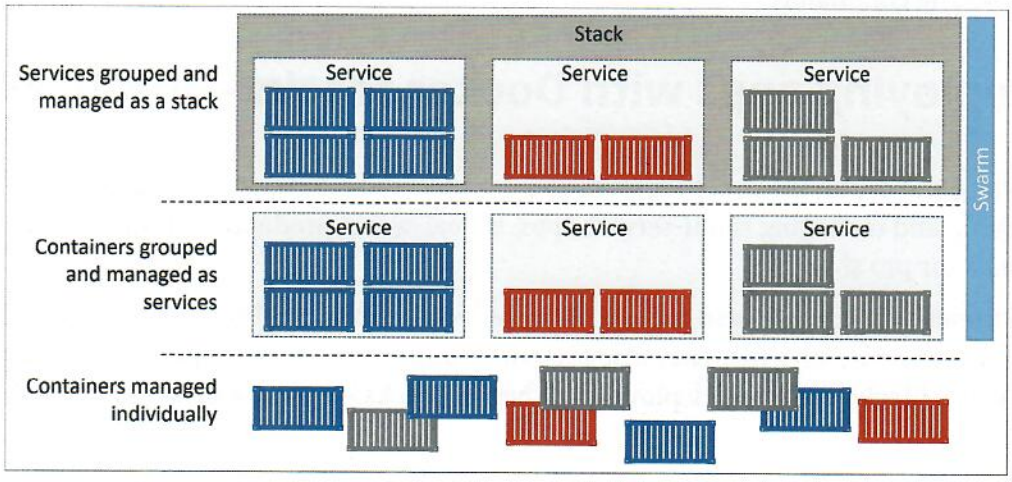

Figure 14.1 AtSea Shop high level architecture

We'll divide this section of the chapter as follows:

- Overview of the sample app
- Looking closer at the stack file
- Deploying the app
- Managing the app

Overview of the sample app

For the rest of the chapter, we'll be using the popular **AtSea Shop** demo app. It lives on GitHub[23] and is open-sourced under the Apache 2.0 license[24].

We're using this app because it's moderately complicated without being too big to list and describe in a book. Beneath the covers, it's a multi-technology microservices app that leverages certificates and secrets. The high-level application architecture is shown in Figure 14.2.

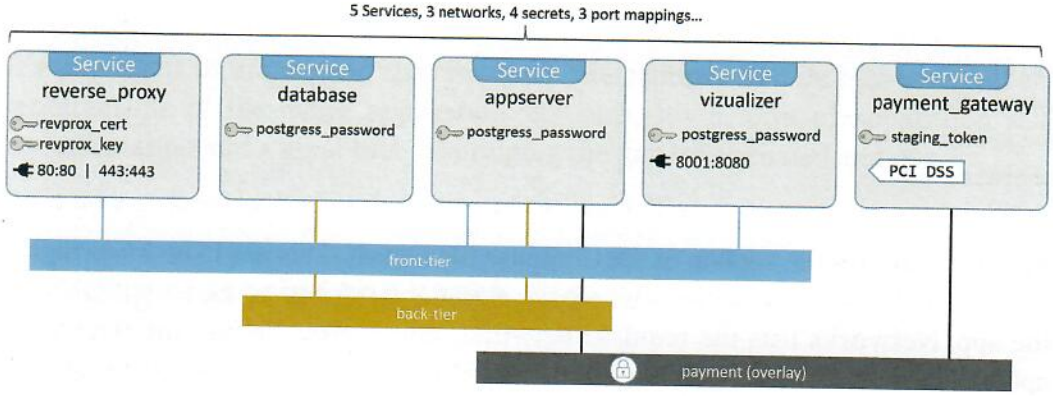

Figure 14.2 AtSea Shop high level architecture

As we can see, it comprises 5 *Services*, 3 networks, 4 secrets, and 3 port mappings. We'll see each of these in detail when we inspect the stack file.

> **Note:** When referring to *services* in this chapter, we're talking about Docker Services (a collection of containers managed as a single object and the service object that exists in the Docker API).

[23] https://github.com/dockersamples/atsea-sample-shop-app
[24] https://github.com/dockersamples/atsea-sample-shop-app/blob/master/LICENSE

Clone the application's GitHub repo so that you have all of the application source files on your local machine.

```
$ git clone https://github.com/dockersamples/atsea-sample-shop-app.git Cloning
into 'atsea-sample-shop-app'...
remote: Counting objects: 636, done.
remote: Total 636 (delta 0), reused 0 (delta 0), pack-reused 636
Receiving objects: 100% (636/636), 7.23 MiB | 28.25 MiB/s, done.
Resolving deltas: 100% (197/197), done.
```

The application consists of several directories and source files. Feel free to explore them all. However, we're going to focus on the `docker-stack.yml` file. We'll refer to this as the *stack file*, as this defines the app and its requirements.

