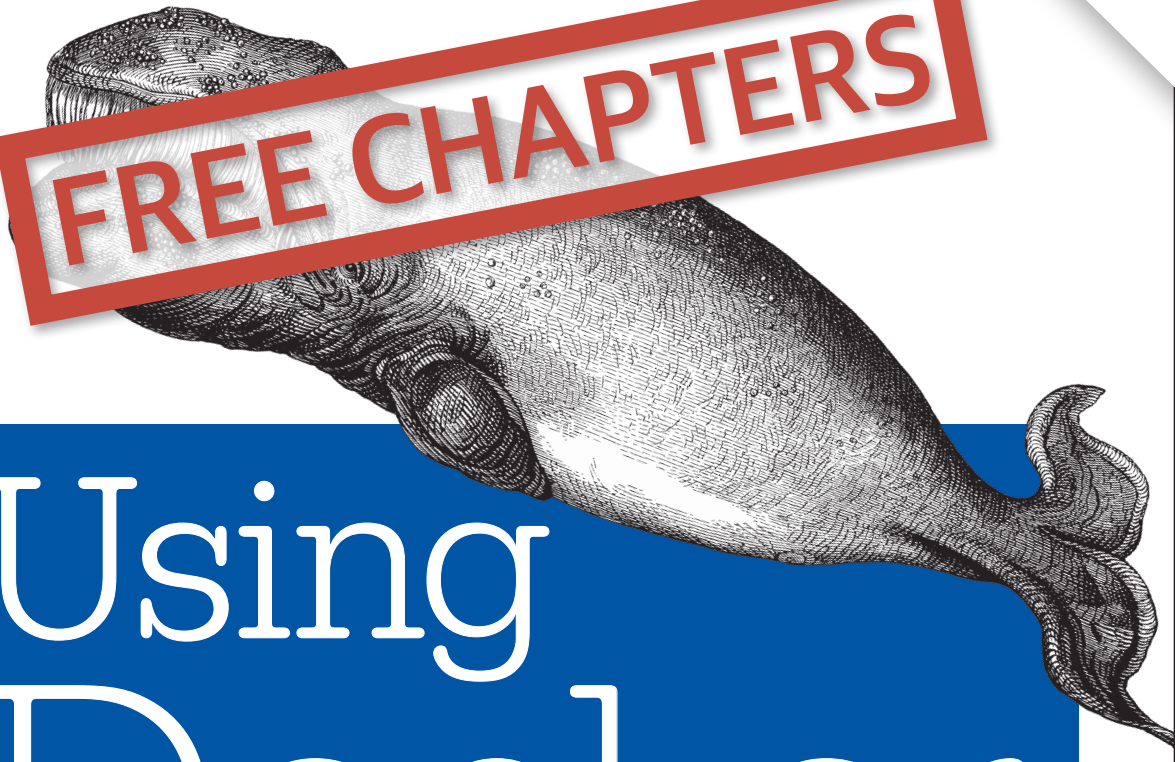
O'REILLY®

Compliments of NGINX

FREE CHAPTERS



Using Docker

DEVELOPING AND DEPLOYING SOFTWARE WITH CONTAINERS

Adrian Mouat

NGINX

Containers without chaos

Flawless application delivery with NGINX Plus

Advanced load balancing and automated routing

Application-aware health checks and container monitoring

Access controls and rate limiting to secure your applications

On-the-fly reconfiguration for scalable service discovery

Content caching for better availability and performance

Learn more at:
nginx.com/microservices

Using Docker

This Excerpt contains Chapters 8 and 9 of the book *Using Docker*. The full book is available at oreilly.com and through other retailers.

Adrian Mouat

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Using Docker
by Adrian Mouat

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Brian Anderson
Production Editor: Melanie Yarbrough
Copyeditor: Christina Edwards
Proofreader: Amanda Kersey
Indexer: WordCo Indexing Services
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

December 2015: First Edition

Revision History for the First Edition
2015-12-07: First Release
2016-04-08: Second Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491915769* for release details.

978-1-491-91576-9

[LSI]

Table of Contents

Foreword

Docker has enjoyed a meteoric rise in the world of IT. In just a few years, it has gone from being a small, open source project to being top-of-mind for CIOs at the world's largest enterprises. We recently surveyed the broad community of NGINX users, who run more than 165 million sites around the world, and found that a full two-thirds of them are investigating or already using containers in some way.

Of course, adopting a new technology like Docker is easier said than done. Using Docker provides a wealth of practical guidance on incorporating Docker into your full software development lifecycle. The final stages of the lifecycle—integrating Docker into your CI/CD workflows and ultimately deploying into production—can be particularly challenging. The chapters provided in this free ebook excerpt lead you through these complex and critical stages.

Once your containerized application is running in production, you face a whole host of new challenges—not the least of which is providing a stable entrypoint for your clients to connect to. Clients can't directly connect to dynamic endpoints like containers. This is where NGINX and NGINX Plus come into play.

If you are looking to deploy Docker-based applications, you need reverse proxies that are stable, easy to use, and reliable, like NGINX and NGINX Plus. Whether you want to perform essential services such as load balancing, caching, and rate limiting, or simply reduce complexity with DNS-based service discovery, NGINX and NGINX Plus help eliminate the need for manual work and provide application owners one less thing to worry about.

We're proud that NGINX is one of the most frequently downloaded applications on Docker Hub, with over 10 million pulls to date. We hope you enjoy this ebook excerpt, and that it helps you succeed as you deploy containers into production.

— Faisal Memon, Product Marketer,
NGINX, Inc.

CHAPTER 1

Continuous Integration and Testing with Docker

In this chapter, we're going to look into how Docker and Jenkins can be used to create a continuous integration (CI) workflow for building and testing our application. We'll also take a look at other aspects of testing with Docker and a brief look at how to test a microservices architecture.

Testing containers and microservices brings a few different challenges to testing. Microservices make for easy unit tests but difficult system and integration tests due to the increased number of services and network links. Mocking of network services becomes more relevant than the traditional mocking of classes in a monolithic Java or C# codebase. Keeping test code in images maintains the portability and consistency benefits of containers, but increases their size.

