

Figure 11.1 The data flow through the four containers that make up the notification registry example

all the containers or some subset of the services managed by Compose. For example, if you want to see all the logs for all the services, run this:

```
docker-compose logs
```

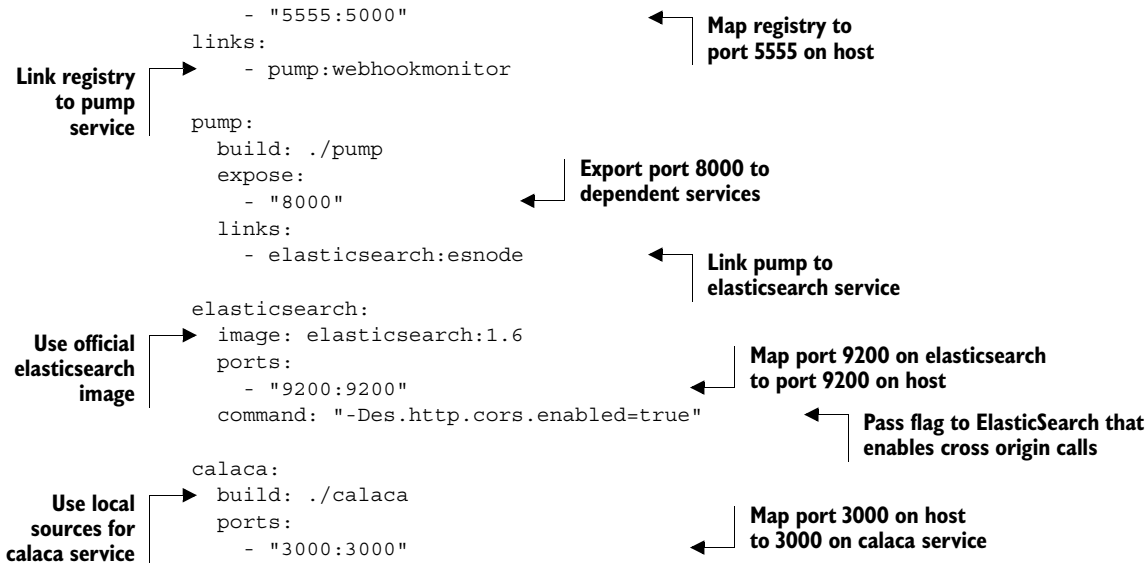
This command will automatically follow the logs, so when you've finished, press Ctrl-C or Control-C to quit. If you want to see only one or more services, then name those services:

```
docker-compose logs pump elasticsearch
```

In this example, you launched the complete environment with a single command and viewed the output with a single command. Being able to operate at such a high level is nice, but the more powerful fact is that you're also in possession of the various sources and can iterate locally with the same ease.

Suppose you have another service that you'd like to bind on port 3000. This would conflict with the `calaca` service in this example. Making the change is as simple as changing `ch11_notifications/docker-compose.yml` and running `docker-compose up` again. Take a look at the file:

```
registry:
  build: ./registry
  ports:
```



Change the last line where it reads `3000:3000` to `3001:3000` and save the file. With the change made, you can rebuild the environment by simply running `docker-compose up -d` again. When you do, it will stop the currently running containers, remove those containers, create new containers, and reattach any volumes that may have been mounted on the previous generation of the environment. When possible, Compose will limit the scope of restarted containers to those that have been changed.

If the sources for your services change, you can rebuild one or all of your services with a single command. To rebuild all the services in your environment, run the following:

```
docker-compose build
```

If you only need to rebuild one or some subset of your services, then simply name the service. This command will rebuild both the `calaca` and `pump` services:

```
docker-compose build calaca pump
```

At this point, stop and remove the containers you created for these services:

```
docker-compose rm -vf
```

By working with these examples, you've touched on the bulk of the development workflow. There are a few surprises: Docker Compose lets the person or people who define the environment worry about the details of working with Docker and frees users or developers to focus on the contained applications.

11.2 *Iterating within an environment*

Learning how Compose fits into your workflow requires a rich example. This section uses an environment similar to one you might find in a real API product. You'll work

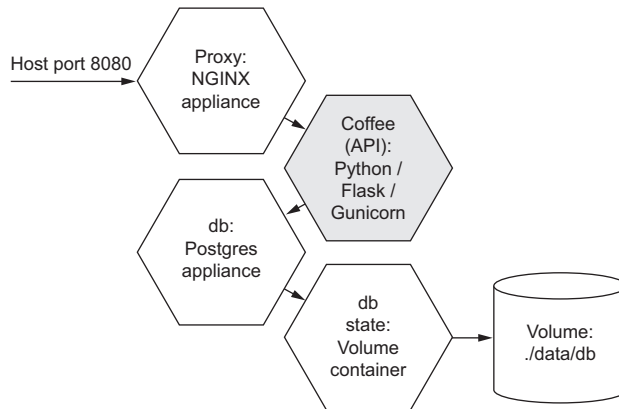


Figure 11.2 Services and service dependencies of the example environment in this chapter

through scenarios and manage the full life cycle for many services. One scenario will guide you through scaling independent services, and another will teach you about state management. Try not to focus too much on how the environment is implemented. The next section covers that.

The environment you’re onboarding with in this section is an API for working with coffee shop metadata. It’s the “brain child of a hot new startup catering to local entrepreneurs and freelancers.” At least it is for the purpose of the example. The environment structure is illustrated in figure 11.2.

Download this example from the GitHub repository:

```
git clone https://github.com/dockerinaction/ch11_coffee_api.git
```

When you run this command, Git will download the most recent copy of the example and place it in a new directory named `ch11_coffee_api` under your current directory. When you’re ready, change into that directory to start working with the environment.

11.2.1 Build, start, and rebuild services

With the sources and environment description copied from version control, start the development workflow by building any artifacts that are declared in the environment. You can do that with the following command:

```
docker-compose build
```

The output from the `build` command will include several lines indicating that specific services have been skipped because they use an image. This environment is made up of four components. Of those, only one requires a build step: the Coffee API. You should see from the output that when Compose built this service, it triggered a Dockerfile build and created an image. The build step runs a `docker build` command for the referenced services.

The Coffee API's source and Dockerfile are contained in the `coffee` folder. It's a simple Flask-based Python application that listens on port 3000. The other services in the environment are out-of-the-box components sourced from images on Docker Hub.

With the environment built, check out the resulting images that have been loaded into Docker. Run `docker images` and look for an image named `ch11coffeeapi-coffee`. Compose uses labels and prefixed names to identify images and containers that were created for a given environment. In this case the image produced for the `coffee` service is prefixed with `ch11coffeeapi_` because that's the derived name for the environment. The name comes from the directory where the `docker-compose.yml` file is located.

You've built a local artifact for the Coffee API, but the environment may reference images that aren't present on your system. You can pull all those with a single Compose command:

```
docker-compose pull
```

This command will pull the most recent images for the tags referenced in the environment. At this point, all the required artifacts should be available on your machine. Now you can start services. Start with the `db` service and pay special attention to the logs:

```
docker-compose up -d db
```

Notice that before Compose started the `db` service, it started the `dbstate` service. This happens because Compose is aware of all the defined services in the environment, and the `db` service has a dependency on the `dbstate` service. When Compose starts any particular service, it will start all the services in the dependency chain for that service. This means that as you iterate, and you only need to start or restart a portion of your environment, Compose will ensure that it comes up with all dependencies attached.

Now that you've seen that Compose is aware of service dependencies, start up the whole environment:

```
docker-compose up
```

When you use an unqualified `docker-compose up` command, Compose will create or re-create every service in the environment and start them all. If Compose detects any services that haven't been built or services that use missing images, it will trigger a build or fetch the appropriate image (just like `docker run`). In this case, you may have noticed that this command re-created the `db` service even though it was already running. This is done to ensure that everything has been brought up in a functioning state. But if you know that the dependencies of a particular service are operating correctly, you can start or restart a service without its dependencies. To do so, include the `--no-dep` flag.

Suppose, for example, that you made a minor adjustment to the configuration for the proxy service (contained in `docker-compose.yml`) and wanted to restart the proxy only. You might simply run the following:

```
docker-compose up --no-dep -d proxy
```

This command will stop any proxy containers that might be running, remove those containers, and then create and start a new container for the `proxy` service. Every other service in the system will remain unaffected. If you had omitted the `--no-dep` flag, then every service would have been re-created and restarted because every service in this environment is either a direct or transitive dependency of `proxy`.

The `--no-dep` flag can come in handy when you're starting systems where components have long-running startup procedures and you're experiencing race conditions. In those cases, you might start those first to let them initialize before starting the rest of the services.

With the environment running, you can try experimenting with and iterating on the project. Load up <http://localhost:8080/api/coffeeshops> (or use your virtual machine IP address) in a web browser. If everything is working properly, you should see a JSON document that looks something like this:

```
{
  "coffeeshops": []
}
```

This endpoint lists all coffee shops in the system. You can see that the list is empty. Next, add some content to become a bit more familiar with the API you're working on. Use the following `cURL` command to add content to your database:

```
curl -H "Content-Type: application/json" \
-X POST \
-d '{"name": "Albina Press", "address": " 5012 Southeast Hawthorne
    Boulevard, Portland, OR", "zipcode": 97215, "price": 2,
    "max_seats": 40, "power": true, "wifi": true}' \
http://localhost:8080/api/coffeeshops/
```

You may need to substitute your virtual machine IP address for “localhost.” The new coffee shop should be in your database now. You can test by reloading `/api/coffeeshops/` in your browser. The result should look like the following response:

```
{
  "coffeeshops": [
    {
      "address": " 5012 Southeast Hawthorne Boulevard, Portland, OR",
      "id": 35,
      "max_seats": 40,
      "name": "Albina Press",
      "power": true,
      "price": 2,
      "wifi": true,
      "zipcode": 97215
    }
  ]
}
```

Now, as is common in the development life cycle, you should add a feature to the Coffee API. The current implementation only lets you create and list coffee shops. It

would be nice to add a basic ping handler for health checks from a load balancer. Open <http://localhost:8080/api/ping> (or use your virtual machine IP address) in a web browser to see how the current application responds.

You're going to add a handler for this path and have the application return the host name where the API is running. Open `./coffee/api/api.py` in your favorite editor and add the following code to the end of the file:

```
@api.route('/ping')
def ping():
    return os.getenv('HOSTNAME')
```

If you're having problems with the next step in the example, or if you're not in the mood to edit files, you can check out a feature branch on the repository where the changes have already been made:

```
git checkout feature-ping
```

Once you've made the change and saved the file (or checked out the updated branch), rebuild and re-create the service with the following commands:

```
docker-compose build coffee
docker-compose up -d
```

The first command will run a `docker build` command for the Coffee API again and generate an updated image. The second command will re-create the environment. There's no need to worry about the coffee shop data you created. The managed volume that was created to store the database will be detached and reattached seamlessly to the new database container. When the command is finished, refresh the web page that you loaded for `/api/ping` earlier. It should display an ID of a familiar style. This is the container ID that's running the Coffee API. Remember, Docker injects the container ID into the `HOSTNAME` environment variable.

In this section you cloned a mature project and were able to start iterating on its functionality with a minimal learning curve. Next you'll scale, stop, and tear down services.

11.2.2 *Scale and remove services*

One of the most impressive and useful features of Compose is the ability to scale a service up and down. When you do, Compose creates more replicas of the containers providing the service. Fantastically, these replicas are automatically cleaned up when you scale down. But as you might expect, containers that are running when you stop an environment will remain until the environment is rebuilt or cleaned up. In this section you'll learn how to scale up, scale down, and clean up your services.

Continuing with the Coffee API example, you should have the environment running. You can check with the `docker-compose ps` command introduced earlier. Remember, Compose commands should be executed from the directory where your `docker-compose.yml` file is located. If the environment isn't running (`proxy`, `coffee`, and `db` services running), then bring it up with `docker-compose up -d`.

Suppose you were managing a test or production environment and needed to increase the parallelism of the `coffee` service. To do so, you'd only need to point your machine at your target environment (as you'll see in chapter 12) and run a single command. In the parameters of this example, you're working with your development environment. Before scaling up, get a list of the containers providing the `coffee` service:

```
docker-compose ps coffee
```

The output should look something like the following:

Name	Command	State	Ports
ch11coffeeapi_coffee_1	./entrypoint.sh	Up	0.0.0.0:32807->3000/tcp

Notice the far-right column, which details the host-to-container port mapping for the single container running the service. You can access the Coffee API served by this container directly (without going through the proxy) by using this public port (in this case, 32807). The port number will be different on your computer. If you load the ping handler for this container, you'll see the container ID running the service. Now that you've established a baseline for your system, scale up the `coffee` service with the following command:

```
docker-compose scale coffee=5
```

The command will log each container that it creates. Use the `docker-compose ps` command again to see all the containers running the `coffee` service:

Name	Command	State	Ports
ch11coffeeapi_coffee_1	./entrypoint.sh	Up	0.0.0.0:32807->3000/tcp
ch11coffeeapi_coffee_2	./entrypoint.sh	Up	0.0.0.0:32808->3000/tcp
ch11coffeeapi_coffee_3	./entrypoint.sh	Up	0.0.0.0:32809->3000/tcp
ch11coffeeapi_coffee_4	./entrypoint.sh	Up	0.0.0.0:32810->3000/tcp
ch11coffeeapi_coffee_5	./entrypoint.sh	Up	0.0.0.0:32811->3000/tcp

As you can see, there are now five containers running the Coffee API. These are all identical with the exception of their container IDs and names. These containers even use identical host port mappings. The reason this example works is that the Coffee API's internal port 3000 has been mapped to the host's ephemeral port (port 0). When you bind to port 0, the OS will select an available port in a predefined range. If instead it were always bound to port 3000 on the host, then only one container could be running at a time.

Test the ping handler on a few different containers (using the dedicated port for the container) before moving on. This example project is used throughout the remainder of the book. At this point, however, there's not much else to do but scale back down to a single instance. Issue a similar command to scale down:

```
docker-compose scale coffee=1
```

← Note the `l` here

The logs from the command indicate which instances are being stopped and removed. Use the `docker-compose ps` command again to verify the state of your environment:

Name	Command	State	Ports
ch1lcoffeeapi_coffee_1	./entrypoint.sh	Up	0.0.0.0:32807->3000/tcp

Before moving on to learning about persistent state, clean up the environment so you can start fresh with `docker-compose rm`.