At the highest level, it defines 4 top-level keys.

```
version:
services:
networks:
secrets:
```

Version indicates the version of the Compose file format. This has to be 3.0 or higher to work with stacks. **Services** is where we define the stack of services that make up the app. **Networks** lists the required networks, and **secrets** defines the secrets the app uses.

If we expand each top-level key, we'll see how things map to Figure 14.1. The stack file has five services called "reverse_proxy", "database", "appserver", "visualizer", and "payment_gateway". So does Figure 14.1. The stack file has three networks called "front-tier", "back-tier", and "payment". So does Figure 14.1. Finally, the stack file has four secrets called "postgres_password", "staging_token", "revprox_key", and "revprox_cert". So does Figure 14.1.

```
version: "3.2"
services:
    reverse_proxy:
    database:
    appserver:
    visualizer:
    payment_gateway:
networks:
    front-tier:
    back-tier:
    payment:
secrets:
    postgres_password:
    staging_token:
    revprox_key:
    revprox_cert:
```

It's important to understand that the stack file captures and defines many of the requirements of the entire application. As such, it's a form of application self-documentation and a great tool for bridging the gap between dev and ops.

Let's take a closer look at each section of the stack file.

Looking closer at the stack file

The stack file is a Docker Compose file. The only requirement is that the `version:` key specify a value of "3.0" or higher. See the the Docker docs[25] for the latest information on Compose file versions.

One of the first things Docker does when deploying an app from a stack file, is check for, and create the networks listed under the `networks:` key. If the networks do not already exist, Docker will create them.

Let's see the networks defined in the stack file.

Networks

[25] https://docs.docker.com/compose/compose-file/

```
networks:
  front-tier:
  back-tier:
  payment:
    driver: overlay
    driver_opts:
      encrypted: 'yes'
```

Three networks are defined; `front-tier`, `back-tier`, and `payment`. By default, they'll all be created as overlay networks by the `overlay` driver. But the `payment` network is special — it requires an encrypted data plane.

By default, the control plane of all overlay networks is encrypted. To encrypt the data plane, you have two choices:

- Pass the `-o encrypted` flag to the `docker network create` command.
- Specify `encrypted: 'yes'` under `driver_opts` in the stack file.

The overhead incurred by encrypting the data plane depends on various factors such traffic type and traffic flow. However, expect it to be in the region of 10%.

As previously mentioned, all three networks will be created before the secrets and services.

Now let's look at the secrets.

Secrets

Secrets are defined as top-level objects, and the stack file we're using defines four:

```
secrets:
  postgres_password:
    external: true
  staging_token:
    external: true
  revprox_key:
    external: true
  revprox_cert:
    external: true
```

Notice that all four are defined as `external`. This means that they must already exist before the stack can be deployed.

It's possible for secrets to be created on-demand when the application is deployed — just replace `external: true` with `file: <filename>`. However, for this to work, a plaintext file containing the unencrypted value of the secret must already exist on the host's filesystem. This has obvious security implications.

We'll see how to create these secrets when we come to deploy the app. For now, it's enough to know that the application defines four secrets that need pre-creating.

Let's look at each of the services.

Services

Services are where most of the action happens.

Each service is a JSON collection (dictionary) that contains a bunch of keys. We'll step through each one and explain what each of the options does.

The reverse_proxy service

As we can see, the `reverse_proxy` service defines an image, ports, secrets, and networks.

```
reverse_proxy:
  image: dockersamples/atseasampleshopapp_reverse_proxy
  ports:
    - "80:80"
    - "443:443"
  secrets:
    - source: revprox_cert
      target: revprox_cert
    - source: revprox_key
      target: revprox_key
  networks:
    - front-tier
```

The image key is the only mandatory key in the service object. As the name suggests, it defines the Docker image that will be used to build the replicas for the service.

Docker is opinionated, so unless you specify otherwise, the **image** will be pulled from Docker Hub. You can specify images from 3rd-party registries by prepending the image name with the DNS name of the registry's API endpoint such as `gcr.io` for Google's container registry.

One difference between Docker Stacks and Docker Compose, is that stacks do not support **builds**. This means all images have to be built prior to deploying the stack.