The code for this chapter is available from this book's GitHub. The tag `v0` is the identidock code as it was at the end of the last chapter, with later tags representing the progression of the code through this chapter. To get this version of the code:

```
$ git clone -b v0 \
https://github.com/using-docker/ci-testing/
...
```

Alternatively, you can download the code for any tag from the Releases page on the GitHub project.

Adding Unit Tests to Identidock

The first thing we should do is add some unit tests to our identidock codebase. These will test some basic functionality of our identidock code, with no reliance on external services.[1]

Start by creating the file *identidock/app/tests.py* with the following contents:

```
import unittest
import identidock


class TestCase(unittest.TestCase):

    def setUp(self):
        identidock.app.config["TESTING"] = True
        self.app = identidock.app.test_client()

    def test_get_mainpage(self):
        page = self.app.post("/", data=dict(name="Moby Dock"))
        assert page.status_code == 200
        assert 'Hello' in str(page.data)
        assert 'Moby Dock' in str(page.data)

    def test_html_escaping(self):
        page = self.app.post("/", data=dict(name='"><b>TEST</b><!--'))
        assert '<b>' not in str(page.data)


if __name__ == '__main__':
    unittest.main()
```

This is just a very simple test file with three methods:

setUp
: Initializes a test version of our Flask web application.

test_get_mainpage
: Test method that calls the URL / with the input "Moby Dock" for the name field. The test then checks that the method returns a 200 status code and the data contains the strings "Hello" and "Moby Dock."

test_html_escaping
: Tests that HTML entities are properly escaped in input.

Let's run these tests:

1 Many developers advocate a test-driven development (TDD) approach, where tests are written before the code that makes them pass. This book hasn't followed this approach, mainly for the sake of narrative.

```
$ docker build -t identidock .
...
$ docker run identidock python tests.py
.F
======================================================================
FAIL: test_html_escaping (__main__.TestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "tests.py", line 19, in test_html_escaping
    assert '<b>' not in str(page.data)
AssertionError

----------------------------------------------------------------------
Ran 2 tests in 0.010s

FAILED (failures=1)
```

Hmm, that's not good. The first test passed, but the second one has failed, because we're not escaping user input properly. This is a serious security issue that in a larger application can lead to data leaks and cross-site scripting attacks (XSS). To see the effect on the application, launch identidock and try inputing a name such as `"><b>pwned!</b><!--"`, including the quotes. An attacker could potentially inject malicious JavaScript into our application and trick users into running it.

Thankfully, the fix is easy. We just need to update our Python application to *sanitize* the user input by replacing HTML entities and quotes with escape codes. Update *identidock.py* so that it looks like:

```
from flask import Flask, Response, request
import requests
import hashlib
import redis
import html

app = Flask(__name__)
cache = redis.StrictRedis(host='redis', port=6379, db=0)
salt = "UNIQUE_SALT"
default_name = 'Joe Bloggs'


@app.route('/', methods=['GET', 'POST'])
def mainpage():

    name = default_name
    if request.method == 'POST':
        name = html.escape(request.form['name'], quote=True) ❶

    salted_name = salt + name
    name_hash = hashlib.sha256(salted_name.encode()).hexdigest()
    header = '<html><head><title>Identidock</title></head><body>'
    body = '''<form method="POST">
```

```
            Hello <input type="text" name="name" value="{0}">
            <input type="submit" value="submit">
            </form>
            <p>You look like a:
            <img src="/monster/{1}"/>
            '''.format(name, name_hash)
    footer = '</body></html>'

    return header + body + footer


@app.route('/monster/<name>')
def get_identicon(name):

    name = html.escape(name, quote=True) ❶
    image = cache.get(name)
    if image is None:
        print ("Cache miss", flush=True)
        r = requests.get('http://dnmonster:8080/monster/' + name + '?size=80')
        image = r.content
        cache.set(name, image)

    return Response(image, mimetype='image/png')

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

❶ Use the `html.escape` method to sanitize the user input.

Now if we build and test our application again:

```
$ docker build -t identidock .
...
$ docker run identidock python tests.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.009s

OK
```

Great—problem solved. You can verify this by restarting identidock with the new containers (remember to run `docker-compose build` to ensure Compose uses the new code) and trying to enter malicious input.[2] If we had used a real templating engine rather than simple string concatenation, the escaping would have been handled for us, avoiding this issue.

2 Embarrassingly, I never noticed this problem until the review stages of the book. I again learned the lesson that it is important to test even trivial-looking code and that it's best to use preexisting, proven code and tools where possible.

Now that we have some tests, we should extend our *cmd.sh* file to support automatically executing them. Replace *cmd.sh* with the following:

```
#!/bin/bash
set -e

if [ "$ENV" = 'DEV' ]; then
  echo "Running Development Server"
  exec python "identidock.py"
elif [ "$ENV" = 'UNIT' ]; then
  echo "Running Unit Tests"
  exec python "tests.py"
else
  echo "Running Production Server"
  exec uwsgi --http 0.0.0.0:9090 --wsgi-file /app/identidock.py \
             --callable app --stats 0.0.0.0:9191
fi
```

Now we can rebuild and run the tests by just changing the environment variable:

```
$ docker build -t identidock .
...
$ docker run -e ENV=UNIT identidock
Running Unit Tests
..
----------------------------------------------------------------------
Ran 2 tests in 0.010s