11.2.3 Iteration and persistent state

You’ve already learned the basics of environment state management with Compose. At the end of the last section you stopped and removed all the services and any managed volumes. Before that you also used Compose to re-create the environment, effectively removing and rebuilding all the containers. This section is focused on the nuances of the workflow and edge cases that can have some undesired effects.

First, a note about managed volumes. Volumes are a major concern of state management. Fortunately, Compose makes working with managed volumes trivial in iterative environments (see figure 11.3). When a service is rebuilt, the attached managed volumes are not removed. Instead they are reattached to the replacing containers for that service. This means that you’re free to iterate without losing your data. Managed volumes are finally cleaned up when the last container is removed using `docker-compose rm` and the `-v` flag.

The bigger issue with state management and Compose is environment state. In highly iterative environments you’ll be changing several things, including the environment configuration. Certain types of changes can create problems.

For example, if you rename or remove a service definition in your `docker-compose.yml`, then you lose the ability to manage it with Compose. Tying this back to the Coffee API example, the `coffee` service was named `api` during development. The environment was in a constant state of flux, and at some point when the `api` service was running, the service was renamed to `coffee`. When that happened, Compose was

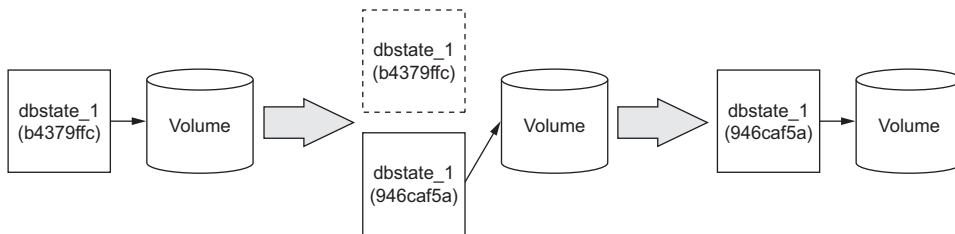


Figure 11.3 A volume container will have the same managed volume reattached after being re-created.

no longer aware of the `api` service. Rebuilds and relaunches worked only on the new `coffee` service, and the `api` service was orphaned.

You can discover this state when you use `docker ps` to list the running containers and notice that containers for old versions of the service are running when none should be. Recovery is simple enough. You can either use `docker` commands to directly clean up the environment or add the orphan service definition back to the `docker-compose.yml` and clean up with Compose.

11.2.4 Linking problems and the network

The last thing to note about using Compose to manage systems of services is remembering the impact of container-linking limitations.

In the Coffee API sample project, the proxy service has a link dependency on the `coffee` service. Remember that Docker builds container links by creating firewall rules and injecting service discovery information into the dependent container's environment variables and `/etc/hosts` file.

In highly iterative environments, a user may be tempted to relaunch only a specific service. That can cause problems if another service is dependent on it. For example, if you were to bring up the Coffee API environment and then selectively relaunch the `coffee` service, the `proxy` service would no longer be able to reach its upstream dependency. When containers are re-created or restarted, they come back with different IP addresses. That change makes the information that was injected into the `proxy` service stale.

It may seem burdensome at times, but the best way to deal with this issue in environments without dynamic service discovery is to relaunch whole environments, at a minimum targeting services that don't act as upstream dependencies. This is not an issue in robust systems that use a dynamic service discovery mechanism or overlay network. Multi-host networking is briefly discussed in chapter 12.

So far, you've used Compose in the context of an existing project. When starting from scratch, you have a few more things to consider.

11.3 Starting a new project: Compose YAML in three samples

Defining an environment is no trivial task, requiring insight and forethought. As project requirements, traffic shape, technology, financial constraints, and local expertise change, so will the environment for your project. For that reason, maintaining clear separation of concerns between the environment and your project is critical. Failing to do so often means that iterating on your environment requires iterating on the code that runs there. This section demonstrates how the features of the Compose YAML can help you build the environments you need.

The remainder of this section will examine portions of the `docker-compose.yml` file included with the Coffee API sample. Relevant excerpts are included in the text.

11.3.1 Prelaunch builds, the environment, metadata, and networking

Begin by examining the `coffee` service. This service uses a Compose managed build, environment variable injection, linked dependencies, and a special networking configuration. The service definition for `coffee` follows:

```
coffee:
  build: ./coffee
  user: 777:777
  restart: always
  expose:
    - 3000
  ports:
    - "0:3000"
  links:
    - db:db
  environment:
    - COFFEEFINDER_DB_URI=postgresql://postgres:development@db:5432/po...
    - COFFEEFINDER_CONFIG=development
    - SERVICE_NAME=coffee
  labels:
    com.dockerinaction.chapter: "11"
    com.dockerinaction.example: "Coffee API"
    com.dockerinaction.role: "Application Logic"
```

1 Builds from Dockerfile located under `./coffee`

2 Set environment to use a database

3 Label the service

4 Expose and map ports for containers

When you have an environment that's closely tied to specific image sources, you might want to automate the build phase of those services with Compose. In the Coffee API sample project this is done for the `coffee` service. But the use case extends beyond typical development environment needs.

If your environments use data-packed volume containers to inject environment configuration, you might consider using a Compose managed build phase for each environment. Whatever the reason, these are available with a simple YAML key and structure. See ❶ in the preceding Compose file.

The value of the `build` key is the directory location of the Dockerfile to use for a build. You can use relative paths from the location of the YAML file. You can also provide an alternative Dockerfile name using the `dockerfile` key.

The Python-based application requires a few environment variables to be set so that it can integrate with a database. Environment variables can be set for a service with the `environment` key and a nested list or dictionary of key-value pairs. In ❷ the list form is used.

Alternatively you can provide one or many files containing environment variable definitions with the `env_file` key. Similar to environment variables, container metadata can be set with a nested list or dictionary for the `labels` key. The dictionary form is used at ❸.

Using detailed metadata can make working with your images and containers much easier, but remains an optional practice. Compose will use labels to store metadata for service accounting.

Last, ❹ shows where this service customizes networking by exposing a port, binding to a host port, and declaring a linked dependency.

The `expose` key accepts a list of container ports that should be exposed by firewall rules. The `ports` key accepts a list of strings that describe port mappings in the same format accepted by the `-p` option on the `docker run` command. The `links` command accepts a list of link definitions in the format accepted by the `docker run --link` flag. Working with these options should be familiar after reading chapter 5.

11.3.2 Known artifacts and bind-mount volumes

Two critical components in the Coffee API sample are provided by images downloaded from Docker Hub. These are the `proxy` service, which uses an official NGINX repository, and the `db` service, which uses the official Postgres repository. Official repositories are reasonably trustworthy, but it's a best practice to pull and inspect third-party images before deploying them in sensitive environments. Once you've established trust in an image, you should use content-addressable images to ensure no untrusted artifacts are deployed.

Services can be started from any image with the `image` key. Both the `proxy` and `db` services are image-based and use content-addressable images:

```
db:
  image: postgres@sha256:66ba100bc635be17...
  volumes_from:
    - dbstate
  environment:
    - PGDATA=/var/lib/postgresql/data/pgdata
    - POSTGRES_PASSWORD=development
  labels:
    com.dockerinaction.chapter: "11"
    com.dockerinaction.example: "Coffee API"
    com.dockerinaction.role: "Database"
```

Use a data container pattern

Use content-addressable images for trusted Postgres version

```
proxy:
  image: nginx@sha256:a2b8bef333864317...
  restart: always
  volumes:
    - ./proxy/app.conf:/etc/nginx/conf.d/app.conf
  ports:
    - "8080:8080"
  links:
    - coffee
  labels:
    com.dockerinaction.chapter: "11"
    com.dockerinaction.example: "Coffee API"
    com.dockerinaction.role: "Load Balancer"
```

Inject configuration via volume

Use content-addressable image for trusted NGINX version

The Coffee API project uses both a database and load balancer with only minimal configuration. The configuration that's provided comes in the form of volumes.

The proxy uses a volume to bind-mount a local configuration file into the NGINX dynamic configuration location. This is a simple way to inject configuration without the trouble of building completely new images.

The `db` service uses the `volumes_from` key to list services that define required volumes. In this case `db` declares a dependency on the `dbstate` service, a volume container service.

In general, the YAML keys are closely related to features exposed on the `docker run` command. You can find a full reference at <https://docs.docker.com/compose/yml/>.

11.3.3 Volume containers and extended services

Occasionally you'll encounter a common service archetype. Examples might include a NodeJS service, Java service, NGINX-based load balancer, or a volume container. In these cases, it may be appropriate to manifest those archetypes as parent services and extend and specialize those for particular instances.

The Coffee API sample project defines a volume container archetype named `data`. The archetype is a service like any other. In this case it specifies an image to start from, a command to run, a UID to run as, and label metadata:

```
data:
  image: gliderlabs/alpine
  command: echo Data Container
  user: 999:999
  labels:
    com.dockerinaction.chapter: "11"
    com.dockerinaction.example: "Coffee API"
    com.dockerinaction.role: "Volume Container"
```

Alone, the service does nothing except define sensible defaults for a volume container. Note that it doesn't define any volumes. That specialization is left to each volume container that extends the archetype:

```
dbstate:
  extends:
    file: docker-compose.yml
    service: data
  volumes:
    - /var/lib/postgresql/data/pgdata
```

**Reference to parent
service in another file**

The `dbstate` service defined a volume container that extends the `data` service. Service extensions must specify both the file and service name being extended. The relevant keys are `extends` and nested `file` and `service`. Service extensions work in a similar fashion to Dockerfile builds. First the archetype container is built and then it is committed. The child is a new container built from the freshly generated layer. Just like a Dockerfile build, these child containers inherit all the parent's attributes including metadata.

The `dbstate` service defines the managed volume mounted at `/var/lib/postgresql/data/pgdata` with the `volumes` key. The `volumes` key accepts a list of volume specifications allowed by the `docker run -v` flag. See chapter 4 for information about volume types, volume containers, and volume nuances.

Docker Compose is a critical tool for anyone who has made Docker a core component of their infrastructure. With it, you'll be able to reduce iteration time, version control environments, and orchestrate ad hoc service interactions with declarative documents. Chapter 12 builds on Docker Compose use cases and introduces Docker Machine to help with forensic and automated testing.

11.4 Summary

This chapter focuses on an auxiliary Docker client named Docker Compose. Docker Compose provides features that relieve much of the tedium associated with command-line management of containers. The chapter covers the following:

- Docker Compose is a tool for defining, launching, and managing services, where a service is defined as one or more replicas of a Docker container.
- Compose uses environment definitions that are provided in YAML configuration files.
- Using the `docker-compose` command-line program, you can build images, launch and manage services, scale services, and view logs on any host running a Docker daemon.
- Compose commands for managing environments and iterating on projects are similar to `docker` command-line commands. Building, starting, stopping, removing, and listing services all have equivalent container-focused counterparts.
- With Docker Compose you can scale the number of containers running a service up and down with a single command.
- Declaring environment configuration with YAML enables environment versioning, sharing, iteration, and consistency.

12

Clusters with Machine and Swarm

This chapter covers

- Creating virtual machines running Docker with Docker Machine
- Integrating with and managing remote Docker daemons
- An introduction to Docker Swarm clusters
- Provisioning whole Swarm clusters with Docker Machine
- Managing containers in a cluster
- Swarm solutions to container scheduling and service discovery

The bulk of this book is about interacting with Docker on a single machine. In the real world, you'll work with several machines at the same time. Docker is a great building block for creating large-scale server software. In such environments, you encounter all sort of new problems that aren't addressed by the Docker engine directly.

How can a user launch an environment where different services run on different hosts? How will services in such a distributed environment locate service

dependencies? How can a user quickly create and manage large sets of Docker hosts in a provider-agnostic way? How should services be scaled for availability and failover? With services spread all over a set of hosts, how will the system know to direct traffic from load balancers?

The isolation benefits that containers provide are localized to a given machine. But as the initial shipping container metaphor prophesied, the container abstraction makes all sorts of tooling possible. Chapter 11 talked about Docker Compose, a tool for defining services and environments. In this chapter, you'll read about Docker Machine and Docker Swarm. These tools address the problems that Docker users encounter when provisioning machines, orchestrating deployments, and running clustered server software.

Docker Engine and Docker Compose simplify the lives of developers and operations personnel by abstracting the host from contained software. Docker Machine and Docker Swarm help system administrators and infrastructure engineers extend those abstractions into clustered environments.

12.1 *Introducing Docker Machine*

The first step in learning about and solving distributed systems problems is building a distributed system. Docker Machine can create and tear down whole fleets of Docker-enabled hosts in a matter of seconds. Learning how to use this tool is essential for anyone who wants to learn how to use Docker in distributed cloud or local virtual environments.

Your choice of driver

Docker Machine ships with a number of drivers out of the box. Each driver integrates Docker Machine with a different virtual machine technology or cloud-based virtual computing provider. Every cloud platform has its advantages and disadvantages. There's no difference between a local host and a remote host from the perspective of a Docker client.

Using a local virtual machine driver like VirtualBox will minimize the cost of running the examples in this chapter, but you should consider choosing a driver for your preferred cloud provider instead. There is something powerful about knowing that the commands you'll issue here are actually managing real-world resources and that the examples you will deploy are going to be running on the internet. At that point, you're only a few domain-specific steps away from building real products.

If you do decide to use a cloud provider for these examples, you'll need to configure your environment with the provider-specific information (like access key and secret key) as well as substitute driver-specific flags in any commands in this chapter.

You can find detailed information about the driver-specific flags by running the `docker-machine help create` command or consulting online documentation.

12.1.1 Building and managing Docker Machines

Between the `docker` command line and Compose, you've been introduced to several commands. This section introduces commands for the `docker-machine` command-line program. Because these tools are all so similar in form and function, this section will make the introduction through a small set of examples. If you want to learn more about the `docker-machine` command line, you can always use the `help` command:

```
docker-machine help
```

The first and most important thing to know how to do with Docker Machine is how to create Docker hosts. The next three commands will create three hosts using the VirtualBox driver. Each command will create a new virtual machine on your computer:

```
docker-machine create --driver virtualbox host1
docker-machine create --driver virtualbox host2
docker-machine create --driver virtualbox host3
```

After you run these three commands (they can take a few minutes), you'll have three Docker hosts managed by Docker Machine. Docker Machine tracks these machines with a set of files in your home directory (under `~/.docker/machine/`). They describe the hosts you have created, the certificate authority certificates used to establish secure communications with the hosts, and a disk image used for VirtualBox-based hosts.