The **ports** key defines two mappings:

- `80:80` maps port 80 on the Swarm to port 80 on each service replica.
- `443:443` maps port 443 on the Swarm to port 443 on each service replica.

By default, all ports are mapped using *ingress mode*. This means they'll be mapped and accessible from every node in the Swarm — even nodes not running a replica. The alternative is *host mode*, where ports are only mapped on Swarm nodes running replicas for the service. However, *host mode* requires you to use the long-form syntax. For example, mapping port 80 in *host mode* using the long-form syntax would be like this:

```
ports:
  - target: 80
    published: 80
    mode: host
```

The long-form syntax is recommended, as it's easier to read and more powerful (it supports ingress mode **and** host mode). However, it requires at least version 3.2 of the Compose file format.

The **secrets** key defines two secrets — `revprox_cert` and `revprox_key`. These must be defined in the top-level `secrets` key, and must exist on the system.

Secrets get mounted into service replicas as a regular file. The name of the file will be whatever you specify as the `target` value in the stack file, and the file will appear in the replica under `/run/secrets` on Linux, and `C:\ProgramData\Docker\secrets` on Windows. Linux mounts `/run/secrets` as an in-memory filesystem, but Windows does not.

The secrets defined in this service will be mounted in each service replica as `/run/secrets/revprox_cert` and `/run/secrets/revprox_key`. To mount one of them as `/run/secrets/uber_secret` you would define it in the stack file as follows:

```
secrets:
  - source: revprox_cert
    target: uber_secret
```

The **networks** key ensures that all replicas for the service will be attached to the `front-tier` network. The network specified here must be defined in the `networks` top-level key, and if it doesn't already exist, Docker will create it as an overlay.

The database service

The database service also defines; an image, a network, and a secret. As well as those, it introduces environment variables and placement constraints.

```
database:
  image: dockersamples/atsea_db
  environment:
    POSTGRES_USER: gordonuser
    POSTGRES_DB_PASSWORD_FILE: /run/secrets/postgres_password
    POSTGRES_DB: atsea
  networks:
    - back-tier
  secrets:
    - postgres_password
  deploy:
    placement:
      constraints:
        - 'node.role == worker'
```

The **environment** key lets you inject environment variables into services replica. This service uses three environment variables to define a database user, the location of the database password (a secret mounted into every service replica), and the name of the database.

```
environment:
  POSTGRES_USER: gordonuser
  POSTGRES_DB_PASSWORD_FILE: /run/secrets/postgres_password
  POSTGRES_DB: atsea
```

> **Note:** It would be more secure to pass all three values in as secrets, as this would avoid documenting the database name and database user in plaintext variables.

The service also defines a *placement constraint* under the `deploy` key. This ensures that replicas for this service will always run on Swarm *worker* nodes.

```
deploy:
  placement:
    constraints:
      - 'node.role == worker'
```

Placement constraints are a form of topology-aware scheduling, and can be a great way of influencing scheduling decisions. Swarm currently lets you schedule against all of the following:

- Node ID. `node.id == o2p4kw2uuw2a`
- Node name. `node.hostname == wrk-12`
- Role. `node.role != manager`
- Engine labels. `engine.labels.operatingsystem==ubuntu 16.04`
- Custom node labels. `node.labels.zone == prod1`

Notice that == and != are both supported.

The appserver service

The `appserver` service uses an image, attaches to three networks, and mounts a secret. It also introduces several additional features under the `deploy` key.

```
appserver:
  image: dockersamples/atsea_app
  networks:
    - front-tier
    - back-tier
    - payment
  deploy:
    replicas: 2
    update_config:
      parallelism: 2
    failure_action: rollback
    placement:
      constraints:
```

```
      - 'node.role == worker'
    restart_policy:
      condition: on-failure
      delay: 5s
      max_attempts: 3
      window: 120s
  secrets:
    - postgres_password
```

Let's take a closer look at the new stuff under the `deploy` key.

First up, `services.appserver.deploy.replicas = 2` will set the desired number of replicas for the service to 2. If omitted, the default value is 1. If the service is running, and you need to change the number of replicas, you should do so declaratively. This means updating `services.appserver.deploy.replicas` in the stack file with the new value, and then redeploying the stack. We'll see this later, but re-deploying a stack does not affect services that you haven't made a change to.

`services.appserver.deploy.update_config` tells Docker how to act when rolling-out updates to the service. For this service, Docker will update two replicas at-a-time (`parallelism`) and will perform a 'rollback' if it detects the update is failing. Rolling back will start new replicas based on the previous definition of the service. The default value for `failure_action` is `pause`, which will stop further replicas being updated. The other option is `continue`.

```
update_config:
  parallelism: 2
  failure_action: rollback
```

The `services.appserver.deploy.restart-policy` object tells Swarm how to restart replicas (containers) if and when they fail. The policy for this service will restart a replica if it stops with a non-zero exit code (`condition: on-failure`). It will try to restart the failed replica 3 times, and wait up to 120 seconds to decide if the restart worked. It will wait 5 seconds between each of the three restart attempts.

```
restart_policy:
  condition: on-failure
  delay: 5s
  max_attempts: 3
  window: 120s
```

visualizer

The visualizer service references an image, maps a port, defines an update config, and defines a placement constraint. It also and mounts a volume and defines a custom grace period for container stop operations.

```
visualizer:
  image: dockersamples/visualizer:stable
  ports:
    - "8001:8080"
  stop_grace_period: 1m30s
  volumes:
    - "/var/run/docker.sock:/var/run/docker.sock"
  deploy:
    update_config:
      failure_action: rollback
    placement:
      constraints:
        - 'node.role == manager'
```

When Docker stops a container, it issues a `SIGTERM` to the process with PID 1 inside the container. The container (its PID 1 process) then has a 10-second grace period to perform any clean-up operations. If it doesn't handle the signal, it will be forcibly terminated after 10 seconds with a `SIGKILL`. The `stop_grace_period` property overrides this 10 second grace period.

The `volumes` key is used to mount pre-created volumes and host directories into a service replica. In this case, it's mounting `/var/run/docker.sock` from the Docker host, into `/var/run/docker.sock` inside of each service replica. This means any reads and writes to `/var/run/docker.sock` in the replica will be passed through to the same directory in the host.

`/var/run/docker.sock` happens to be the IPC socket that the Docker daemon exposes all of its API endpoints on. This means giving a container access to it allows the container to consume all API endpoints — essentially giving the container the ability to query and manage the Docker daemon. In most situations this is a huge "No!". However, this is a demo app in a lab environment.

The reason this service requires access to the Docker socket is because it provides a graphical representation of services on the Swarm. To do this, it needs to be able to query the Docker daemon on a manager node. To accomplish this, a placement constraint forces all service replicas onto manager nodes, and the Docker socket is bind-mounted into each service replica. The *bind mount* is shown in Figure 14.3.

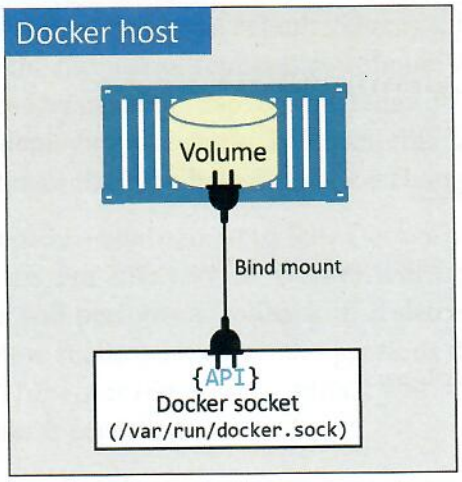

Figure 14.3

payment_gateway

The `payment_gateway` service specifies an image, mounts a secret, attaches to a network, defines a partial deployment strategy, and then imposes a couple of placement constraints.

```
payment_gateway:
  image: dockersamples/atseasampleshopapp_payment_gateway
  secrets:
    - source: staging_token
      target: payment_token
  networks:
    - payment
  deploy:
    update_config:
      failure_action: rollback
    placement:
      constraints:
        - 'node.role == worker'
        - 'node.labels.pcidss == yes'
```

We've seen all of these options before, except for the `node.label` in the placement constraint. Node labels are custom-defined labels added to Swarm nodes with the `docker node update` command. As such, they're only applicable within the context of the nodes role in the Swarm (you can't leverage them on standalone containers or outside of the Swarm).

In this example, the `payment_gateway` service performs operations that require it to run on a Swarm node that has been hardened to PCI DSS standards. To enable this, you can apply a custom *node label* to any Swarm node meeting these requirements. We'll do this when we build the lab to deploy the app.

As this service defines two placement constraints, replicas will only be deployed to nodes that match both. I.e. a **worker** node with the `pcidss=yes` node label.

Now that we're finished examining the stack file, we should have a good understanding of the application's requirements. As mentioned previously, the stack file is a great piece of application documentation. We know that the application has 5 services, 3 networks, and 4 secrets. We know which services attach to which networks, which ports need publishing, which images are required, and we even know that some services need to run on specific nodes.