Docker Machine can be used to list, inspect, and upgrade your fleet as well. Use the `ls` subcommand to get a list of managed machines:

```
docker-machine ls
```

That command will display results similar to the following:

NAME	ACTIVE	DRIVER	STATE	URL	SWARM
host1		virtualbox	Running	tcp://192.168.99.100:2376	
host2		virtualbox	Running	tcp://192.168.99.101:2376	
host3		virtualbox	Running	tcp://192.168.99.102:2376	

This command will list each machine, the driver it was created with, its state, and the URL where the Docker daemon can be reached. If you're using Docker Machine to run Docker locally, you'll have another entry in this list, and that entry will likely be marked as `active`. The active machine (indicated with an asterisk under the ACTIVE column) is the one that your environment is currently configured to communicate with. Any commands issued with the `docker` or `docker-compose` command-line interface will connect with the daemon on the active machine.

If you want to know more about a specific machine or look up a specific part of its configuration, you can use the `inspect` subcommand:

```
docker-machine inspect host1
```

```
docker-machine inspect --format "{{.Driver.IPAddress}}" host1
```

← JSON document describing
the machine

← Just the
IP address

The `inspect` subcommand for `docker-machine` is very similar to the `docker inspect` command. You can even use the same Go template syntax (<http://golang.org/pkg/text/template/>) to transform the raw JSON document that describes the machine. This example used a template to retrieve the IP address of the machine. If you needed that in practice, you would use the `ip` subcommand:

```
docker-machine ip host1
```

Docker Machine lets you build a fleet with relative ease, and it's important that you can maintain that fleet with the same ease. Any managed machine can be upgraded with the `upgrade` subcommand:

```
docker-machine upgrade host3
```

This command will result in output like the following:

```
Stopping machine to do the upgrade...
Upgrading machine host3...
Downloading ...
Starting machine back up...
Starting VM...
```

The upgrade procedure stops the machine, downloads an updated version of the software, and restarts the machine. With this command, you can perform rolling upgrades to your fleet with minimal effort.

You will occasionally need to manipulate files on one of your machines or access the terminal on a machine directly. It could be that you need to retrieve or prepare the contents of a bind mount volume. Other times you may need to test the network from the host or customize the host configuration. In those cases, you can use the `ssh` and `scp` subcommands.

When you create or register a machine with Docker Machine, it creates or imports an SSH private key file. That private key can be used to authenticate as a privileged user on the machine over the SSH protocol. The `docker-machine ssh` command will authenticate with the target machine and bind your terminal to a shell on the machine.

For example, if you wanted to create a file on the machine named `host1`, you could issue the following commands:

```
docker-machine ssh host1
touch dog.file
exit
```

← **Bind your terminal
to shell on host1**

← **Exit remote shell
and stop command**

It seems a bit silly to use a fully bound terminal to run a single command. If you don't need a fully interactive terminal, you can alternatively specify the command to run as an additional argument to the `ssh` subcommand. Run the following command to write the name of a dog to the file you just created:

```
docker-machine ssh host1 "echo spot > dog.file"
```

If you have files on one machine that you need to copy elsewhere, you can use the `scp` subcommand to do so securely. The `scp` subcommand takes two arguments: a source

host and file and a destination host and file. Try it for yourself and copy the file you just created from host1 to host2, and then use the `ssh` subcommand to view it on host2:

```
docker-machine scp host1:dog.file host2:dog.file
docker-machine ssh host2 "cat dog.file"
```

← Outputs: spot

The SSH-related commands are critical for customizing configuration, retrieving volume contents, and performing other host-related maintenance. The rest of the commands that you need to build and maintain fleets are predictable.

The commands for starting, stopping (or killing), and removing a machine are just like equivalent commands for working with containers. The `docker-machine` command offers four subcommands: `start`, `stop`, `kill`, and `rm`:

```
docker-machine stop host2
docker-machine kill host3
docker-machine start host2
docker-machine rm host1 host2 host3
```

This section covered the bulk of the basic mechanics for building and maintaining a fleet with Docker Machine. The next section demonstrates how you can use Docker Machine to configure your client environment to work with those machines and how to access the machines directly.

12.1.2 *Configuring Docker clients to work with remote daemons*

Docker Machine accounts for and tracks the state of the machines that it manages. You can use Docker Machine to upgrade Docker on remote hosts, open SSH connections, and securely copy files between hosts. But Docker clients like the `docker` command-line interface or `docker-compose` are designed to connect to a single Docker host at a time. For that reason, one of the most important functions of Docker Machine is producing environment configuration for an active Docker host.

The relationship between Docker Machine, Docker clients, and the environment is illustrated in figure 12.1.

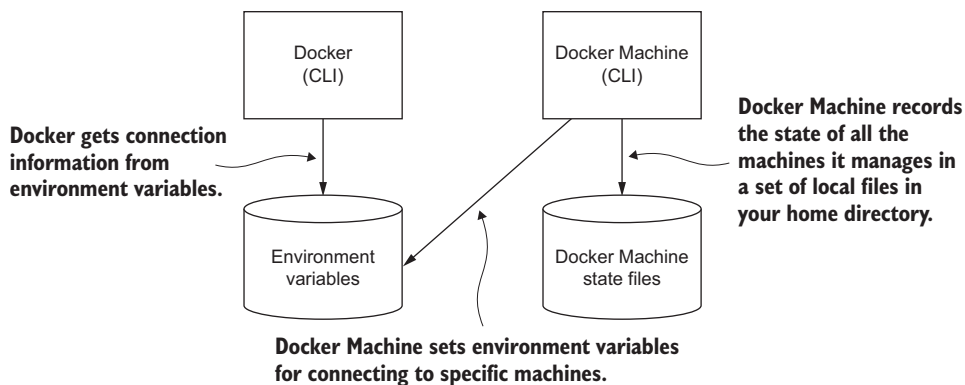


Figure 12.1 The relationship between `docker`, `docker-machine`, and relevant state sources

Get started learning how to manage your Docker environment by creating a couple new machines and activating one. Start by running `create`:

```
docker-machine create --driver virtualbox machine1
docker-machine create --driver virtualbox machine2
```

In order to activate this new machine, you must update your environment. Docker Machine includes a subcommand named `env`. The `env` subcommand attempts to automatically detect the user's shell and print commands to configure the environment to connect to a specific machine. If it can't automatically detect the user's shell, you can set the specific shell with the `--shell` flag:

	<code>docker-machine env machine1</code>	← Let env autodetect your shell
Get PowerShell configuration	<code>docker-machine env --shell powershell machine1</code>	
	<code>docker-machine env --shell cmd machine1</code>	← Get CMD configuration
Get fish configuration	<code>docker-machine env --shell fish machine1</code>	
	<code>docker-machine env --shell bash machine1</code>	← Get the default (POSIX) configuration

Each of these commands will print out the list of shell-specific commands that need to be run along with a comment on how to invoke `docker-machine` so that these are executed automatically. For example, to set `machine1` as the active machine, you can execute the `docker-machine env` command in a POSIX shell:

```
eval "$(docker-machine env machine1)"
```

If you use Windows and run PowerShell, you would run a command like the following:

```
docker-machine env --shell=powershell machine1 | Invoke-Expression
```

You can validate that you've activated `machine1` by running the `active` subcommand. Alternatively, you can check the `ACTIVE` column on the output from the `ls` subcommand:

```
docker-machine active
docker-machine ls
```

Any client that observes the environment configuration on your local computer will use the Docker Remote API provided at the specified URL for the active machine. When the active machine is changed, so will the targets of any Docker client commands be changed. The state of this environment is illustrated in figure 12.2.

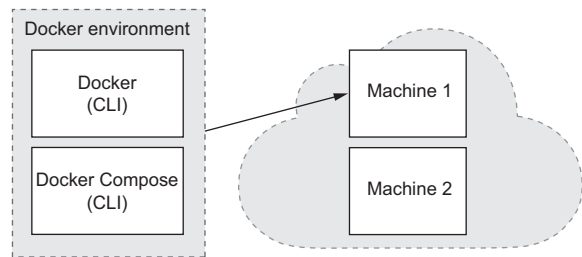


Figure 12.2 One of two machines created with Docker Machine has been activated in the local environment. Docker clients will use the Docker API provided by that machine.

Create a few containers and experience for yourself how simple and subtle it is to work with multiple machines. Start by pulling an image onto the active machine:

```
docker pull dockerinaction/ch12_painted
```

Images pulled onto an active machine will only be pulled onto that machine. This means that if you'll be using some common image across your fleet, you'll need to pull that image on each of those machines. This is important to understand if minimizing container startup time is important. In those cases, you'll want to pull on as many machines in parallel as possible and before container-creation time. Before you start a container with this image, change the active machine to machine2 and list the images:

```
eval "$(docker-machine env machine2)"
docker images
```

← Replace with equivalent command appropriate for your shell

The output from the `images` subcommand should be empty. This example helps illustrate the need to pull images on each machine independently. This machine, machine2, has never had any images installed.

Next, pull the image and run a container for `dockerinaction/ch12_painted` on machine2:

```
docker run -t dockerinaction/ch12_painted \
    Tiny turtles tenderly touch tough turnips.
```

Now compare the list of containers on machine1 with those on machine2:

```
docker ps -a
eval "$(docker-machine env machine1)"
docker ps -a
```

← Replace with equivalent command appropriate for your shell

Again, the list is empty because no containers have been created on machine1. This simple example should help illustrate how easy it is to work with multiple machines. It should also illustrate how confusing it can be to manually connect to, orchestrate, and schedule work across a sizable fleet of machines. Unless you specifically query for the active host, it's difficult to keep track of where your Docker clients are directed as the number of machines you use increases.

Orchestration can be simplified by taking advantage of Docker Compose, but Compose is like any other Docker client and will use only the Docker daemon that your environment has been configured to use. If you were to launch an environment with Compose and your current configuration where machine1 is active, all the services described by that environment would be created on machine1. Before moving on, clean up your environment by removing both machine1 and machine2:

```
docker-machine rm machine1 machine2
```

Docker Machine is a great tool for building and managing machines in a Docker-based fleet. Docker Compose provides orchestration for Docker container-based services. The main problems that remain are scheduling containers across a fleet of Docker machines and later discovering where those services have been deployed. Docker Swarm will provide the solutions.

12.2 Introducing Docker Swarm

Unless you've worked in a distributed systems environment before or built out dynamic deployment topologies, the problems that Docker Swarm solves will require some effort to understand. This section dives deeply into those and explains how Swarm addresses each from a high level.

When people encounter the first problem, they may ask themselves, "Which machine should I choose to run a given container?" Organizing the containers you need to run across a fleet of machines is not a trivial task. It used to be the case that we would deploy different pieces of software to different machines. Using the machine as a unit of deployment made automation simpler to reason about and implement, given existing tooling. When a machine is your unit of deployment, figuring out which machine should run a given program is not a question you need to answer. The answer is always "a new one." Now, with Linux containers for isolation and Docker for container tooling, the remaining major concerns are efficiency of resource usage, the performance characteristics of each machine's hardware, and network locality. Selecting a machine based on these concerns is called *scheduling*.

After someone figures out a solution to the first problem, they'll immediately ask, "Now that my service has been deployed somewhere in my network, how can other services find it?" When you delegate scheduling to an automated process, you can't know where services will be deployed beforehand. If you can't know where a service will be located, how can other services use it? Traditionally, server software uses DNS to resolve a known name to a set of network locations. DNS provides an appropriate lookup interface, but writing data is another problem altogether. Advertising the availability of a service at a specific location is called registration, and resolving the location of a named service is called service discovery.

In this chapter you learn how Swarm solves these problems by building a Swarm cluster with Docker Machine, exploring scheduling algorithms, and deploying the Coffee API example.

12.2.1 Building a Swarm cluster with Docker Machine

A Swarm cluster is made up of two types of machines. A machine running Swarm in management mode is called a manager. A machine that runs a Swarm agent is called a node.

In all other ways, Swarm managers and nodes are just like any other Docker machines. These programs require no special installation or privileged access to the

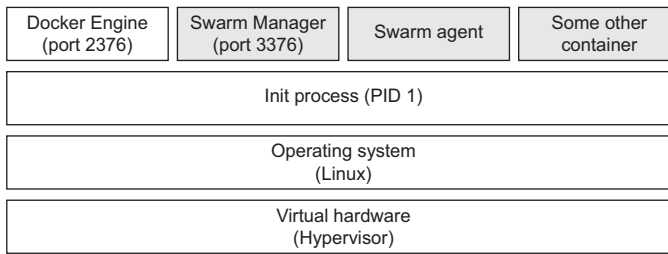


Figure 12.3 Swarm, Swarm Manager, and another program run in containers on an otherwise typical virtual machine running the Docker engine.

machines. They run in Docker containers. Figure 12.3 illustrates the computing stack of a typical Swarm manager machine. It is running the manager program, the Swarm agent, and another container.

Docker Machine can provision Swarm clusters as easily as standalone Docker machines. The only difference is a small set of additional command-line parameters that are included when you use the `create` subcommand.

The first, `--swarm`, indicates that the machine being created should run the Swarm agent software and join a cluster. Second, using the `--swarm-master` parameter will instruct Docker Machine to configure the new machine as a Swarm manager. Third, every type of machine in a Swarm cluster requires a way to locate and identify the cluster it's joining (or managing). The `--swarm-discovery` parameter takes an additional argument that specifies the unique identifier of the cluster. Figure 12.4 illustrates a small cluster of machines and a standalone machine for contrast.