Let's deploy it.

Deploying the app

There's a few pre-requisites that need taking care of before we can deploy the app:

- **Swarm mode:** We'll deploy the app as a Docker Stack, and stacks require Swarm mode.
- **Labels:** One of the Swarm worker nodes needs a custom node label.
- **Secrets:** The app uses secrets which need pre-creating before we can deploy it.

Building a lab for the sample app

In this section we'll build a three-node Linux-based Swarm cluster that satisfies all of the application's pre-req's. Once we're done, the lab will look like this.

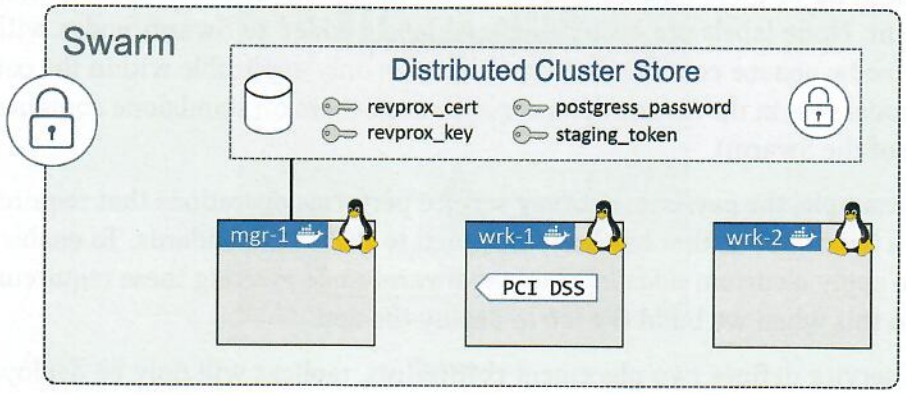

Figure 14.4 Sample lab

We'll complete the following three steps:

- Create a new Swarm
- Add a node label
- Create the secrets

Let's create a new three-node Swarm cluster.

1. Initialize a new Swarm.

   Run the following command on the node that you want to be your Swarm manager.

```
$ docker swarm init
Swarm initialized: current node (lhma...w4nn) is now a manager.
<Snip>
```

2. Add worker nodes.

   Copy the `docker swarm join` command that displayed in the output of the previous command. Paste it into the two nodes you want to join as workers.

```
//Worker 1 (wrk-1)
wrk-1$ docker swarm join --token SWMTKN-1-2hl6...-...3lqg 172.31.40.192:2377
This node joined a swarm as a worker.

//Worker 2 (wrk-2)
wrk-2$ docker swarm join --token SWMTKN-1-2hl6...-...3lqg 172.31.40.192:2377
This node joined a swarm as a worker.
```

3. Verify that the Swarm is configured with one manager and two workers.

   Run this command from the manager node.

```
$ docker node ls
ID            HOSTNAME   STATUS    AVAILABILITY    MANAGER STATUS
lhm...4nn *   mgr-1      Ready     Active          Leader
b74...gz3     wrk-1      Ready     Active
o9x...um8     wrk-2      Ready     Active
```

The Swarm is now ready.

The `payment_gateway` service has set of placement constraints forcing it to only run on **worker nodes** with the `pcidss=yes` node label. In this step we'll add that node label to `wrk-1`.

In the real world you would harden at least one of your Docker nodes to PCI standards before labelling it. However, this is just a lab, so we'll skip the hardening step and just add the label to `wrk-1`.

Run the following commands from the Swarm manager.

1. Add the node label to `wrk-1`.

```
$ docker node update --label-add pcidss=yes wrk-1
```

   Node labels only apply within the Swarm.

2. Verify the node label.

```
$ docker node inspect wrk-1
[
{
    "ID": "b74rzajmrimfv7hood6l4lgz3",
    "Version": {
        "Index": 27
    },
    "CreatedAt": "2018-01-25T10:35:18.146831621Z",
    "UpdatedAt": "2018-01-25T10:47:57.189021202Z",
    "Spec": {
        "Labels": {
            "pcidss": "yes"

        },
        <Snip>
```

The `wrk-1` worker node is now configured so that it can run replicas for the `payment_gateway` service.

The application defines four secrets, all of which need creating before the app can be deployed:

- `postgress_password`
- `staging_token`
- `revprox_cert`
- `revprox_key`

Run the following commands from the manager node to create them.

1. Create a new key pair.

   Three of the secrets will be populated with cryptographic keys. We'll create the keys in this step and then place them inside of Docker secrets in the next steps.

```
$ openssl req -newkey rsa:4096 -nodes -sha256 \
  -keyout domain.key -x509 -days 365 -out domain.crt
```

   You'll have two new files in your current directory. We'll use them in the next step.

2. Create the `revprox_cert`, `revprox_key`, and `postgress_password` secrets.

```
$ docker secret create revprox_cert domain.crt
cqblzfpyv5cxb5wbvtrbpvrrj

$ docker secret create revprox_key domain.key
jqd1ramk2x7g0s2e9ynhdyl4p

$ docker secret create postgres_password domain.key
njpdklhjcg8noy64aileyod6l
```

3. Create the `staging_token` secret.

```
$ echo staging | docker secret create staging_token -
sqy21qep9w17h04k3600o6qsj
```

4. List the secrets.

```
$ docker secret ls
ID          NAME                DESIRED STATE   CREATED               UPDATED
njp...d6l   postgres_password   47 seconds ago        47 seconds ago
cqb...rrj   revprox_cert        About a minute ago    About a minute ago
jqd...l4p   revprox_key         About a minute ago    About a minute ago
sqy...qsj   staging_token       23 seconds ago        23 seconds ago
```

That's all of the pre-requisites taken care of. Time to deploy the app!

Deploying the sample app

If you haven't already done so, clone the app's GitHub repo to your Swarm manager.

```
$ git clone https://github.com/dockersamples/atsea-sample-shop-app.git
Cloning into 'atsea-sample-shop-app'...
remote: Counting objects: 636, done.
Receiving objects: 100% (636/636), 7.23 MiB | 3.30 MiB/s, done. remote:
Total 636 (delta 0), reused 0 (delta 0), pack-reused 636 Resolving
deltas: 100% (197/197), done.
Checking connectivity... done.

$ cd atsea-sample-shop-app
```

Now that you have the code, you are ready to deploy the app.

Stacks are deployed using the `docker stack deploy` command. In its basic form, it accepts two arguments:

- name of the stack file
- name of the stack

The application's GitHub repository contains a stack file called `docker-stack.yml`, so we'll use this as stack file. We'll call the stack `seastack`, though you can choose a different name if you don't like that.

Run the following commands from within the `atsea-sample-shop-app` directory on the Swarm manager.

Deploy the stack (app).

```
$ docker stack deploy -c docker-stack.yml seastack
Creating network seastack_default
Creating network seastack_back-tier
Creating network seastack_front-tier
Creating network seastack_payment
Creating service seastack_database
Creating service seastack_appserver
Creating service seastack_visualizer
Creating service seastack_payment_gateway
Creating service seastack_reverse_proxy
```

You can run `docker network ls` and `docker service ls` commands to see the networks and services that were deployed as part of the app.

A few things to note from the output of the command.

The networks were created before the services. This is because the services attach to the networks, so need the networks to be created before they can start.

Docker prepends the name of the stack to every resource it creates. In our example, the stack is called `seastack`, so all resources are named `seastack_<resource>`. For example, the `payment` network is called `seastack_payment`. Resources that were created prior to the deployment, such as secrets, do not get renamed.

Another thing to note is the presence of a network called `seastack_default`. This isn't defined in the stack file, so why was it created? Every service needs to attach to a network, but the `visualizer` service didn't specify one. Therefore, Docker created one called `seastack_default` and attached it to that.

You can verify the status of a stack with a couple of commands. `docker stack ls` lists all stacks on the system, including how many services they have. `docker stack ps <stack-name>` gives more detailed information about a particular stack, such as *desired state* and *current state*. Let's see them both.

```
$ docker stack ls
NAME                SERVICES
seastack            5

$ docker stack ps seastack
NAME                                NODE      DESIRED STATE    CURRENT STATE
seastack_reverse_proxy.1            wrk-2     Running          Running 7 minutes ago
seastack_payment_gateway.1          wrk-1     Running          Running 7 minutes ago
seastack_visualizer.1               mgr-1     Running          Running 7 minutes ago
seastack_appserver.1                wrk-2     Running          Running 7 minutes ago
seastack_database.1                 wrk-2     Running          Running 7 minutes ago
seastack_appserver.2                wrk-1     Running          Running 7 minutes ago
```

The `docker stack ps` command is a good place to start when troubleshooting services that fail to start. It gives an overview of every service in the stack, including which node each replica is scheduled on, current state, desired state, and error

message. The following output shows two failed attempts to start a replica for the `reverse_proxy` service on the `wrk-2` node.

```
$ docker stack ps seastack
NAME                NODE      DESIRED      CURRENT   ERROR
                              STATE        STATE
reverse_proxy.1     wrk-2     Shutdown     Failed    "task: non-zero exit (1)"
\_reverse_proxy.1   wrk-2     Shutdown     Failed    "task: non-zero exit (1)"
```

For more detailed logs of a particular service you can use the `docker service logs` command. You pass it either the service name/ID, or replica ID. If you pass it the service name or ID, you'll get the logs for all service replicas. If you pass it a particular replica ID, you'll only get the logs for that replica.