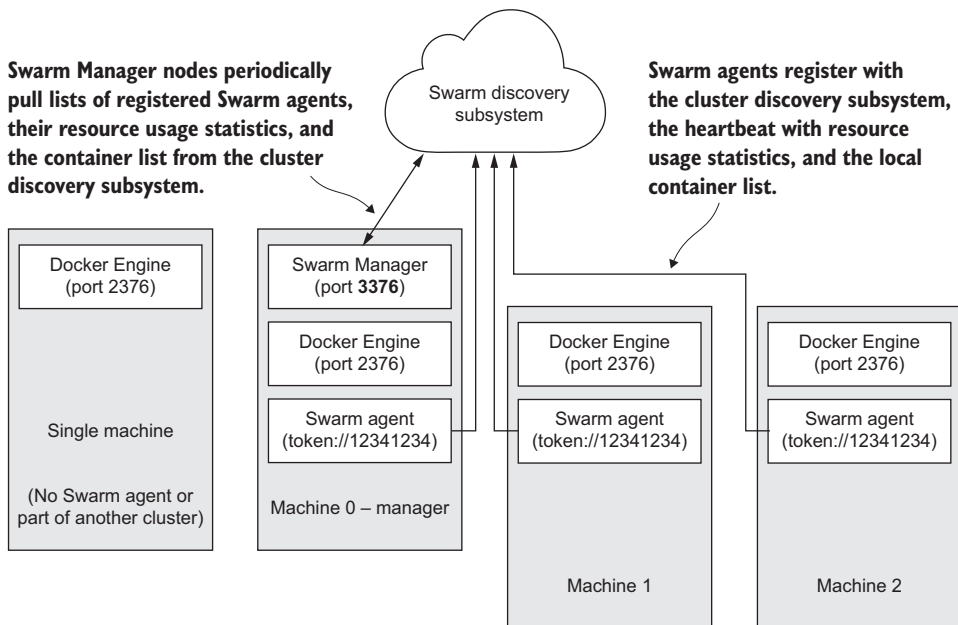


Figure 12.4 Swarm manager and agent interaction through a cluster discovery subsystem

In this illustration you can see that the Swarm agent on each node communicates with a Swarm discovery subsystem to advertise its membership in a cluster identified by token://12341234. Additionally, the single machine running the Swarm manager polls the Swarm discovery subsystem for an updated list of nodes in the cluster. Before diving into the Swarm discovery subsystem and other mechanics at work here, use Docker Machine to create a new Swarm cluster of your own. The next few commands will guide you through this process.

The first step in creating your own Swarm cluster is creating a cluster identifier. Like most subsystems abstracted by Docker, the Swarm discovery subsystem can be changed to fit your environment. By default, Swarm uses a free and hosted solution provided on Docker Hub. Run the following commands to create a new cluster identifier:

Create a new
local Docker

```
docker-machine create --driver virtualbox local
eval "$(docker-machine env local)"
```

Replace with equivalent command
appropriate for your shell

```
docker run --rm swarm create
```

The last command should output a hexadecimal identifier that looks like this:

```
b26688613694dbc9680cd149d389e279
```

Copy the resulting value and substitute that for `<TOKEN>` in the next three commands. The following set of commands will create a three-node Swarm cluster using virtual machines on your computer. Note that the first command uses the `--swarm-master` parameter to indicate that the machine being created should manage the new Swarm cluster:

```
docker-machine create \
  --driver virtualbox \
  --swarm \
  --swarm-discovery token://<TOKEN> \
  --swarm-master \
  machine0-manager
```

Note this flag

```
docker-machine create \
  --driver virtualbox \
  --swarm \
  --swarm-discovery token://<TOKEN> \
  machine1
```

```
docker-machine create \
  --driver virtualbox \
  --swarm \
  --swarm-discovery token://<TOKEN> \
  machine2
```

These machines can be identified as part of a Swarm cluster in output from the `ls` subcommand of `docker-machine`. The name of the cluster manager is included in the column labeled `SWARM` for any node in a cluster:

NAME	... URL	SWARM
machine0-manager	tcp://192.168.99.106:2376	machine0-manager (manager)
machine1	tcp://192.168.99.107:2376	machine0-manager
machine2	tcp://192.168.99.108:2376	machine0-manager

A Docker client could be configured to work with any of these machines individually. They're all running the Docker daemon with an exposed TCP socket (like any other Docker machine). But when you configure your clients to use the Swarm endpoint on the master, you can start working with the cluster like one big machine.

12.2.2 *Swarm extends the Docker Remote API*

Docker Swarm manager endpoints expose the Swarm API. Swarm clients can use that API to control or inspect a cluster. More than that, though, the Swarm API is an extension to the Docker Remote API. This means that any Docker client can connect directly to a Swarm endpoint and treat a cluster as if it were a single machine.

Figure 12.5 illustrates how Swarm manager delegates work—specified by Docker clients—to nodes in the cluster.

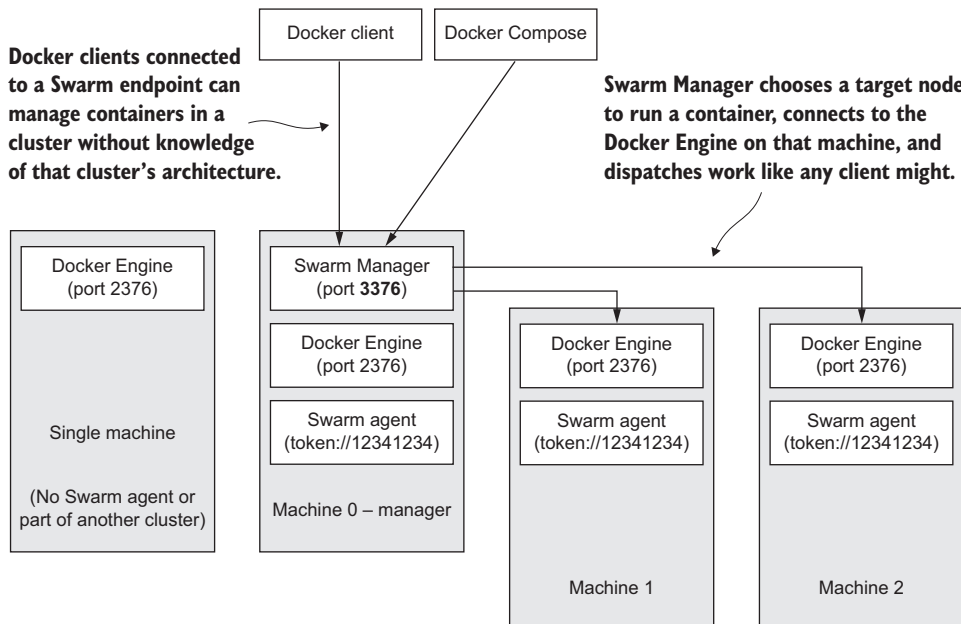


Figure 12.5 Deploying a container in a cluster requires no knowledge of the cluster because the Swarm API extends the Docker Remote API.

The implementation of the Docker Remote API provided by Swarm is very different from the Docker Engine. Depending on the specific feature, a single request from a client may impact one or many Swarm nodes.

Configure your environment to use the Swarm cluster that you created in the last section. To do so, add the `--swarm` parameter to the `docker-machine env` subcommand. If you're using a POSIX-compatible shell, run the following command:

```
eval "$(docker-machine env --swarm machine0-manager)"
```

If you're using PowerShell, the run the following:

```
docker-machine env --swarm machine0-master | Invoke-Expression
```

When your environment is configured to access a Swarm endpoint, the `docker` command-line interface will use Swarm features. For example, using the `docker info` command will report information for the whole cluster instead of details for one specific daemon:

```
docker info
```

That output will look similar to the following:

```
Containers: 4
Images: 3
Role: primary
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 3
 machine0-manager: 192.168.99.110:2376
   ? Containers: 2
   ? Reserved CPUs: 0 / 1
   ? Reserved Memory: 0 B / 1.022 GiB
   ? Labels: executiondriver=native-0.2, kernelversion=4.0.9-...
 machine1: 192.168.99.111:2376
   ? Containers: 1
   ? Reserved CPUs: 0 / 1
   ? Reserved Memory: 0 B / 1.022 GiB
   ? Labels: executiondriver=native-0.2, kernelversion=4.0.9-...
 machine2: 192.168.99.112:2376
   ? Containers: 1
   ? Reserved CPUs: 0 / 1
   ? Reserved Memory: 0 B / 1.022 GiB
   ? Labels: executiondriver=native-0.2, kernelversion=4.0.9-...
CPUs: 3
Total Memory: 3.065 GiB
Name: 942f56b2349a
```

Notice that this sample output includes a configuration summary for the cluster, a list of the nodes in the cluster, and a description of the resources available on each node. Before moving on, take a moment to reflect on what's happening here. This is the first evidence you've seen that the nodes in your cluster are advertising their endpoint and that the manager is discovering those nodes. Further, all this information is specific to

If you want to make sure that the images you use are pulled on each of the machines in the cluster, you can use the `pull` subcommand. Doing so will eliminate any warm-up latency if a container is scheduled on a node that doesn't have the required image:

```
docker pull dockerinaction/ch12_painted
```

This command will launch a pull operation on each node:

```
machine0-manager: Pulling dockerinaction/ch12_painted:latest... :  
    downloaded  
machine1: Pulling dockerinaction/ch12_painted:latest... : downloaded  
machine2: Pulling dockerinaction/ch12_painted:latest... : downloaded
```

Similarly, removing containers will remove the named container from whichever machine it is located on, and removing an image will remove it from all nodes in the cluster. Swarm hides all this complexity from the user and in doing so frees them to work on more interesting problems.

Not every decision can be made in a vacuum, though. Different algorithms for scheduling containers can have significant impact on the efficiency and performance characteristics of your cluster. Next, you'll learn about different algorithms and hints the user can provide to manipulate the Swarm scheduler.

12.3 Swarm scheduling

One of the most powerful arguments for adopting Linux containers as your unit of deployment is that you can make more efficient use of your hardware and cut hardware and power costs. Doing that requires intelligent placement of containers on your fleet.

Swarm provides three different scheduling algorithms. Each has its own advantages and disadvantages. The scheduling algorithm is set when you create a Swarm manager. A user can tune the scheduling algorithms for a given Swarm cluster by providing constraints for specific containers.

Constraints can be set for each container, but because the scheduling algorithm is set on the Swarm manager, you need to specify that setting when you create your cluster. The Docker Machine `create` subcommand provides the `--swarm-strategy` parameter for this purpose. The default selection is `spread`.

12.3.1 The Spread algorithm

A Swarm cluster that uses the Spread algorithm will try to schedule containers on under-used nodes and spread a workload over all nodes equally. The algorithm specifically ranks all the nodes in the fleet by their resource usage and then ranks those with the same resource rank according to the number of containers each is running. The machine with the most resources available and fewest containers will be selected to run a new container. Figure 12.6 illustrates how three equal-size containers might be scheduled in a cluster with three similar nodes.

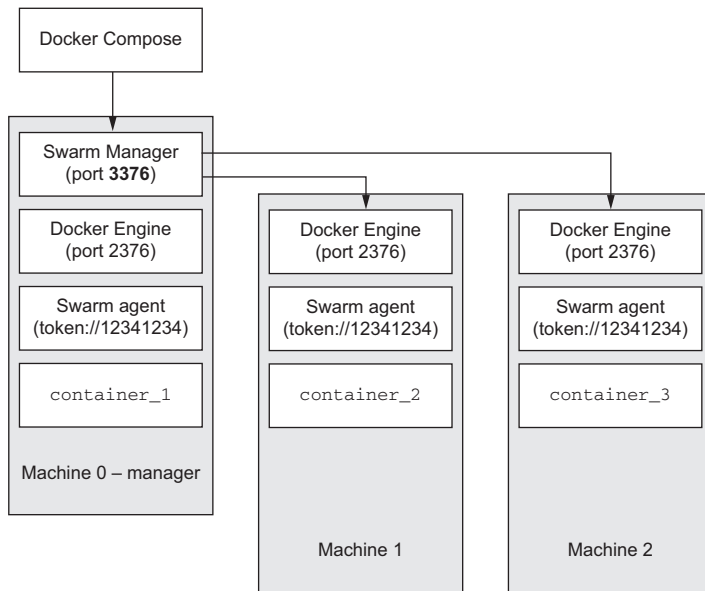


Figure 12.6 The Spread algorithm spreads a workload evenly across the nodes in a cluster.

You can see in figure 12.6 that each of the three scheduled containers was placed on a separate host. At this point, any of these three machines would be a candidate to run a fourth container. The chosen machine will be running two containers when that fourth container is scheduled. Scheduling a fifth container will cause the cluster to choose between the two machines that are running only a single container. The selection rank of a machine is determined by the number of containers it's running relative to all other machines in the cluster. Those machines running fewer containers will have a higher rank. In the event that two machines have the same rank, one will be chosen at random.

The Spread algorithm will try to use your whole fleet evenly. Doing so minimizes the potential impact of random machine failure and ties individual machine congestion with fleet congestion. Consider an application that runs a large, varying number of replica containers.

Create a new Compose environment description in `flock.json` and define a single service named `bird`. The `bird` service periodically paints `bird` on standard out:

```
bird:
  image: dockerinaction/ch12_painted
  command: bird
  restart: always
```

Watch Swarm distribute 10 copies of the `bird` service across the cluster when you use Compose to scale up:

Create some birds → `docker-compose -f flock.yml scale bird=10`
`docker ps`

← Check out container distribution

The output for the `ps` command will include 10 containers. The name of each container will be prefixed with the name of the machine where it has been deployed. These will have been evenly distributed across the nodes in the cluster. Use the following two commands to clean up the containers in this experiment:

```
docker-compose -f flock.yml kill
docker-compose -f flock.yml rm -vf
```

This algorithm works best in situations where resource reservations have been set on containers and there is a low degree of variance in those limits. As the resources required by containers and the resources provided by nodes diversify, the Spread algorithm can cause issues. Consider the distribution in figure 12.7.