The following `docker service logs` command shows the logs for all replicas in the `seastack_reverse_proxy` service that had the two failed replicas in the previous output.

```
$ docker service logs seastack_reverse_proxy
seastack_reverse_proxy.1.zhc3cjeti9d4@wrk-2 | [emerg] 1#1: host not found...
seastack_reverse_proxy.1.6m1nmbzmwh2d@wrk-2 | [emerg] 1#1: host not found...
seastack_reverse_proxy.1.6m1nmbzmwh2d@wrk-2 | nginx: [emerg] host not found..
seastack_reverse_proxy.1.zhc3cjeti9d4@wrk-2 | nginx: [emerg] host not found..
seastack_reverse_proxy.1.1tmya243m5um@mgr-1 | 10.255.0.2 "GET / HTTP/1.1" 302
```

The output is trimmed to fit the page, but you can see that logs from all three service replicas are shown (the two that failed and the one that's running). Each line starts with the name of the replica, which includes the service name, replica number, replica ID, and name of host that it's scheduled on. Following that is the log output.

> **Note:** You might have noticed that all of the replicas in the previous output showed as replica number 1. This is because Docker created one at a time and only started a new one when the previous one had failed.

It's hard to tell because the output is trimmed to fit the book, but it looks like the first two replicas failed because they were relying on something in another service that was still starting (a sort of race condition when dependent services are starting).

You can follow the logs (`--follow`), tail them (`--tail`), and get extra details (`--details`).

Now that the stack is up and running, let's see how to manage it.

Managing the app

We know that a *stack* is set of related services and infrastructure that gets deployed and managed as a unit. And while that's a fancy sentence full of buzzwords, it reminds us that the stack is built from normal Docker resources — networks, volumes, secrets, services etc. This means we can inspect and reconfigure these with their normal docker commands: `docker network`, `docker volume`, `docker secret`, `docker service`...

With this in mind, it's possible to use the `docker service` command to manage services that are part of the stack. A simple example would be using the `docker service scale` command to increase the number of replicas in the `appserver` service. However, **this is not the recommended method!**

The recommended method is the declarative method, which uses the stack file as the ultimate source of truth. As such, all changes to the stack should be made to the stack file, and the updated stack file used to redeploy the app.

Here's a quick example of why the imperative method (making changes via the CLI) is bad:

> *Imagine that we have a stack deployed from the `docker-stack.yml` file that we cloned from GitHub earlier in the chapter. This means we have two replicas of the `appserver` service. If we use the `docker service scale` command to change that to 4 replicas, the current state of the cluster will be 4 replicas, but the stack file will still define 2. Admittedly, that doesn't sound like the end of the world. However, imagine we then make a different change to the stack, this time via the stack file, and we roll it out with the `docker stack deploy` command. As part of this rollout, the number of `appserver` replicas in the cluster will be rolled back to 2, because this is what the stack file defines. For this kind of reason, it is recommended to make all changes to the application via the stack file, and to manage the file in a proper version control system.*

Let's walk through the process of making a couple of declarative changes to the stack. We'll make the following changes:

- Increase the number of `appserver` replicas from 2 to 10
- Increase the stop grace period for the visualizer service to 2 minutes

Edit the `docker-stack.yml` file and update the following two values:

- `.services.appserver.deploy.replicas=10`
- `.services.visualizer.stop_grace_period=2m`

The relevant sections of the stack file will now look like this:

```
<Snip>
appserver:
  image: dockersamples/atsea_app
  networks:
    - front-tier
    - back-tier
    - payment
  deploy:
    replicas: 10              <<Updated value
<Snip>
visualizer:
  image: dockersamples/visualizer:stable
  ports:
    - "8001:8080"
stop_grace_period: 2m         <<Updated value
<Snip
```

Save the file and redeploy the app.

```
$ docker stack deploy -c docker-stack.yml seastack
Updating service seastack_reverse_proxy (id: z4crmmrz7zi83o0721heohsku)
Updating service seastack_database (id: 3vvpkgunetxaatbvyqxfic115)
Updating service seastack_appserver (id: ljht639w33dhv0dmht1q6mueh)
Updating service seastack_visualizer (id: rbwoyuciglre01hsm5fviabjf)
Updating service seastack_payment_gateway (id: w4gsdxfnb5gofwtvmdiooqvxs)
```

Re-deploying the app like this will only update the changed components.