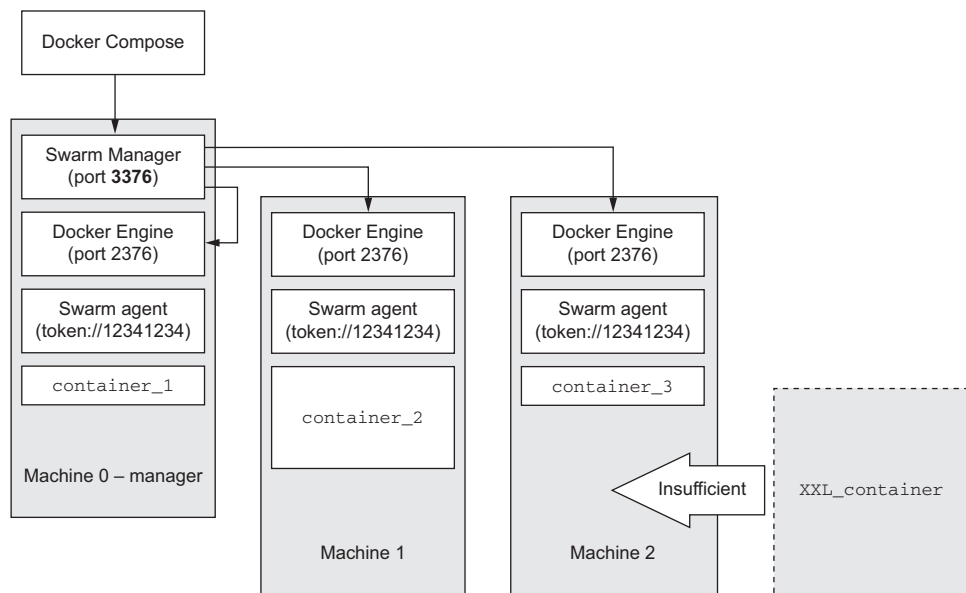


Figure 12.7 A new container is unable to be scheduled due to a poor distribution.

Figure 12.7 illustrates how introducing higher variance in the resources required by containers can cause poor performance of the Spread algorithm. In this scenario, the new container, `XXL_container`, can't be scheduled because no machine has sufficient resources. In retrospect, it's clear that the scheduler might have avoided this situation if it had scheduled `container_3` on either Machine0 or Machine1. You can avoid this type of situation without changing the scheduling algorithm if you use filters.

12.3.2 Fine-tune scheduling with filters

Before the Swarm scheduler applies a scheduling algorithm, it gathers and filters a set of candidate nodes according to the Swarm configuration and the needs of the

container. Each candidate node will pass through each of the filters configured for the cluster. The set of active filters used by a cluster can be discovered with the `docker info` command. When a `docker` client is configured for a Swarm endpoint, output from the `info` subcommand will include a line similar to the following:

```
Filters: affinity, health, constraint, port, dependency
```

A cluster with these filters enabled will reduce the set of candidate nodes to those that have any affinity, constraint, dependency, or port required by the container being scheduled and to those in which the node is healthy.

An affinity is a requirement for colocation with another container or image. A constraint is a requirement on some machine property like kernel version, storage driver, network speed, disk type, and so on. Although you can use a set of predefined machines for defining constraints, you can also create constraints on any label that has been applied to a node's daemon. A dependency is a modeled container dependency such as a link or shared volume. Finally, a port filter will reduce the set of candidate nodes to those with the requested host port available.

The poor distribution described in figure 12.7 could have been avoided with labeled nodes and a label constraint defined on the containers. Figure 12.8 illustrates a better way to configure the system.

In figure 12.8, both Machine0 and Machine1 were created with the label `size=small`, and Machine2 was labeled `size=xxl`. When containers 1–3 were created,

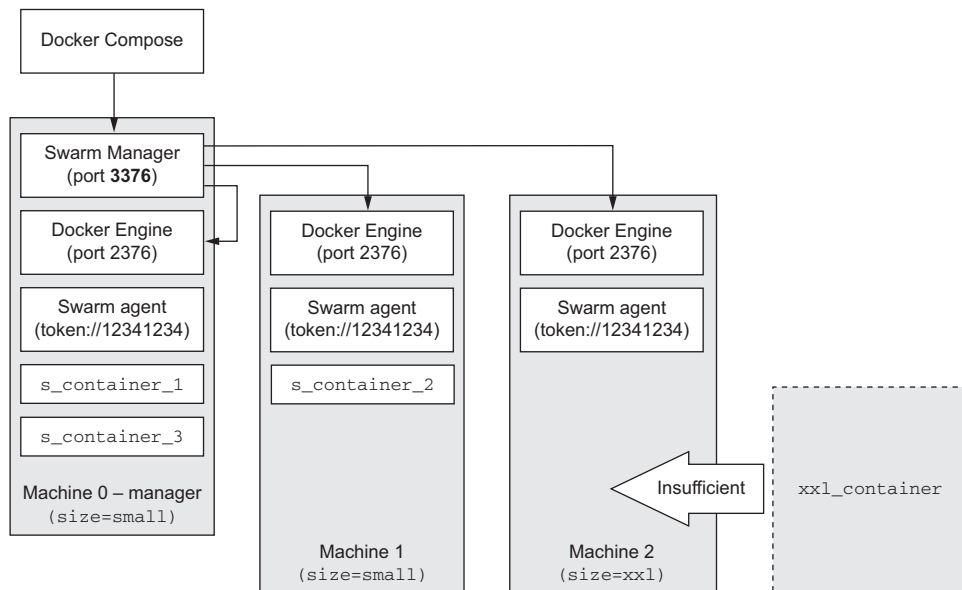


Figure 12.8 Node labels and constrained containers prune the scheduling candidates to appropriate nodes.

they provided an environment variable that indicated a constraint on nodes labeled `size=small`. That constraint narrows the node candidate pool to Machine0 or Machine1. Those containers are spread across those nodes as in other examples. When the `xxl_container` is created with a constraint on nodes labeled `size=xxl`, the candidate pool is narrowed to a single node. As long as that one node is not filtered for any other reason, it will be scheduled on Machine2.

You can label the nodes that you create in your cluster by setting the `--engine-label` parameter of the `docker-machine create` command. For example, you'd use the following command to create a new node labeled with `size=small` in your cluster:

```
docker-machine create -d virtualbox \
  --swarm \
  --swarm-discovery token://<YOUR TOKEN> \
  --engine-label size=small \
  little-machine
```

← Apply an engine label

```
docker-machine create -d virtualbox \
  --swarm \
  --swarm-discovery token://<YOUR TOKEN> \
  --engine-label size=xxl \
  big-machine
```

← Apply an engine label

In addition to whatever labels you might apply to nodes, containers can specify constraints on standard properties that all nodes will specify by default:

- `node`—The name or ID of the node in the cluster
- `storagedriver`—The name of the storage driver used by the node
- `executiondriver`—The name of the execution driver used by the node
- `kernelversion`—The version of the Linux kernel powering the node
- `operatingsystem`—The operating system name on the node

Each node provides these values to the Swarm master. You can discover what each node in your cluster is reporting with the `docker info` subcommand.

Containers communicate their affinity and constraint requirements using environment variables. Each constraint or affinity rule is set using a separate environment variable. A rule that the `xxl_container` might have applied in the previous example would look like this:


```
docker run -d -e constraint:size==xxl \
  -m 4G \
  -c 512 \
  postgres
```

← Constraint environment variable

Constraint rules are specified using the prefix `constraint:` on the environment variable. To set a container affinity rule, set an environment variable with a prefix of `affinity:` and an appropriate affinity definition. For example, if you want to schedule a container created from the `nginx:latest` image, but you want to make sure that

the candidate node already has the image installed, you would set an environment variable like so:


```
docker run -d -e affinity:image==nginx \
  -p 80:80 \
  nginx
```



Affinity environment variable

Any node where the `nginx` image has been installed will be among the candidate node set. If, alternatively, you wanted to run a similar program, like `ha-proxy`, for comparison and needed to make sure that program runs on separate nodes, you could create a negated affinity:

```
docker run -d -e affinity:image!=nginx \
  -p 8080:8080 \
  haproxy
```




Anti-affinity environment variable

This command would eliminate any node that contains the `nginx` image from the candidate node set. The two rules you've seen so far use different rule operators, `==` and `!=`, but other components of the rule syntax make it highly expressive. An affinity or constraint rule is made of a key, an operator, and a value. Whereas keys must be known and fully qualified, values can have one of three forms:

- Fully qualified with alphanumeric characters, dots, hyphens, and underscores (for example, `my-favorite.image-1`)
- A pattern specified in glob syntax (for example, `my-favorite.image-*`)
- A full (Go-flavored) regular expression (for example, `/my-[a-z]+\image-[0-9]+/`)

The last tool for creating effective rules is the soft operator. Add a tilde to the end of the operator when you want to make a scheduling suggestion instead of a rule. For example, if you wanted to suggest that Swarm schedule an NGINX container on a node where the image has already been installed but schedule it if the condition can't be met, then you would use a rule like the following:

```
docker run -d -e affinity:image==~nginx \
  -p 80:80 \
  nginx
```



Suggested affinity environment variable

Filters can be used to customize any of the scheduling algorithms. With foresight into your anticipated workload and the varying properties of your infrastructure, filters can be used to great effect.

The Spread algorithm's defining trait (even fleet usage by container volume) will always be applied to the filtered set of nodes, and so it will always conflict with one cloud feature. Automatically scaling the number of nodes in a cluster is feasible when only some nodes become unused. Until that condition is met, the underlying technology (be that Amazon Web Services EC2 or any other) will be unable to scale down your cluster. This results in low utilization of an oversized fleet. Adopting the BinPack scheduling algorithm can help in this case.

12.3.3 Scheduling with BinPack and Random

Two other scheduling algorithms you should understand are BinPack and Random. The BinPack scheduling algorithm prefers to make the most efficient use of each node before scheduling work on another. This algorithm uses the fewest number of nodes required to support the workload. Random provides a distribution that can be a compromise between Spread and BinPack. Each node in the candidate pool has an equal opportunity of being selected, but that doesn't guarantee that the distribution will realize evenly across that pool.

There are a few caveats worth reviewing before you adopt either algorithm over Spread. Figure 12.9 illustrates how the BinPack algorithm might schedule three containers.

BinPack can make informed decisions about packing efficiency only if it knows the resource requirements of the container being scheduled. For that reason, BinPack makes sense only if you're dedicated to creating resource reservations for the containers in your system. Using resource-isolation features such as memory limits, CPU weights, and block IO weights will isolate your containers from neighboring resource abuses. Although these limits don't create local reservations for resources, the Swarm scheduler will treat them as reservations to prevent overburdening any one host.

BinPack addresses the initial problem you encountered with the Spread algorithm. BinPack will reserve large blocks of resources on nodes by prioritizing efficient use of each node. The algorithm takes a greedy approach, selecting the busiest node with

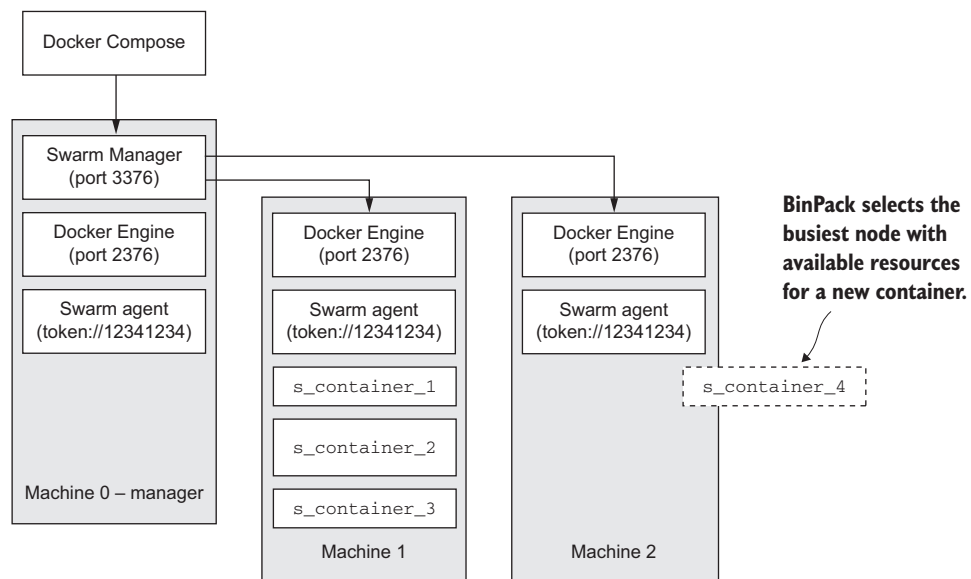


Figure 12.9 BinPack scheduling on a new node only when the busiest node has insufficient resources to take more work

the fewest resources that are still sufficient to meet the requirements of the container being scheduled. This is sometimes called a best fit.

BinPack is particularly useful if the containers in your system have high variance in resource requirements or if your project requires a minimal fleet and the option of automatically downsizing. Whereas the Spread algorithm makes the most sense in systems with a dedicated fleet, BinPack makes the most sense in a wholly virtual machine fleet with scale-on-demand features. This flexibility is gained at the cost of reliability.

BinPack creates a minimal number of maximally critical nodes. This distribution increases the likelihood of failure on the few busy nodes and increases the impact of any such failure. This may be an acceptable trade, but it's certainly one you should keep in mind when building critical systems.

The Random algorithm provides a compromise between Spread and BinPack. It relies on probability alone for selecting from the candidate nodes. In practice, this means it's possible for your fleet to simultaneously contain busy nodes and idle nodes. Figure 12.10 illustrates one possible distribution of three containers in a three-node candidate set.

This algorithm will probably distribute work fairly over a fleet and probably accommodate high variance in resource requirements in your container set. But *probably* means that you have no guarantees. It's possible that the cluster will contain hot spots or that it will be used inefficiently and lack large resource blocks to accommodate large containers. Entropy-focused engineers may gravitate toward this algorithm so that the greater system must account for these possibilities.

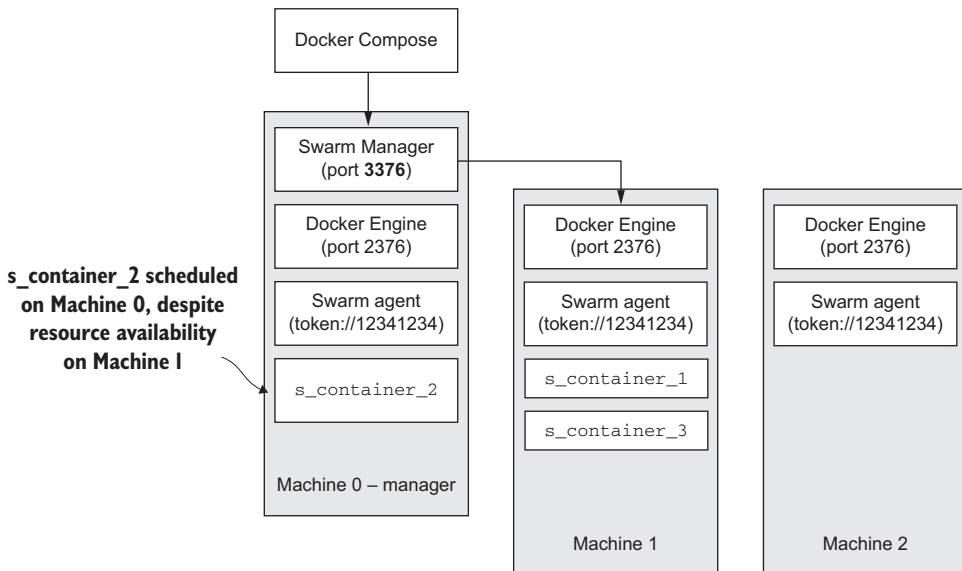


Figure 12.10 The Random scheduling algorithm considers only probability in selecting a machine.