Run a `docker stack ps` to see the number of `appserver` replicas increasing.

```
$ docker stack ps seastack
NAME                    NODE      DESIRED STATE   CURRENT STATE
seastack_visualizer.1   mgr-1     Running         Running 1 second ago
seastack_visualizer.1   mgr-1     Shutdown        Shutdown 3 seconds ago
seastack_appserver.1    wrk-2     Running         Running 24 minutes ago
seastack_appserver.2    wrk-1     Running         Running 24 minutes ago
seastack_appserver.3    wrk-2     Running         Running 1 second ago
seastack_appserver.4    wrk-1     Running         Running 1 second ago
seastack_appserver.5    wrk-2     Running         Running 1 second ago
seastack_appserver.6    wrk-1     Running         Starting 7 seconds ago
seastack_appserver.7    wrk-2     Running         Running 1 second ago
seastack_appserver.8    wrk-1     Running         Starting 7 seconds ago
seastack_appserver.9    wrk-2     Running         Running 1 second ago
seastack_appserver.10   wrk-1     Running         Starting 7 seconds ago
```

The output has been trimmed so that it fits on the page, and so that only the affected services are shown.

Notice that there are two lines for the `visualizer` service. One line shows a replica that was shutdown 3 seconds ago, and the other line shows a replica that has been running for 1 second. This is because we pushed a change to the `visualizer` service, so Swarm terminated the existing replica and started a new one with the new `stop_grace_period` value.

Also note that we now have 10 replicas for the `appserver` service, and that they are in various states in the "CURRENT STATE" column — some are *running* whereas others are still *starting*.

After enough time, the cluster will converge so that *desired state* and *current state* match. At that point, what is deployed and observed on the cluster will exactly match what is defined in the stack file. This is a happy place to be :-D

This update pattern should be used for all updates to the app/stack. I.e. **all changes should be made declaratively via the stack file, and rolled out using `docker stack deploy`.**

The correct way to delete a stack is with the `docker stack rm` command. Be warned though! It deletes the stack without asking for confirmation.

```
$ docker stack rm seastack
Removing service seastack_appserver
Removing service seastack_database
Removing service seastack_payment_gateway
Removing service seastack_reverse_proxy
Removing service seastack_visualizer
Removing network seastack_front-tier
Removing network seastack_payment
Removing network seastack_default
Removing network seastack_back-tier
```

Notice that the networks and services were deleted, but the secrets were not. This is because the secrets were pre-created and existed before the stack was deployed. If your stack defines volumes at the top-level, these will not be deleted by `docker stack rm` either. This is because volumes are intended as long-term persistent data stores and exist independent of the lifecycle of containers, services, and stacks.

Congratulations! You know how to deploy and manage a multi-service app using Docker Stacks.

Deploying apps with Docker Stacks - The Commands

- `docker stack deploy` is the command we use to deploy **and** update stacks of services defined in a stack file (usually `docker-stack.yml`).

- `docker stack ls` will list all stacks on the Swarm, including how many services they have.
- `docker stack ps` gives detailed information about a deployed stack. It accepts the name of the stack as its main argument, lists which node each replica is running on, and shows *desired state* and *current state.*
- `docker stack rm` is the command to delete a stack from the Swarm. It does not ask for confirmation before deleting the stack.

Chapter Summary

Stacks are the native Docker solution for deploying and managing multi-service applications. They're baked into the Docker engine, and offer a simple declarative interface for deploying and managing the entire lifecycle of an application.

We start with application code and a set of infrastructure requirements — things like networks, ports, volumes and secrets. We containerize the application and group together all of the app services and infrastructure requirements into a single declarative stack file. We set the number of replicas, as well as rolling update and restart policies. Then we take the file and deploy the application from it using the `docker stack deploy` command.

Future updates to the deployed app should be done declaratively by checking the stack file out of source control, updating it, re-deploying the app, and checking the stack file back in to source control.

Because the stack file defines things like number of service replicas, you should maintain separate stack files for each of your environments, such as dev, test and prod.