As Swarm matures and more data from nodes becomes available, new scheduling algorithms may emerge. The project itself is young, but these three provide the fundamental building blocks for robust systems. Scheduling lets the system choose where your containers will run, and the last problem Swarm tackles is helping those containers locate each other with a service discovery abstraction.

12.4 Swarm service discovery

Any distributed system requires some mechanism to locate its pieces. If that distributed system is made up of several processes on the same machine, they need to agree on some named shared memory pool or queue. If the components are designed to interact over a network, they need to agree on names for each other and decide on a mechanism to resolve those names. Most of the time, networked applications rely on DNS for name-to-IP address resolution.

Docker uses container links to inject static configuration into a container's name-resolution system. In doing so, contained applications need not be aware that they're running in a container. But a stand-alone Docker engine has no visibility into services running on other hosts, and service discovery is limited to the containers running locally.

Alternatively, Docker allows a user to set the default DNS servers and search domains for each container or every container on a host. That DNS server can be any system that exposes a DNS interface. In the last few years, several such systems have emerged, and a rich ecosystem has evolved to solve service discovery in a multi-host environment.

As those systems evolved, Docker announced the Swarm project and a common interface for this type of system. The goal of the Swarm project is to provide a "batteries included" but optional solution for clustering containers. A major milestone for this project is abstracted service discovery in a multi-host environment. Delivering on that milestone requires the development of several technologies and enhancements to the Docker Engine.

As you reach the end of this book, remember that you're diving into the tail of an unfinished story. These are some of the most rapidly advancing features and tools in the system. You'll see each of these approaches to service discovery in the wild. Just as with most other pieces of the Docker toolset, you're free to adopt or ignore the parts that make sense in your situation. The remainder of this section should help you make an informed decision.

12.4.1 Swarm and single-host networking

The Docker Engine creates local networks behind a bridge network on every machine where it's installed. This topic is explored in depth in chapter 5. Situated as a container deployed on a Docker node, it's beyond the scope of a Swarm agent to restructure that network in reaction to the discovery of other Swarm nodes and containers in the cluster. For that reason, if a Swarm cluster is deployed on Docker machines that

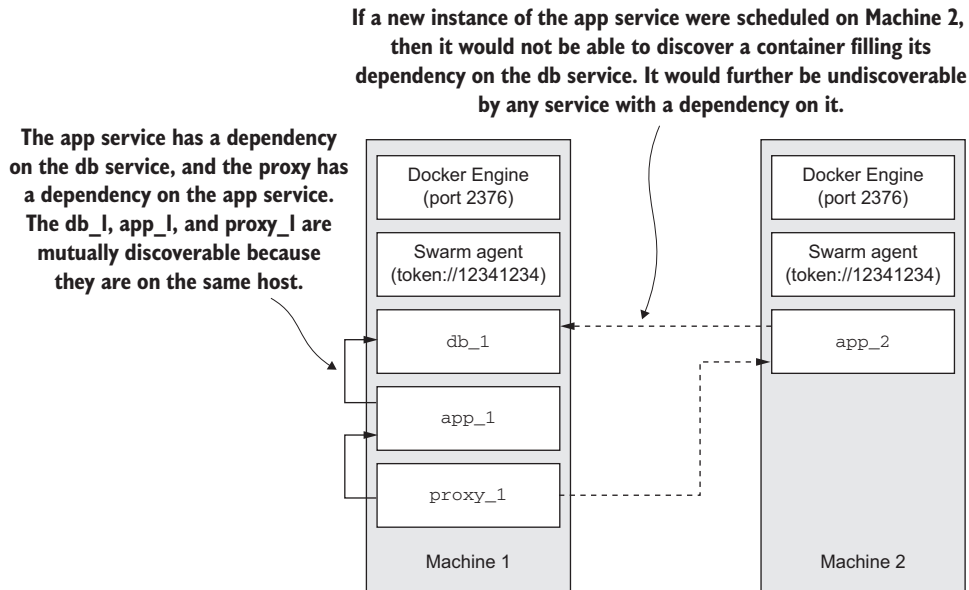


Figure 12.11 A three-tiered application deployed in a cluster with single-host networking

operate single-host networks, then containers deployed with Swarm can only discover other containers running on the same host. Figure 12.11 illustrates how single-host networking limits the candidate set for a scheduler.

The dependency filter ensures that a container will never be scheduled on a host where one of its dependencies is undiscoverable. You can try this for yourself by deploying the Coffee API on your existing Swarm cluster.

First, use Git to clone the Compose environment description for a rich Coffee API example:

```
git clone git@github.com:dockerinaction/ch12_coffee_api.git
```

After you run that command, change into the new directory and make sure that your Docker environment is pointed at your Swarm cluster:

```
cd ch12_coffee_api
eval "$(docker-machine env machine0-manager) "
```

Once you've set up your environment, you're ready to launch the example using Docker Compose. The following command will start the environment in your Swarm cluster:

```
docker-compose up -d
```

Now use the `docker` CLI to view the distribution of the new containers on your cluster. The output will be very wide due to the machine named prefixing all the container

names and aliases. For example, a container named “bob” running on machine1 will be displayed as “machine1/bob.” You may want to redirect or copy the output to a file for review without wrapping lines:

```
docker ps
```

If you have the `less` command installed, you can use the `-S` parameter to chop the long lines and arrows to navigate:

```
docker ps | less -S
```

← The `less` command may not be available on your system

In examining the output, you’ll notice that every container is running on the same machine even though your Swarm is configured to use the Spread algorithm. Once the first container that’s a dependency of another was scheduled, the dependency filter excluded any other node from the candidate pool for the others.

Now that the environment is running, you should be able to query the service. Note the name of the machine where the environment has been deployed and substitute that name for `<MACHINE>` in the following `cURL` command:

```
curl http://$(docker-machine ip <MACHINE>):8080/api/coffeeshops/
```

If you don’t have `cURL` on your system, you can use your web browser to make a similar request. The output from the request should be familiar:

```
{
  "coffeeshops": []
}
```

Take time to experiment by scaling the coffee service up and down. Examine where Swarm schedules each container. When you’ve finished with the example, shut it down with `docker-compose stop` and remove the containers with `docker-compose rm -vf`.

Clustering an application is viable for some use cases in spite of this limitation, but the most common use cases are underserved. Server software typically requires multi-host distribution and service discovery. The community has built and adopted several new and existing tools to fill the absence of a solution integrated with Docker.

12.4.2 Ecosystem service discovery and stop-gap measures

The primary interface for network service discovery is DNS. Although software providing DNS has been around for some time, traditional DNS server software uses heavy caching, struggles to provide high-write throughput, and typically lacks membership monitoring. These systems fail to scale in systems with frequent deployments. Modern systems use distributed key-value databases that support high-write throughput, membership management, and even distributed locking facilities.

Examples of modern software include etcd, Consul, ZooKeeper, and Serf. These can be radically different in implementation, but they’re all excellent service discovery

Service names are resolved by an external service discovery system that has been integrated through the DNS configuration of the containers. In these topologies containers are either left to register themselves with the external system, or a bystander watching individual daemon event streams will handle registration of new containers.

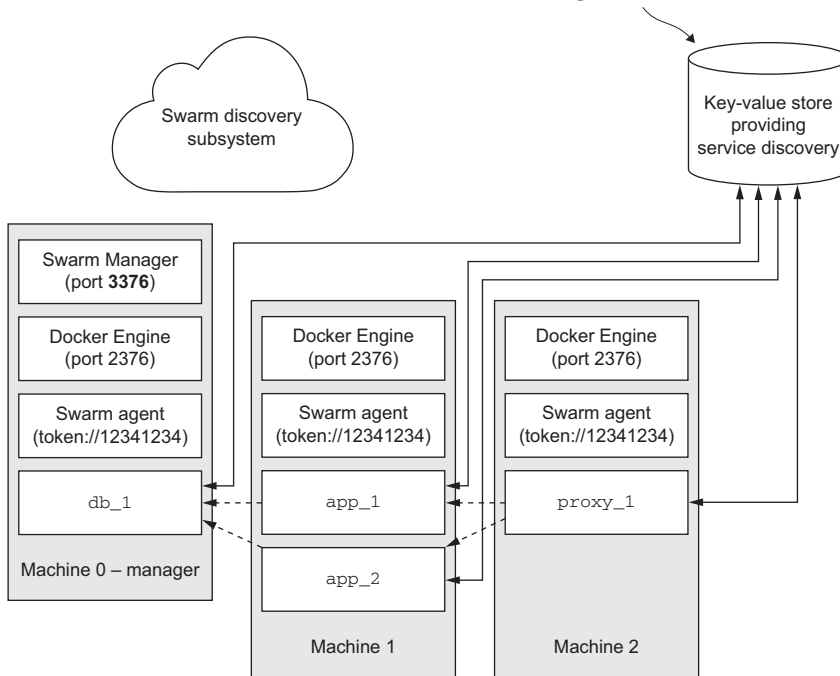


Figure 12.12 Containers on the fleet are responsible for registration and service discovery with an external tool like etcd, Consul, ZooKeeper, or Serf.

providers. Each tool has nuances, advantages, and disadvantages that are beyond the scope of this book. Figure 12.12 illustrates how containers integrate directly with an external service discovery tool over DNS or another protocol like HTTP.

These ecosystem tools are well understood and battle-tested, but integrating this infrastructure concern at the container layer leaks implementation details that should remain hidden by some basic abstraction. Integrating an application inside a container with a specific service-discovery mechanism diminishes the portability of that application. The ideal solution would integrate service registration and discovery at the clustering or network layer provided by Docker Engine and Docker Swarm. With multi-host, networking-enabled Docker Engine and an overlay network technology integrated with a pluggable key-value store, that can become a reality.

12.4.3 Looking forward to multi-host networking

The experimental branch of Docker Engine has abstracted networking facilities behind a pluggable interface. That interface is implemented by a number of drivers

Docker engine creates an overlay network and abstracts the service discovery mechanism from the containers running on any one host. While the underlying mechanics of service discovery and registration are the same, the abstraction requires less specialization of individual containers.

A Swarm cluster with multi-host networking enabled acts like a single machine.

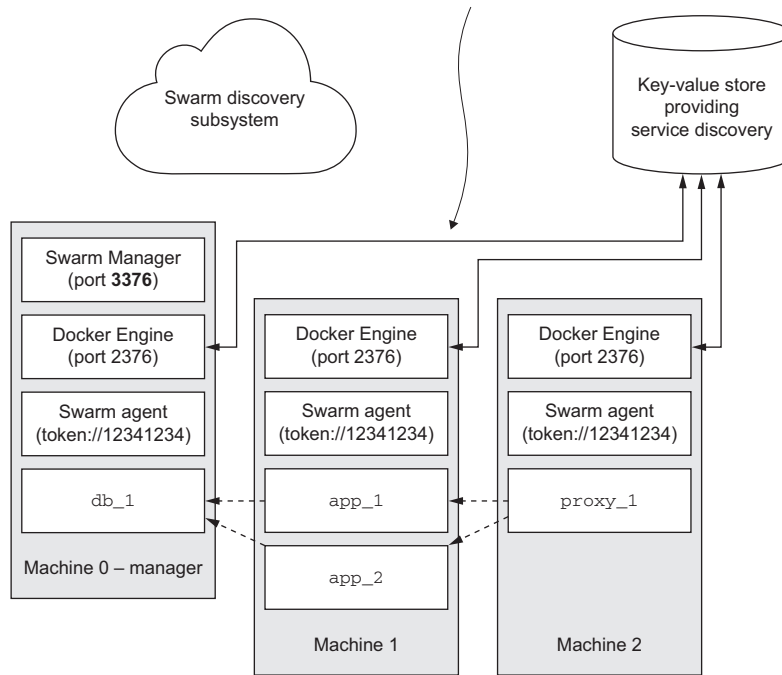


Figure 12.13 Swarm on top of Docker Engine with multi-host networking enables a cluster to behave as a single machine and containers to act as equal members of a network.

including bridge, host, and overlay. The bridge and host drivers implement the single-host networking features you're already familiar with. The overlay driver implements an overlay network with IP encapsulation or VXLAN. The overlay network provides routable IP addresses for every container managed by a Docker machine configured to use the same key-value store. Figure 12.13 illustrates the interactions in building such a network.

With an overlay network in place, each container gets a unique IP address that's routable from any other container in the overlay network. The application can stop doing work to identify or advertise its host IP address and map container ports to host ports. All that work is performed in the infrastructure layer provided by Docker and the integrated key-value store.

While we wait for multi-host networking to land in the release branch of Docker Engine and multi-host Swarm node provisioning with Docker Machine, we can get by with direct integrations to other ecosystem projects. When this does land, it will be a major relief to developers and system architects alike.

12.5 Summary

Both Docker Machine and Docker Swarm provide functionality that enhances the applications for the Docker Engine. These and other related technologies will help you apply what you have learned in this book as you grow from using Docker on a single computer into managing fleets of containers distributed across many computers. It is important to have a thorough understanding of these points as you grow into this space:

- A user can use Docker Machine to create a local virtual machine or machine in the cloud with a single `create` command.
- The Docker Machine `env` and `config` subcommands can be used to configure Docker clients to work with remote Docker Engines that were provisioned with Docker Machine.
- Docker Swarm is a protocol that is backward-compatible with the Docker Remote API and provides clustering facilities over a set of member nodes.
- A Swarm manager program implements the Swarm API and handles container scheduling for the cluster.
- Docker Machine provides flags for provisioning both Swarm nodes and managers.
- Docker Swarm provides three different scheduling algorithms that can be tuned through the use of filters.
- Labels and other default attributes of a Docker Engine can be used as filtering criteria via container scheduling constraints.
- Container scheduling affinities can be used to place containers on the same host as other containers or images that match a provided pattern or expression.
- When any Docker client is configured to communicate with a Swarm endpoint, that client will interact with the entire Swarm as if it were a single Docker machine.
- Docker Swarm will schedule dependent containers on the same node until multi-host networking is released or you provide another service-discovery mechanism and disable the dependency Swarm filter.
- Multi-host networking will abstract container locality from the concerns of applications within Docker containers. Each container will be a host on the overlay network.

Symbols

!= rule operator 266
- (hyphen) character 141
== rule operator 266

A

abstractions 11
access control, distribution method and 171
accesskey subproperty 214
accountkey subproperty 213
accountname subproperty 213
ACTIVE column 253
active subcommand 253
ADD instruction 153, 155
-add-host 99
addr property 218
affinity prefix 265
Amazon Web Services CloudFront storage
 middleware 228
Amazon's Simple Storage Service. *See* S3
Apache Cassandra project 58–61
Apache Web Server 205
apache2 37
api service 242–243
AppArmor 120
apt-get tool 129
AUDIT_CONTROL capability 117
AUDIT_WRITE capability 117
AUTH command 219
auth section 198, 207
automation. *See* build automaton
availability, distribution method and 171
aws program 156

AWS Simple Storage Service 214
azure property 213
Azure Storage container 213
Azure, hosted remote storage with 213–214

B

bind mount volumes 62–64
BinPack algorithm 267–269
-bip flag 93
bird service 262
blob (binary large object) storage, durable 212–216
 hosted remote storage
 with Amazon's Simple Storage Service
 (S3) 214–215
 with Microsoft's Azure 213–214
 internal remote storage with RADOS 216
blobdescriptor subproperty 218
boot2docker ip 107
boot2docker ssh command 116
bridged containers 85–94
 creating 85–86
 custom name resolution 86–89
 inter-container communication 91–92
 modifying bridge interface 92–94
 opening inbound communication 89–91
bridges 80
bucket subproperty 214
build automation 145–166
 Dockerfile 149–156
 file system instructions 153–156
 metadata instructions 150–153
 packaging Git with 146–149
 hardened application images 161–166
 content addressable image identifiers 162

- build automation (*continued*)
 - SUID and SGID permissions 165–166
 - user permissions 163–164
- injecting downstream build-time behavior 156–158
- using startup scripts and multiprocess containers 159–161
 - environmental preconditions validation 159–160
 - initialization processes 160–161
- build command 237
- build key 244

C

- c flag 141
- cache subproperty 218
- CAIID (content addressable image identifier) 162
- calaca service 235–236
- Calaca web interface 222
- cap-add flag 118
- cap-drop flag 118
- capabilities, feature access using 117–118
- Cassandra project 58–61
- centralized registries, enhancements for 198–211
 - adding authentication layer 205–208
 - before going to production 210–211
 - client compatibility 208–210
 - configuring TLS on reverse proxy 201–204
 - creating reverse proxy 199–201
- Ceph 215–216
- certificate property 207
- Cgroups 6
- chroot() function 6
- chunksize property 214, 216
- chunksize subproperty 214, 216
- CloudFront middleware 220
- cloudfront middleware 221
- CMD instruction 153–155, 173
- Coffee API 237–242, 245
- coffee service 238, 241–244
- Compose environment 262
- confidentiality, distribution method and 171–172
- config subcommands 274
- Configuration variables 211
- constraint prefix 265
- container subproperty 213
- containers 15–40
 - bridged 85–94
 - creating 85–86
 - custom name resolution 86–89
 - inter-container communication 91–92
 - modifying bridge interface 92–94
 - opening inbound communication 89–91
 - building environment-agnostic systems 30–35
 - environment variable injection 32–35
 - read-only file systems 30–32
 - building images from 127–132
 - committing new image 130–131
 - configurable image attributes 131–132
 - packaging Hello World 128
 - preparing packaging for Git 129
 - reviewing file system changes 129–130
 - cleaning up 39
 - container file system abstraction and isolation 53–54
 - container patterns using volumes 71–76
 - data-packed volume containers 73–74
 - polymorphic container pattern 74–76
 - volume container pattern 72–73
 - container-independent data management 58
 - creating 17–18
 - durable, building 35–39
 - automatically restarting containers 36
 - keeping containers running with supervisor and startup processes 37–39
 - eliminating metaconflicts 24–30
 - container state and dependencies 28–30
 - flexible container identification 25–28
 - inter-container dependencies 97–103
 - environment modifications 100–101
 - link aliases 99–100
 - link nature and shortcomings 102–103
 - links for local service discovery 97–99
 - interactive, running 18–19
 - joined 94–96
 - multiprocess 159–161
 - networking 81–83
 - four network container archetypes 82–83
 - local Docker network topology 81–82
 - open 96
 - open memory container 111–112
 - output of 20–21
 - PID namespace and 21–24
 - running software in, for isolation 5–6
 - running with full privileges 118–119
 - sharing IPC primitives between 110–111
 - shipping containers 7
 - use-case-appropriate containers 122–123
 - applications 122
 - high-level system services 123
 - low-level system services 123
 - vs. virtualization 5
 - with enhanced tools 119–121
 - fine-tuning with Linux Containers (LXC) 121
 - specifying additional security options 120–121
- Containers numbers 260
- content addressable image identifier. *See* CAIID

- COPY instruction 153, 155, 204
 - core.windows.net 213
 - cp command 74
 - cpu-shares flag 107
 - CPU, limits on resources 107–109
 - cpuset-cpus flag 108–109
 - create command 274
 - create subcommand 256, 261
 - cron tool 123
 - cURL command 239, 271
 - curl command 196, 201, 204
 - cURL command-line tool 195
 - customized registries, running 192–228
 - centralized registries, enhancements for 198–211
 - adding authentication layer 205–208
 - before going to production 210–211
 - client compatibility 208–210
 - configuring TLS on reverse proxy 201–204
 - creating reverse proxy 199–201
 - durable blob (binary large object) storage 212–216
 - hosted remote storage with Amazon’s Simple Storage Service (S3) 214–215
 - hosted remote storage with Microsoft’s Azure 213–214
 - internal remote storage with RADOS 216
 - integrating through notifications 221–228
 - personal registry 194–198
 - customizing image 197–198
 - reintroducing image 194–195
 - V2 Registry API 195–197
 - scaling access and latency improvements 217–221
 - integrating metadata cache 217–219
 - streamlining blob transfer with storage middleware 219–221
 - link aliases 99–100
 - link nature and shortcomings 102–103
 - links for local service discovery 97–99
 - detach flag 18
 - device flag 109
 - devices, access to 109
 - dialtimeout property 219
 - disabled attribute 226
 - distribution
 - choosing method of 169–172
 - distribution spectrum 169
 - selection criteria 170–172
 - image source distribution workflows 188–191
 - manual image publishing and distribution 183–188
 - private registries 179–182
 - consuming images from 182
 - using registry image 181–182
 - using hosted registries 172–179
 - private hosted repositories 177–179
 - public projects with automated builds 175–177
 - public repositories 172–175
 - Distribution project 193–194, 197, 205, 210, 222
 - Distribution registry 222
 - Distribution-based registry 227
 - DNS (Domain Name System) 86
 - Docker 3, 12–14
 - containers and 4–5
 - importance of 11
 - overview 4–7
 - problems solved by 7–10
 - getting organized 8–9
 - improving portability 9–10
 - protecting computer 10
 - use of, where/when 11–12
 - docker build command 50, 137, 147, 151, 175, 184, 189, 201, 204, 214–215, 237, 240
 - docker CLI 270
 - docker client 264
 - docker command 233, 250, 260
 - docker command-line tool 15–16, 18, 247, 252, 259
 - docker commit command 130, 132, 135, 137
 - Docker Compose 231–247, 254
 - building, starting, and rebuilding services 237–240
 - docker-compose.yml file 243–247
 - iteration and persistent state 242–243
 - linking problems and network 243
 - overview 232–236
 - scaling and removing services 240–242
 - docker cp command 140
- ## D
-
- d option 234
 - daemons 18
 - data service 246
 - data volume container archetype 246
 - data-packed volume containers 73–74
 - DATABASE_PORT variable 99
 - db service 238, 245–246
 - dbstate service 238, 246
 - dbus tool 123
 - Debian 197
 - dependencies
 - container state and 28–30
 - inter-container
 - environment modifications 100–101

- docker create command 105–106
 - cap-drop flag 118
 - cpu-shares flag 107
 - cpuset-cpus flag 108
 - lxc-conf flag 121
 - privileged flag 118
 - security-opt flag 120
 - user (–u) flag 114
 - volume flag 154
- docker diff command 134
- Docker Engine 269, 272
- docker exec command 22, 74
- docker export command 140, 184
- docker history command 139
- Docker Hub 13, 17, 44–48, 172, 222, 245
- docker images command 51–52, 130
- docker import command 140–141, 184
- docker info command 259, 264
- docker info subcommand 265
- docker inspect command 31, 65, 69, 91, 113, 152, 251
- docker kill command 39
- docker load command 49, 184, 187
- docker login command 45, 173, 177
- docker logout command 45
- docker logs command 20, 234
- Docker Machine 249–255
 - building and managing 250–252
 - building Swarm cluster using 255–258
 - configuring Docker clients to work with remote daemons 252–255
- docker port command 91
- docker ps –a 39
- docker ps command 20, 28, 39, 91, 243
- docker pull command 44, 51
- docker push command 173
- Docker Remote API 253, 258–261
- docker rename command 25
- docker rm –f command 39–40
- docker rm command 40, 70, 234
- docker rmi command 52
- docker run –link flag 245
- docker run –v flag 246
- docker run command 44, 105–106, 197, 211, 234, 245–246
 - cap-drop flag 118
 - cpu-shares flag 107
 - cpuset-cpus flag 108
 - lxc-conf flag 121
 - net option 96
 - privileged flag 118
 - security-opt flag 120
 - user (–u) flag 114
 - volume flag 154
- flags of 84, 86, 88, 90–91
- docker save command 49, 184
- docker search command 45–46
- docker start command 29
- docker stop command 21, 39–40
- Docker Swarm 255–261
 - building Swarm cluster with Docker Machine 255–258
 - extension of Docker Remote API using 258–261
 - scheduling algorithms 261–269
 - BinPack algorithm 267–269
 - fine-tune scheduling with filters 263–266
 - Random algorithm 267–269
 - Spread algorithm 261–263
 - service discovery 269–274
 - ecosystem service discovery and stop-gap measures 271–272
 - multi-host networking 272
 - single-host networking 269–271
- docker tag command 53, 137, 142
- docker top command 37
- docker-compose command 233, 250
- docker-compose command-line program 232, 247
- docker-compose kill command 234
- docker-compose logs command 234
- docker-compose ps command 240–242
- docker-compose rm –v 271
- docker-compose rm command 234, 242
- docker-compose stop command 234, 271
- docker-compose up command 233, 235, 238
- docker-compose up –d command 236, 240
- docker-compose.yml file 243–247
- docker-machine command 251–253
- docker-machine command-line program 250
- docker-machine create command 265
- docker-machine env command 253
- docker-machine env subcommand 259
- docker-machine help create command 249
- docker-machine ssh command 251
- Dockerfile 149–156, 200, 237–238
 - distributing project with, on GitHub 189
 - file system instructions 153–156
 - installing software from 50
 - metadata instructions 150–153
 - packaging Git with 146–149
- dockerfile key 244
- Dockerfiles 197
- .dockerignore file 21–24, 150
- Domain Name System. *See* DNS
- downstream build-time behavior 156–158

E

- e flag 211
- echo command 72

- ecosystem service discovery 271–272
- email flag 173
- encrypt property 214
- engine-label parameter 265
- entrypoint flag 38, 131
- ENTRYPOINT instruction 148, 152–154
- entrypoints 38
- env (-e) flag 33, 151
- ENV instruction 151–153
- env subcommand 253, 274
- env_file key 244
- environment key 244
- environment variables 211
- environment-agnostic systems, building 30–35
 - environment variable injection 32–35
 - read-only file systems 30–32
- es-pump container 224
- Ethernet interface 78, 86
- exec-driver=lxc option 121
- executiondriver 265
- exporting, flat file systems 140–141
- EXPOSE command 151–153
- expose flag 91, 98
- expose key 245
- extends key 246

F

- f flag 39
- f option 111
- features 117–118
- file (-f) flag 147
- file key 246
- file systems
 - changes to, reviewing 129–130
 - flat, importing and exporting 140–141
 - structure of 54
- filesystem property 212
- filters, fine-tune scheduling with 263–266
- flat file systems 140–141
- format (-f) option 113
- freegeoip program 43
- FROM instruction 152, 157–158, 162, 173
- FTP (File Transfer Protocol), sample distribution
 - infrastructure using 185–188
- ftp-transport container 186
- full privileges, running containers with 118–119

G

- GET request 196
- Git 237, 270
 - packaging with Dockerfile 146–149
 - preparing packaging for 129

- GitHub, distributing project with Dockerfile
 - on 189
- gosu program 164

H

- ha-proxy program 266
- hardened application images, building 161–166
 - content addressable image identifiers 162
 - SUID and SGID permissions 165–166
 - user permissions 163–164
- Hello World, packaging 128
- hello-swarm container 260
- help command 250
- high-level system services 123
- host-dependent sharing 66–67
- hosted registries, distribution 172–179
 - private hosted repositories 177–179
 - public projects with automated builds 175–177
 - public repositories 172–175
- hosted remote storage
 - with Amazon's Simple Storage Service (S3) 214–215
 - with Microsoft's Azure 213–214
- HOSTNAME environment variable 240
- hostname flag 86, 88
- Htpasswd 205
- HTTP (Hypertext Transfer Protocol) 78, 195
- HTTP HEAD request 196
- http section, of configuration file 198
- Hypertext Transfer Protocol See HTTP
- hyphen (-) character 141

I

- id command 113
- idletimeout property 219
- image key 245
- image layers 51–52
- image-dev container 130
- images 127, 135–144
 - building from containers 127–132
 - committing new image 130–131
 - configurable image attributes 131–132
 - packaging Hello World 128
 - preparing packaging for Git 129
 - reviewing file system changes 129–130
 - consuming from private registries 182
 - exporting and importing flat file systems 140–141
 - hardened application images 166
 - content addressable image identifiers 162
 - SUID and SGID permissions 165–166
 - user permissions 163–164

- images (*continued*)
 - loading as files 48–50
 - size of 138–139
 - union file systems 132–135
 - versioning best practices 141
- Images number 260
- images subcommand 254
- importing, flat file systems 140–141
- in flock.json 262
- indexes 7
- info subcommand 55, 264
- InfoSec conversations 119
- inmemory value 218
- inspect subcommand 113, 250–251
- installing software 41–55
 - from Dockerfile 50
 - identifying software 42–44
 - installation files and isolation 51–55
 - container file system abstraction and isolation 53–54
 - file system structure 54
 - image layers 51–52
 - layer relationships 53
 - union file systems, weaknesses of 54
 - loading images as files 48–50
 - searching Docker Hub for repositories 44–48
 - using alternative registries 48
- integrity, distribution method and 171
- inter-container dependencies 97–103
 - environment modifications 100–101
 - link aliases 99–100
 - link nature and shortcomings 102–103
 - links for local service discovery 97–99
- interactive (–i) flag 18
- interactive containers, running 18–19
- interfaces 78–79
- internal remote storage 216
- IP (Internet Protocol) 78
- ip subcommand 251
- ipc flag 110, 112
- IPC namespace 6
- isolation 104–123
 - containers with enhanced tools 119–121
 - fine-tuning with Linux Containers (LXC) 121
 - specifying additional security options 120–121
 - feature access 117–118
 - resource allowances 105–109
 - access to devices 109
 - CPU 107–109
 - memory limits 105–107
 - running container with full privileges 118–119
 - running software in containers for 5–6
 - shared memory 109–112

- open memory container 111–112
- sharing IPC primitives between containers 110–111
- use-case-appropriate containers 122–123
 - applications 122
 - high-level system services 123
 - low-level system services 123
- users 112–117
 - Linux user namespace 112
 - run-as user 113–115
 - volumes and 115–117

J

- Java Virtual Machine *See* JVM
- joined containers 94–96
- JSON objects 224
- JVM (Java Virtual Machine) 9

K

- k option 204
- kernelversion 265
- key property 207
- kill program 37
- kill subcommand 252

L

- label flag 151
- LABEL instruction 151–152
- labels key 244
- LAMP (Linux, Apache, MySQL PHP) stack 37
- latest tag 137, 143, 194
- layer relationships 53
- layers 42, 135–139
- less command 271
- libraries 45
- links
 - aliases for 99–100
 - for local service discovery 97
 - links for local service discovery 99
 - nature of 102–103
 - shortcomings of 102–103
- links command 245
- Linux containers. *See* LXC
- Linux Security Modules. *See* LSM
- Linux user namespace 112
- Linux USR namespace 164
- local tag 194
- log section, of configuration file 198
- logs subcommand 260
- longevity, distribution method and 170–171
- loopback interface 78, 86

low-level system services 123
 ls subcommand 250, 253, 258
 LSM (Linux Security Modules) 120
 LXC (Linux Containers) 121, 261
 -lxc-conf flag 121

M

-m (-memory) flag 105, 130
 MAC_ADMIN capability 117
 MAC_OVERRIDE capability 117
 Machine. *See* Docker Machine
 mailer-base image 150
 MAINTAINER instruction 152
 maxactive property 219
 maxidle property 219
 Memcached technology 79
 memory
 limits on 105–107
 shared 109–112
 open memory container 111–112
 sharing IPC primitives between
 containers 110–111
 metaconflicts, eliminating 24–30
 container state and dependencies 28–30
 flexible container identification 25–28
 metadata cache, integrating 217–219
 metadata, Dockerfile and 150–153
 Microsoft's Azure, hosted remote storage
 with 213–214
 middleware property 221
 middleware section, of configuration file 198
 mmap() function 54
 MNT namespace 6
 mod_ubuntu container 133, 136–137
 multi-host networking 272
 multiprocess containers 159–161

N

-name flag 25
 NAMES column 260
 NAT 79–80
 -nB flags 205
 -net flag 84, 110
 NET namespace 6
 -net option 96
 NET_ADMIN capability 117
 networks 77–103
 bridged containers 85–94
 creating 85–86
 custom name resolution 86–89
 inter-container communication 91–92
 modifying bridge interface 92–94

 opening inbound communication 89–91
 closed containers 83–85
 container networking 81–83
 four network container archetypes 82–83
 local Docker network topology 81–82
 inter-container dependencies 97–103
 environment modifications 100–101
 link aliases 99–100
 link nature and shortcomings 102–103
 links for local service discovery 97–99
 interfaces 78–79
 joined containers 94–96
 multi-host 272
 NAT 79–80
 open containers 96
 overview 78–80
 port forwarding 79–80
 ports 78–79
 protocols 78–79
 single-host 269–271
 nginx image 266
 nginx:latest image 265
 -no-cache flag 149
 -no-dep flag 238–239
 Node.js 222–223
 NoSQL database, using volumes with 58–61
 notifications section, of configuration file 198

O

ONBUILD instruction 149, 156–158
 open containers 96
 operatingssystem 265
 -output (-o) 140

P

-P (-publish-all) flag 90
 -p=[] (-publish=[]) flag 90
 -p option 245
 -password flag 173
 password property 219
 PATH 197
 path property 207
 permissions 163–166
 personal registry 194–198
 customizing image 197–198
 reintroducing image 194–195
 V2 Registry API 195–197
 personal_registry container 201
 PID namespace 6
 polymorphic container pattern 74–76
 polymorphic tools 58
 pool property 219

- poolname property 216
- poolname subproperty 216
- portability, improving 9–10
- ports key 245
- ports, forwarding 78–80
- Postgres repository 245
- private hosted repositories 177–179
- private registries 179–182
 - consuming images from 182
 - using registry image 181–182
- privileged flag 118
- problems solved by Docker
 - getting organized 8–9
 - improving portability 9–10
 - protecting computer 10
- protocols 78–79
- proxy service 239, 243, 245
- ps command 22, 263
- ps subcommand 260
- public and private software distribution 168
- public projects with automated builds 175–177
- public repositories 172–175
- pull subcommand 261
- pull type 226
- pump service 236
- push command 194
- Python package manager 232
- Python-based application 244

Q

- quiet (-q) flag 148

R

- RADOS (Reliable Autonomic Distributed Object Store) 215–216
- rados storage property 216
- Random algorithm 267–269
- read-only file systems 30–32
- readtimeout property 219
- realm property 207, 213
- redis property 218–219
- redis section, of configuration file 198
- region subproperty 214
- registries 7
 - alternative 48
 - hosted 172–179
 - private hosted repositories 177–179
 - public projects with automated builds 175–177
 - public repositories 172–175
 - private 179–182
 - consuming images from 182
 - using registry image 181–182

- See also* customized registries, running
- REGISTRY program 211
- registry program 197, 200
- REGISTRY_HTTP_DEBUG environment variable 211
- REGISTRY_HTTP_SECRET environment variable 211
- REGISTRY_LOG_LEVEL environment variable 211
- registry:2 image 227
- Reliable Autonomic Distributed Object Store. *See* RADOS
- remote daemons, configuring Docker clients to work with 252–255
- remote storage
 - hosted
 - with Amazon's Simple Storage Service (S3) 214–215
 - with Microsoft's Azure 213–214
 - internal, with RADOS 216
- reporting section, of configuration file 198
- repositories 42–43, 135–138
 - private hosted 177–179
 - public 172–175
 - searching Docker Hub for 44–48
- resource allowances 105–109
 - access to devices 109
 - CPU 107–109
 - memory limits 105–107
- restart flag 36, 107
- restarting containers 36
- RESTful API 195
- rm command 111, 234
- rm flag 60
- rm subcommand 252
- rootdirectory property 212, 214
- rsync tool 188
- run command 260
- RUN instruction 148, 153, 155, 164
- run-as user 113–115

S

- s option 55
- S parameter 271
- S3 (Amazon's Simple Storage Service), hosted remote storage with 214–215
- s3 storage property 214
- scheduling 255
- scheduling algorithms, Docker Swarm 261–269
 - BinPack algorithm 267–269
 - fine-tune scheduling with filters 263–266
 - Random algorithm 267–269
 - Spread algorithm 261–263

- scp subcommands 251
- secretkey subproperty 214
- secure property 214
- Secure Shell. *See* SSH
- secure subproperty 214
- security-opt flag 120
- security, InfoSec conversations 10, 119
- SELinux 120
- service discovery, Docker Swarm 269–274
 - ecosystem service discovery and stop-gap measures 271–272
 - multi-host networking 272
 - single-host networking 269–271
- service key 246
- SETPCAP capability 117
- SGID permission set 165–166
- shared memory 109–112
 - open memory container 111–112
 - sharing IPC primitives between containers 110–111
- sharing volumes 66–69
 - generalized sharing and volumes-from flag 67–69
 - host-dependent sharing 66–67
- shell flag 253
- shipping containers 7
- SIG_HUP signal 39
- SIG_KILL signal 39
- Simple Email Service example 155–156
- Simple Storage Service. *See* S3
- single-host networking 269–271
- size=small label 264–265
- size=xxl label 264–265
- software. *See* installing software
- solutions provided by Docker
 - getting organized 8–9
 - improving portability 9–10
 - protecting computer 10
- Spread algorithm 261–263
- spread default selection 261
- SSH (Secure Shell) 202
- ssh subcommand 251–252
- sshd tool 123
- start subcommand 252
- startup process 37–39
- startup scripts 159–161
 - environmental preconditions validation 159–160
 - initialization processes 160–161
- stop subcommand 252
- stop-gap measures 271–272
- storage property 218
- storage section, of configuration file 198

- storage subproperty 221
- storage-driver option 55
- storagedriver 265
- SUID permission set 165–166
- supervisor process 37–39
- supervisord program 37
- swarm 256
- SWARM column 258
- swarm image 260
- swarm parameter 259
- Swarm scheduler 267
- swarm-discovery parameter 256
- swarm-master parameter 256–257
- swarm-strategy parameter 261
- Swarm. *See* Docker Swarm
- SYS_ADMIN capability 117
- SYS_MODULE capability 117
- SYS_NICE capability 117
- SYS_PACCT capability 117
- SYS_RAWIO capability 117
- SYS_RESOURCE capability 117
- SYS_TIME capability 117
- SYS_TTY capability 117
- SYSLOG capability 117
- syslogd tool 123

T

- t option, docker build 50
- tag (–t) flag 147
- tags 43–44, 135–138
- TCP (Transmission Control Protocol) 79
- TLS (transport layer security), configuring on reverse proxy 201–204
- tls section 207
- <TOKEN> 257
- Transmission Control Protocol *See* TCP
- transport layer security. *See* TLS
- transportation, distribution method and 170
- tty (–t) flag 18

U

- UFS (union file system) 54, 127
- upgrade subcommand 251
- use-case-appropriate containers 122–123
 - applications 122
 - high-level system services 123
 - low-level system services 123
- user (–u) flag 114–115
- USER instruction 153, 163–164
- user permissions 163–164
- username flag 173
- username subproperty 216

- users 112–117
 - Linux user namespace 112
 - run-as user 113–115
 - volumes and 115–117
- USR namespace 6
- UTS namespace 6

V

- v (–volume) option 63–64, 70–71
- v flag 40, 242
- v option 111
- /v2/ component 196
- V2 Registry API 195–197, 210
- v4auth property 214
- version section, of configuration file 198
- VERSION variable 151
- versioning, best practices 141
- VirtualBox 65
- virtualization, vs. containers 5
- visibility, distribution method and 170
- volume flag 154
- VOLUME instruction 153–154
- volumes 56–76
 - advanced container patterns using 71–76
 - data-packed volume containers 73–74
 - polymorphic container pattern 74–76
 - volume container pattern 72–73
 - container-independent data management using 58
 - managed volume life cycle 69–71
 - cleaning up volumes 70–71
 - volume ownership 69–70

- overview 57–61
- sharing 66–69
 - generalized sharing and volumes-from flag 67–69
 - host-dependent sharing 66–67
- types of 61–66
 - bind mount volumes 62–64
 - Docker managed volumes 64–66
- users and 115–117
 - using with NoSQL database 58–61
- volumes key 246
- volumes_from key 246
- volumes-from flag 67–69, 72–73

W

- WEB_HOST environment variable 160
- webhooks 175
- wget program 19
- wheezy tag 143
- whoami command 113, 165
- WORDPRESS_DB_HOST variable 33
- WORKDIR instruction 151–153
- writetimeout property 219

X

- XXL_container 263
- xxl_container 265

Y

- YAML file 227

Docker IN ACTION

Jeff Nickoloff



The idea behind Docker is simple. Create a tiny virtual environment, called a container, that holds just your application and its dependencies. The Docker engine uses the host operating system to build and account for these containers. They are easy to install, manage, and remove. Applications running inside containers share resources, making their footprints small.

Docker in Action teaches readers how to create, deploy, and manage applications hosted in Docker containers. After starting with a clear explanation of the Docker model, you will learn how to package applications in containers, including techniques for testing and distributing applications. You will also learn how to run programs securely and how to manage shared resources. Using carefully designed examples, the book teaches you how to orchestrate containers and applications from installation to removal. Along the way, you'll discover techniques for using Docker on systems ranging from dev-and-test machines to full-scale cloud deployments.

What's Inside

- Packaging containers for deployment
- Installing, managing, and removing containers
- Working with Docker images
- Distributing with DockerHub

Readers need only have a working knowledge of the Linux OS. No prior knowledge of Docker is assumed.

A software engineer, **Jeff Nickoloff** has presented Docker and its applications to hundreds of developers and administrators at Desert Code Camp, Amazon.com, and technology meetups.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/docker-in-action

“All there is to know about Docker. Clear, complete, and precise.”

—Jean-Pol Landrain
Agile Partner Luxembourg

“A compelling narrative for real-world Docker solutions. A must-read!”

—John Guthrie, Pivotal, Inc.

“An indispensable guide to understanding Docker and how it fits into your infrastructure.”

—Jeremy Gailor, Gracenote

“Will help you transition quickly to effective Docker use in complex real-world situations.”

—Peter Sellars, Fraedon

ISBN 13: 978-1-63343-023-5
ISBN 10: 1-63343-023-5



9 781633 430235