OK
```

There are more unit tests we could write. In particular, there are no tests for the `get_identicon` method. To test this method in a unit test, we would need to either bring up test versions of the dnmonster and Redis services, or use a *test double*. A test double stands in for the real service, and is commonly either a *stub*, which simply returns a canned answer (e.g., the stub for a stock price service might always return "42") or a *mock* that can be programmed with expectations for how it expects to be called (such as being called exactly once for a given transaction). For more information on test doubles, see the Python mock module as well as specialist HTTP tools such as Pact, Mountebank, and Mirage.

Including Tests in Images

In this chapter, we bundle the tests for identidock into the identidock image, which is in line with the Docker philosophy of using a single image through development, testing, and production. This also means we can easily check the tests on images running in different environments, which can be useful to rule out issues when debugging.

The disadvantage is that it creates a larger image—you have to include the test code plus any dependencies such as testing libraries. In turn, this also means there is a greater *attack surface*; it's possible, if unlikely, that an attacker could use test utilities or code to break the system in production.

In most cases, the advantages of the simplicity and reliability of using a single image will outweigh the disadvantages of the slightly increased size and theoretical security risk.

The next step is to get our tests automatically run in a CI server so we can see how our code could be automatically tested when code is checked in to source control and before moving to staging or production.

Using Containers for Fast Testing

All tests, and in particular unit tests, need to run quickly in order to encourage developers to run them often without getting stuck waiting on results. Containers represent a fast way to boot a clean and isolated environment, which can be useful when dealing with tests that mutate their environment. For example, imagine you have a suite of tests that make use of a service[3] that has been prepopulated with some test data. Each test that uses the service is likely to mutate the data in some way, either adding, removing, or modifying data. One way to write the tests is to have each test attempt to clean up the data after running, but this is problematic; if a test (or the cleanup) fails, it will pollute the test data for all following tests, making the source of the failure difficult to diagnose and requiring knowledge of the service being tested (it is no longer a black box). An alternative is to destroy the service after each test and start with a fresh one for each test. Using VMs for this purpose would be far too slow, but it is achievable with containers.

Another area of testing where containers shine is running services in different environments/configurations. If your software has to run across a range of Linux distribu-

3 Tests like these are likely to be system or integration tests rather than unit tests, or they could be unit tests in a nonmockist test configuration. Many unit test experts will advise that components such as databases should be replaced with mocks, but in situations where the component is stable and reliable, it is often easiest and sensible to use the component directly.

tions with different databases installed, set up an image for each configuration and you can fly through your tests. The caveat of this approach is that it won't take into account kernel differences between distributions.

Creating a Jenkins Container

Jenkins is a popular open source CI server. There are other options for CI servers and hosted solutions, but we'll use Jenkins for our web app, simply because of its popularity. We want to set up Jenkins so that whenever we push changes to our identidock project, Jenkins will automatically check out the changes, build the new images, and run some tests against them—both our unit tests and some system tests. It will then create a report on the results of the tests.

We'll base our solution on an image from the official Jenkins repository. I've used version 1.609.3, but new Jenkins releases are constantly appearing—feel free to try using a newer version, but I can't guarantee it will work without modification.

In order to allow our Jenkins container to build images, we're going to mount the Docker socket[4] from the host into the container, effectively allowing Jenkins to create "sibling" containers. An alternative to this is to use Docker-in-Docker (DinD), where the Docker container can create its own "child" containers. The two approaches are contrasted in Figure 1-1.

4 The Docker socket is the endpoint used for communicating between the client and the daemon. By default, this is an IPC socket accessed via the file */var/run/docker.sock*, but Docker also supports TCP sockets exposed via a network address and systemd-style sockets. This chapter assumes you are using the default socket at */var/run/docker.sock*. As the socket is accessed via a file descriptor, we can simply mount this endpoint as a volume in the container.

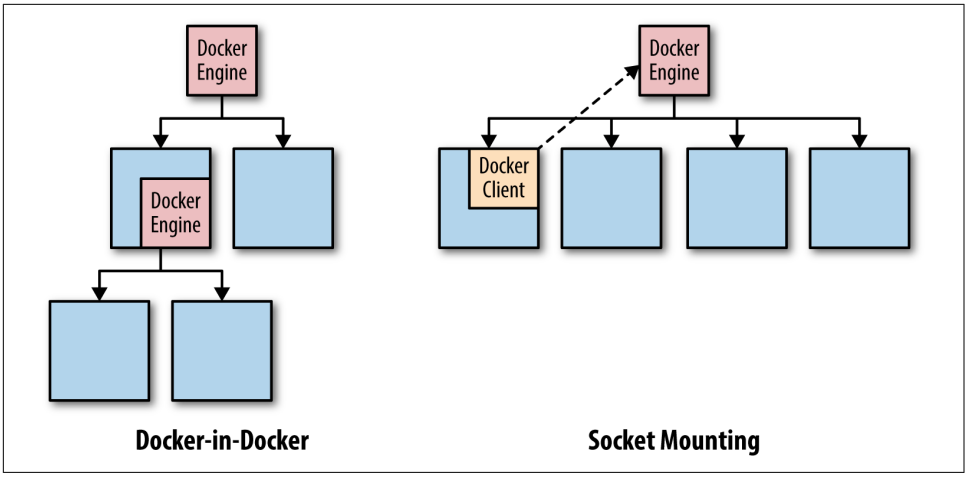


Figure 1-1. Docker-In-Docker versus socket mounting

Docker-in-Docker

Docker-in-Docker (or DinD) is simply running Docker itself inside a Docker container. There is some special configuration necessary to get this to work, primarily running the container in privileged mode and dealing with some filesystem issues. Rather than work this out yourself, it's easiest to use Jérôme Petazzoni's DinD project, which is available at *https://github.com/jpetazzo/dind* and describes all the required steps. You can quickly get started by using Jérôme's DinD image from the Docker Hub:

```
$ docker run --rm --privileged -t -i -e LOG=file jpetazzo/dind
ln: failed to create symbolic link '/sys/fs/cgroup/systemd/name=systemd':
Operation not permitted
root@02306db64f6a:/# docker run busybox echo "Hello New World!"
Unable to find image 'busybox:latest' locally
Pulling repository busybox
d7057cb02084: Download complete
cfa753dfea5e: Download complete
Status: Downloaded newer image for busybox:latest
Hello New World!
```

The major difference between DinD and the socket-mounting approach is that the containers created by DinD are isolated from the host containers; running `docker ps` in the DinD container will only show the containers created by the DinD Docker daemon. In contrast, running `docker ps` under the socket-mounting approach will show all the containers, regardless of where the command is run from.

In general, I prefer the simplicity of the socket-mounting approach, but in certain circumstances, you may want the extra isolation of DinD. If you do choose to run DinD, be aware of the following:

- You will have your own cache, so builds will be slower at first, and you will have to pull all your images again. This can be mitigated by using a local registry or mirror. Don't try mounting the build cache from the host; the Docker engine assumes exclusive access to this, so bad things can happen when shared between two instances.
- The container has to run in privileged mode, so it's not any more secure than the socket-mounting technique (if an attacker gains access, she can mount any device, including drives). This should get better in the future as Docker adds support for finer-grained privileges, which will allow users to choose the devices DinD has access to.
- DinD uses a volume for the */var/lib/docker* directory, which will quickly eat up your disk space if you forget to delete the volume when removing the container.

For more information on why you should be careful with DinD, see jpetazzo's GitHub article.

In order to mount the socket from the host, we need to make sure that the Jenkins user inside the container has sufficient access privileges. In a new directory called *identijenk*, create a Dockerfile with the following contents:

```
FROM jenkins:1.609.3

USER root
RUN echo "deb http://apt.dockerproject.org/repo debian-jessie main" \
        > /etc/apt/sources.list.d/docker.list \
      && apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 \
         --recv-keys 58118E89F3A912897C070ADBF76221572C52609D \
      && apt-get update \
      && apt-get install -y apt-transport-https \
      && apt-get install -y sudo \
      && apt-get install -y docker-engine \
      && rm -rf /var/lib/apt/lists/*
RUN echo "jenkins ALL=NOPASSWD: ALL" >> /etc/sudoers

USER jenkins
```

This Dockerfile takes the Jenkins base image, installs the Docker binary, and adds password-less `sudo` rights to the `jenkins` user. We intentionally haven't added `jenkins` to the `docker` group, so we will have to prefix all our Docker commands with `sudo`.

Don't Use the Docker Group

Instead of using `sudo`, we could have added the `jenkins` user to the host's `docker` group. The problem is that this requires us to find and use the GID of the `docker` group on the CI host and hardcode it into the Dockerfile. This makes our Dockerfile nonportable, as different hosts will have different GIDs for the `docker` group. To avoid the confusion and pain this can cause, it is preferable to use `sudo`.

Build the image:

```
$ docker build -t identijenk .
...
Successfully built d0c716682562
```

Test it:

```
$ docker run -v /var/run/docker.sock:/var/run/docker.sock \
        identijenk sudo docker ps
CONTAINER ID  IMAGE       COMMAND                CREATED         STATUS  ...
a36b75062e06  identijenk  "/bin/tini -- /usr/lo"  1 seconds ago  Up Less tha...
```

In the `docker run` command, we have mounted the Docker socket in order to connect to the host's Docker daemon. In older versions of Docker, it was common to also mount the Docker binary, rather than install Docker inside the container. This had the advantage of keeping the version of Docker on the host and in the container in sync. However, from version 1.7.1, Docker began using dynamic libraries, which means any dependencies also need to be mounted in the container. Rather than deal with the problems of finding and updating the correct libraries to mount, it is easier to simply install Docker in the image.

Now that we've got Docker working inside the container, we can install some other stuff we need to get our Jenkins' build working. Update the Dockerfile like so:

```
FROM jenkins:1.609.3

USER root
RUN echo "deb http://apt.dockerproject.org/repo debian-jessie main" \
          > /etc/apt/sources.list.d/docker.list \
      && apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 \
          --recv-keys 58118E89F3A912897C070ADBF76221572C52609D \
      && apt-get update \
      && apt-get install -y apt-transport-https \
      && apt-get install -y sudo \
      && apt-get install -y docker-engine \
      && rm -rf /var/lib/apt/lists/*
RUN echo "jenkins ALL=NOPASSWD: ALL" >> /etc/sudoers

RUN curl -L https://github.com/docker/compose/releases/download/1.4.1/\
docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose; \
```

```
    chmod +x /usr/local/bin/docker-compose ❶

USER jenkins
COPY plugins.txt /usr/share/jenkins/plugins.txt ❷
RUN /usr/local/bin/plugins.sh /usr/share/jenkins/plugins.txt
```

❶ Install Docker Compose, which we will use to build and run our images.

❷ Copy in and process a *plugins.txt* file, which defines a list of plugins to install in Jenkins.

Create the file *plugins.txt* in the same directory as the Dockerfile with the following contents:

```
scm-api:0.2
git-client:1.16.1
git:2.3.5
greenballs:1.14
```

The first three plugins set up an interface we can use to set up access to the Identidock project in Git. The "greenballs" plugin replaces the default Jenkins blue balls for successful builds with green ones.

We're now just about ready to launch our Jenkins container and start configuring our build, but first we should create a data container to persist our configuration:

```
$ docker build -t identijenk .
...
$ docker run --name jenkins-data identijenk echo "Jenkins Data Container"
Jenkins Data Container
```

We've used the Jenkins image for data container so we can be sure the permissions are set correctly. The container exits once the `echo` command completes, but as long as it's not deleted, it can be used in `--volumes-from` arguments. For more details on data containers, see ???.

Now we're ready to launch the Jenkins container:

```
$ docker run -d --name jenkins -p 8080:8080 \
    --volumes-from jenkins-data \
    -v /var/run/docker.sock:/var/run/docker.sock \
    identijenk
75c4b300ade6a62394a328153b918c1dd58c5f6b9ac0288d46e02d5c593929dc
```

If you open a browser at *http://localhost:8080*, you should see Jenkins initializing. In a moment, we'll set it up with a build and test for our identidock project. But first we need to make a minor change to the identidock project itself. Currently, the *docker-compose.yml* file for our project initializes a development version of identidock, but we are about to develop some system tests we want to run on something much closer to production. For this reason, we need to create a new file *jenkins.yml* that we will use to start the production version of identidock inside Jenkins:

```
identidock:
  build: .
  expose:
   - "9090" ❶
  environment:
    ENV: PROD ❷
  links:
   - dnmonster
   - redis

dnmonster:
  image: amouat/dnmonster:1.0

redis:
  image: redis:3.0
```

❶ As Jenkins lives in a sibling container, we don't need to publish ports on the host in order to connect to it. I've included the `expose` command mainly as documentation; you will still be able to access the identidock container from Jenkins without it, assuming you haven't played with the default networking settings.

❷ Set the environment to production.

This file needs to be added to the identidock repository that Jenkins will retrieve the source code from. You can either add it to your own repository if you configured one earlier or use the existing repository.

We're now ready to start configuring our Jenkins build. Open the Jenkins web interface running at *http://localhost:8080* and follow these instructions:

1. Click the Create new jobs link.
2. Enter **`identidock`** for the Item name, select Freestyle project, and click OK.
3. Configure the Source Code Management settings. If you used a public GitHub repository, you just need to select Git and enter the repository URL. If you used a private repository, you will need to set up credentials of some sort (several repositories, including BitBucket, have *deployment keys* that can be used to set up read-only access for this purpose). Alternatively, you can use the version available on GitHub.
4. Click "Add build step" and select "Execute shell." In the Command box, enter the following:

```
#Default compose args
COMPOSE_ARGS=" -f jenkins.yml -p jenkins "

#Make sure old containers are gone
sudo docker-compose $COMPOSE_ARGS stop ❶
sudo docker-compose $COMPOSE_ARGS rm --force -v
```

```
#build the system
sudo docker-compose $COMPOSE_ARGS build --no-cache
sudo docker-compose $COMPOSE_ARGS up -d

#Run unit tests
sudo docker-compose $COMPOSE_ARGS run --no-deps --rm -e ENV=UNIT identidock
ERR=$?

#Run system test if unit tests passed
if [ $ERR -eq 0 ]; then
  IP=$(sudo docker inspect -f {{.NetworkSettings.IPAddress}} \
          jenkins_identidock_1) ❷
  CODE=$(curl -sL -w "%{http_code}" $IP:9090/monster/bla -o /dev/null) || true ❸
  if [ $CODE -ne 200 ]; then
    echo "Site returned " $CODE
    ERR=1
  fi
fi

#Pull down the system
sudo docker-compose $COMPOSE_ARGS stop
sudo docker-compose $COMPOSE_ARGS rm --force -v

return $ERR
```

❶ Note that `sudo` is used to call Docker Compose, again because the Jenkins user isn't in the `docker` group.

❷ We use `docker inspect` to discover the IP address of the identidock container.

❸ We use `curl` to access the identidock service and check that it returns an HTTP 200 code indicating it is functioning correctly. Note that we are using the path `/monster/bla` to ensure that identidock can connect to the dnmonster service.

You can also get this code from GitHub. Normally, scripts like this would be checked into source control with other code, but for our example, simply pasting into Jenkins is enough.

Now, you should be able to test this out by clicking Save followed by Build Now. You can view the details of the build by clicking on the build ID and selecting Console Output. You should see something similar to Figure 1-2.

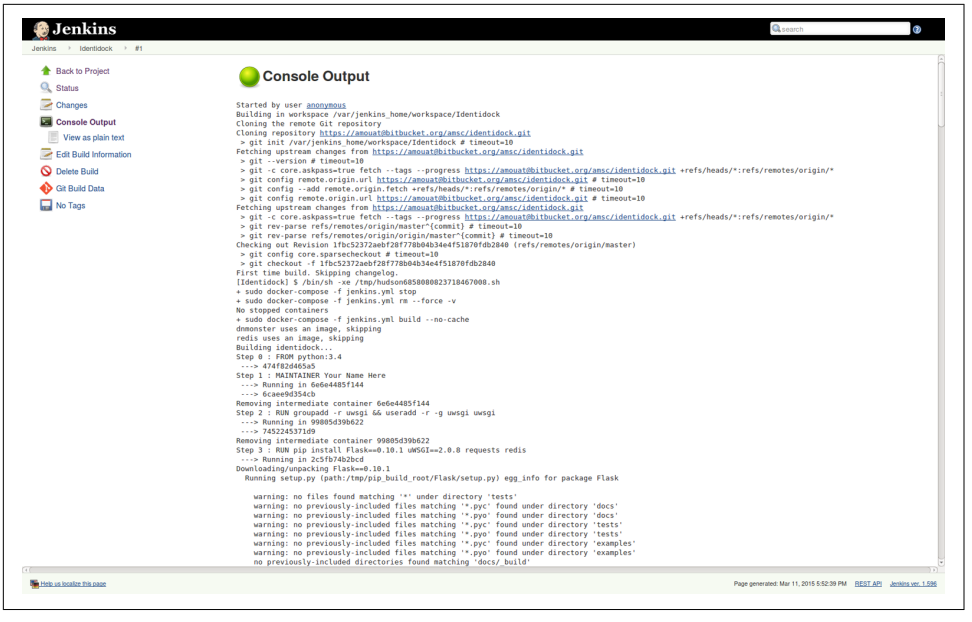


Figure 1-2. Successful Jenkins build

This is pretty good in as far as it goes; we've successfully got Docker running and managed to execute our unit tests, plus a simple "smoke test" on our application. However, if this was a real application, we would be looking to have a full suite of tests that ensure the application is functioning correctly and can handle a range of inputs, but this is all we need for our simple demo.

Triggering Builds

At the moment, builds are triggered manually by clicking Build Now. A major improvement to this is to have builds happen automatically on check-in to the GitHub project. To do this, enable the Poll SCM method in the identidock configuration and enter `H/5 * * * *` into the text box. This will cause Jenkins to check the repository every five minutes for any changes and schedule a build if any changes have occurred.

This is a simple solution and it works well enough, but it is somewhat wasteful and means builds are constantly lagging by up to five minutes. A better solution is to configure the repository to notify Jenkins of updates. This can be done using Web Hooks from either BitBucket or GitHub but requires that the Jenkins server is accessible on the public Internet.

Using the Docker Hub Image

At this point, some of you may be asking, "Why are we building an image at all?" If you followed the previous section, you should have an automated build set up on the Docker Hub that is firing on check-ins to the source repository. It is possible to take advantage of this by using the Webhooks feature on Docker Hub to automatically kick off a Jenkins build after a successful build on the Docker Hub repository. We can then pull, rather than build, the image in our script. This also requires the Jenkins server to be accessible on the public Internet.

This solution may be useful for small projects that are creating standalone Docker images, but larger projects will probably want the extra speed and security of controlling their own build.

Pushing the Image

Now that we've tested our identidock image, we need to push it through the rest of our pipeline somehow. The first step in this is to tag it and push it to a registry. From here it can be picked up by the next stage in the pipeline and pushed to staging or production.

Responsible Tagging

Tagging images correctly is essential for maintaining control and provenance over a container-based pipeline. Get it wrong and you will have images running in production that are difficult—if not impossible—to relate back to builds, making debugging and maintenance unnecessarily tricky. For any given image, we should be able to point to the exact Dockerfile and build context that was used to create it.[5]

Tags can be overwritten and changed at any time. Because of this, *it is up to you to create and enforce a reliable process for tagging and versioning images*.

For our example application, we will add two tags to the image: the Git hash of the repository and `newest`. This way the `newest` tag will always refer to the newest build that has passed our tests, and we can use the Git hash to recover the build files for any image. I've intentionally avoided using the `latest` tag due to the issues discussed in ???. Update the build script in Jenkins to:

```
#Default compose args
COMPOSE_ARGS=" -f jenkins.yml -p jenkins "
```

5 Note that this doesn't guarantee you will be able to re-create an identical container, as dependencies may have changed. See ??? for details on how to mitigate this.

```
#Make sure old containers are gone
sudo docker-compose $COMPOSE_ARGS stop
sudo docker-compose $COMPOSE_ARGS rm --force -v

#build the system
sudo docker-compose $COMPOSE_ARGS build --no-cache
sudo docker-compose $COMPOSE_ARGS up -d

#Run unit tests
sudo docker-compose $COMPOSE_ARGS run --no-deps --rm -e ENV=UNIT identidock
ERR=$?

#Run system test if unit tests passed
if [ $ERR -eq 0 ]; then
  IP=$(sudo docker inspect -f {{.NetworkSettings.IPAddress}} \
          jenkins_identidock_1)
  CODE=$(curl -sL -w "%{http_code}" $IP:9090/monster/bla -o /dev/null) || true
  if [ $CODE -eq 200 ]; then
    echo "Test passed - Tagging"
    HASH=$(git rev-parse --short HEAD) ❶
    sudo docker tag -f jenkins_identidock amouat/identidock:$HASH ❷
    sudo docker tag -f jenkins_identidock amouat/identidock:newest ❷
    echo "Pushing"
    sudo docker login -e joe@bloggs.com -u jbloggs -p jbloggs123❸
    sudo docker push amouat/identidock:$HASH ❹
    sudo docker push amouat/identidock:newest ❹
  else
    echo "Site returned " $CODE
    ERR=1
  fi
fi

#Pull down the system
sudo docker-compose $COMPOSE_ARGS stop
sudo docker-compose $COMPOSE_ARGS rm --force -v

return $ERR
```

❶ Get the short version of the Git hash.

❷ Add the tags.

❸ Log in to the registry.

❹ Push the images to the registry.

Note that you will need to rename the tag appropriately for the repository you wish to push to. For example, if your repository is running at `myhost:5000`, you will need to use `myhost:5000/identidock:newest`. Similarly, you will need to change the `docker login` credentials to match.

If you start a new build, you should find that the script now tags and pushes the images to the registry, ready for the next stage in the pipeline. This is great for our example application and is probably a good start for most projects. But as things get more complex, you are likely to want to use more tags and more descriptive names. The `git describe` command can be put to good use in generating more meaningful names based on tags.

Finding All Tags for an Image

Each tag for an image is stored separately. This means that in order to discover all the tags for an image, you need to filter the full image list based on the image ID. For example, to find all tags for the image with tag `amouat/identidock:newest`:

```
$ docker images --no-trunc | grep \
    $(docker inspect -f {{.Id}} amouat/identidock:newest)
amouat/identidock   51f6152  96c7b4c094c8f76ca82b6206f...
amouat/identidock   newest   96c7b4c094c8f76ca82b6206f...
jenkins_identidock  latest   96c7b4c094c8f76ca82b6206f...
```

And we can see that the same image is also tagged `51f6152`.

Remember that you will only see a tag if it exists in your image cache. For example, if I pull `debian:latest`, I don't get the `debian:7` tag even though (at the time of writing) it has the image ID. Similarly, if I have both the `debian:latest` and `debian:7` images, and I pull a new version of `debian:latest`, the `debian:7` tagged image will not be affected and will remain linked to the previous image ID.

Staging and Production

Once an image has been tested, tagged, and pushed to a registry, it needs to be passed on to the next stage in the pipeline, probably *staging* or *production*. This can be triggered in several ways, including by using Registry webhook notifications, or by using Jenkins to call the next step.

Image Sprawl

In a production system, you will need to address the problem of *image sprawl*. The Jenkins server should be periodically purged of images, and you will also need to control the number of images in the Registry, or it will rapidly fill with old and obsolete images. One solution is to remove all images older than a given date, possibly saving

them to a backup store if space allows.[6] Alternatively, you may want to look at more advanced tooling such as the CoreOS Enterprise Registry or Docker Trusted Registry, both of which include advanced features for managing repositories.

Test the Right Thing

It is important to make sure you test the same container image that is run in production. Don't build the image from a Dockerfile in testing and build again for production—you want to be certain that you are running the same thing you tested and no differences have crept in. For this reason, it is essential to run some form of registry or store for your images that can be shared between testing, staging, and production.

Using Docker to Provision Jenkins Slaves

As your build requirements grow, you will require more and more resources to run your tests. Jenkins uses the concept of "build slaves," which essentially form a task farm Jenkins can use to outsource builds.

If you would like to use Docker to dynamically provision these slaves, take a look at the Docker plugin for Jenkins.

Backing Up Jenkins

Because we used a data container for our Jenkins service, backing up Jenkins should be as simple as:

```
$ docker run --volumes-from jenkins-data -v $PWD:/backup \
          debian tar -zcvf /backup/jenkins-data.tar.gz /var/jenkins_home
```

This should result in the file *jenkins-data.tar.gz* appearing in your *$PWD/backup* directory. You may want to stop or pause the Jenkins container prior to running this command. You can then run something like the following command to create a new data container and extract the backup into it:

```
$ docker run --name jenkins-data2 identijenk echo "New Jenkins Data Container"
$ docker run --volumes-from jenkins-data2 -v $PWD:/backup \
            debian tar -xzvf /backup/backup.tar
```

Unfortunately, this approach does require you to be aware of the mount points of your container. This can be automated by inspecting the container, so you can also

6 At the time of writing, this is easier said than done with locally hosted Docker registries, as the remove function hasn't been implemented. There are several issues to be overcome, which are described in detail on the distribution roadmap.

use tools like docker-backup to do this for you, and I expect to see more support for workflows like this in future versions of Docker.

Hosted CI Solutions

There are also numerous hosted solutions for CI, from companies that will maintain a Jenkins installation in the cloud for you, to more specialized solutions such as Travis, Wercker, CircleCI, and Drone. Most of these solutions seem to be targeted at running unit tests for predefined language stacks rather than running tests against systems of containers. There does seem to be some movement in this area, and I expect to see offerings aimed at testing Docker containers soon.

Testing and Microservices

If you're using Docker, there's a good chance you've also adopted a microservice architecture. When testing a microservice architecture, you will find that there are more levels of testing that are possible, and it is up to you to decide how and what to test. A basic framework might consist of:

Unit tests
: Each service[7] should have a comprehensive set of unit tests associated with it. Unit tests should only test small, isolated pieces of functionality. You may use test doubles to replace dependencies on other services. Due to the number of tests, it is important that they run as quickly as possible to encourage frequent testing and avoid developers waiting on results. Unit tests should make up the largest proportion of tests in your system.

Component tests
: These can be on the level of testing the external interface of individual services, or on the level of subsystem testing of groups of services. In both cases, you are likely to find you have dependencies on other services, which you may need to replace with test doubles as described earlier. You may also find it useful to expose metrics and logging via your service's API when testing, but make sure this is kept in a separate namespace (e.g., use a different URL prefix) to your functional API.

End-to-end tests
: Tests that ensure the entire system is working. Because these are quite expensive to run (in terms of both resources and time), there should only be a few of these —you really don't want a situation where it takes hours to run the tests, seriously

7 Normally, there will be one container per service, or multiple containers per service if more resources are needed.

delaying deployments and fixes (consider *scheduled runs*, which we describe shortly). Some parts of the system may be impossible or prohibitively expensive to run in testing and may still need to be replaced with test doubles (launching nuclear missiles in testing is probably a bad idea). Our identidock test falls under end-to-end testing; the test runs the full system from end to end with no use of test doubles.

In addition, you may want to consider:

Consumer contract tests
: These tests, which are also called consumer-driven contracts, are written by the *consumer* of a service and primarily define the expected input and output data. They can also cover side effects (changing state) and performance expectations. There should be a separate contract for each consumer of the service. The primary benefit of such tests is that it allows the developers of a service to know when they risk breaking compatibility with consumers; if a contract test fails, they know they need to either change their service, or work with the developers of the consumer to change the contract.

Integration tests
: These are tests to check that the communication channels between each component are working correctly. This sort of testing becomes important in a microservice architecture where the amount of plumbing and coordination between components is an order of magnitude greater than monolithic architectures. However, you are likely to find that most of your communication channels are covered by your component and end-to-end testing.

Scheduled runs
: Because it's important to keep the CI build fast, there often isn't enough time to run extensive tests, such as testing against unusual configurations or different platforms. Instead, these tests can be scheduled to run overnight when there is spare capacity.

Many of these tests can be classified as *preregistry* and *postregistry*, depending on whether they occur prior to adding the image to the registry. For example, unit testing is preregistry: no image should be pushed to the registry if it fails a unit test. The same goes for some consumer contract tests and some component tests. On the other hand, an image will have already been pushed to a registry before it can be end-to-end tested. If a postregistry test fails, there is a question about what to do next. While any new images should not be pushed to production (or should be rolled back if they have already been deployed), the fault may actually be due to other, older images or the interaction between new images. These sort of failures may require a greater level of investigation and thought to handle correctly.

Testing in Production

Finally, you may want to think about testing in production. Don't worry, this isn't as crazy as it sounds. In particular, it can make a lot of sense when dealing with a large number of users with widely different environments and configurations that are hard to test for.

One common approach is sometimes called *blue/green deployment*. Say we want to update an existing production service (let's call it the "blue" version) to new a version (let's call it the "green" version). Rather than just replace the blue version with the green version, we can run them in tandem for a given time period. Once the green version is up and running, we flip the switch to start routing traffic to it. We then monitor the system for any unexpected changes in behavior, such as increased error rates or latency. If we're not happy with the new version, all we have to do is flip the switch back to return the blue version to production. Once we're satisfied things are working correctly, we can turn off the blue version.

Other methods follow a similar principle—both the old and new versions should run in tandem. In *A/B*, or *multivariate testing*, two (or more) versions of a service are run together for a test period, with users randomly split between two. Certain statistics are monitored, and based on the results at the end of testing, one of the versions is kept. In *ramped deployment*, the new version of a service is only made available to a small subset of users. If these users find no problems, the new version will be progressively made available to more and more users. In *shadowing*, both versions of the service are run for all requests, but only the results from the old, stable version are used. By comparing the results from the old version and the proposed new version, it is possible to ensure the new version has identical behavior to the old version (or differs in an expected and positive way). Shadowing is particularly useful when testing new versions that do not have functional changes such as performance improvements.

Conclusion

The key idea to take away is that containers fit naturally into a continuous integration/delivery workflow. There are a few things to bear in mind—primarily that you must push the same image through the pipeline rather than rebuilding at separate stages—but you should be able to adapt existing CI tooling to containers without too many problems, and the future is likely to bring further specialized tooling in this area.

If you're embracing a large microservice architecture, it's worth taking more time to think about how you are going to do testing and researching some of the techniques outlined in this chapter.

CHAPTER 2
Deploying Containers

Now it's time to start getting to the business end of things and thinking about how to actually run Docker in production. At the time of writing, everybody is talking about Docker, and many are experimenting with Docker, but comparatively few run Docker in production. While detractors sometimes point to this as a failing of Docker, they seem to miss a couple of key points. Given the relative youth of Docker, it is very encouraging that so many people are using it in production (including Spotify, Yelp, and Baidu) and that those who only use it in development and testing are still gaining many advantages.

That being said, it is perfectly possible and reasonable to use containers in production today. Larger projects and organizations may want to start small and build up over time, but it is already a feasible and straightforward solution for the majority of projects.

As things currently stand, the most common way of deploying containers is by first provisioning VMs and then starting containers on the VMs. This isn't an ideal solution—it creates a lot of overhead, slows down scaling, and forces users to provision on a multicontainer granularity. The main reason for running containers inside VMs is simply security. It's essential that customers cannot access other customers' data or network traffic, and containers by themselves only provide weak guarantees of isolation at the moment. Further, if one container monopolizes kernel resources, or causes a panic, it will affect all containers running on the same host. Even most of the specialist solutions—Google Container Engine (GKE) and the Amazon EC2 Container Service (ECS)—still use VMs internally. There are currently two exceptions to this rule, Giant Swarm and Triton from Joyent, both of which are discussed later.

Throughout this chapter, we will show how our simple web application can be deployed on a range of clouds, as well as specialized Docker hosting services. We will

also look at some of the issues and techniques for running containers in production, both in the cloud and using on-premise resources.

The code for this chapter is available at this book's GitHub. We won't build on the previous Python code anymore but will continue to use the images we have created. You can choose to use your own version of the identidock image or simply use the `amouat/identidock` repository.

You can check out the code for the start of the chapter using the `v0` tag:

```
$ git clone -b v0 \
  https://github.com/using-docker/deploying-containers/
...
```

Later tags represent the progression of the code throughout the chapter.

Alternatively, you can download the code for any tag from the Releases page on the GitHub project.

Provisioning Resources with Docker Machine

The fastest and simplest way to provision new resources and run containers on them is via *Docker Machine*, which can create servers, install Docker on them, and configure the local Docker client to access them. Docker Machine comes with drivers for most of the major cloud providers (including AWS, Google Compute Engine, Microsoft Azure, and Digital Ocean) as well as VMWare and VirtualBox.

Beta Software Alert!

At the time of writing, Docker Machine is in beta (I tested Docker Machine version 0.4.1). This means you are likely to encounter bugs and missing functionality, but it should still be usable and reasonably stable. Unfortunately, it also means the commands and syntax are likely to change slightly from what you see here. For this reason, I don't recommend using Machine in production yet, although it is very useful for testing and experimentation.

(And yes, this warning is true for nearly everything in this book—it just felt like a good time to point that out again.)

Let's have a look at how to use Machine to get identidock up and running in the cloud. To begin with, you'll need to install Machine on your local computer. If you installed Docker via Docker Toolbox, it should already be available. If not, you can download a binary from GitHub, which can then be placed on your path (e.g., */usr/*

local/bin/docker-machine). Once you've done this, you should be able to start running commands:

```
$ docker-machine ls
NAME       ACTIVE   DRIVER         STATE     URL                         SWARM
default             virtualbox     Running   tcp://192.168.99.100:2376
```

You may or may not get any output here, depending on what hosts Machine has detected. In my case, it picked up my local `boot2docker` VM. What we want to do next is add a host somewhere in the cloud. I'll walk through this using Digital Ocean, but AWS and the other cloud providers should be very similar. In order to follow along, you'll need to have registered online and generated a personal access token; to do this, open the "Applications & API" page. You will be charged for resource usage, so make sure to remove the machine when you're finished with it:

```
$ docker-machine create --driver digitalocean \
     --digitalocean-access-token 4820... \
     identihost-do
Creating SSH key...
Creating Digital Ocean droplet...
To see how to connect Docker to this machine, run: docker-machine env identi...
```

We've now created a Docker host on Digital Ocean. The next thing to do is to point our local client at it, using the command given in the output:

```
$ docker-machine env identihost-do
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://104.236.32.178:2376"
export DOCKER_CERT_PATH="/Users/amouat/.docker/machine/machines/identihost-do"
export DOCKER_MACHINE_NAME="identihost-do"
# Run this command to configure your shell:
# eval "$(docker-machine env identihost-do)"
$ eval "$(docker-machine env identihost-do)"
$ docker info
Containers: 0
Images: 0
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 0
 Dirperm1 Supported: false
Execution Driver: native-0.2
Logging Driver: json-file
Kernel Version: 3.13.0-57-generic
Operating System: Ubuntu 14.04.3 LTS
CPUs: 1
Total Memory: 490 MiB
Name: identihost-do
ID: PLDY:REFM:PU5B:PRJK:L4QD:TRKG:RWL6:5T6W:AVA3:2FXF:ESRC:6DCT
Username: amouat
Registry: https://index.docker.io/v1/
```

```
WARNING: No swap limit support
Labels:
 provider=digitalocean
```

And we can see that we're connected to an Ubuntu host running on Digital Ocean. If we now run `docker run hello-world`, it will execute on the remote server.

Now to run identidock, you can use the previous *docker-compose.yml* from the end of ???, or use the following *docker-compose.yml,* which uses an image from the Docker Hub for identidock:

```
identidock:
  image: amouat/identidock:1.0
  ports:
   - "5000:5000"
   - "9000:9000"
  environment:
    ENV: DEV
  links:
    - dnmonster
    - redis
dnmonster:
  image: amouat/dnmonster:1.0
redis:
  image: redis:3
```

Note that if the Compose file includes a `build` instruction, this build will occur on the remote server. Any volume mounts will need to be removed, as they will refer to the disk on the remote server, not your local computer.

Run Compose normally:

```
$ docker-compose up -d ❶
...
Creating identidock_identidock_1...
$ curl $(docker-machine ip identihost-do):5000 ❷
<html><head><title>Hello...
```

❶ This will take a while as it will need to first download and build the required images.

❷ We can use the `docker-machine ip` command to find where our Docker host is running.

So now identidock is running in the cloud and accessible to anyone.[1] It's fantastic that we were able to get something up and running so quickly, but there are a few things that aren't quite right. Notably, the application is running the development Python

1 Some providers, including AWS, may require you to open port 5000 in the firewall first.

web server on port 5000. We should change to use the production version, but it would also be nice to put a reverse proxy or load balancer in front of the application, which would allow us to make changes to the identidock infrastructure without changing the external IP address. Nginx has support for load balancing, so it also makes it simple to bring up several identidock instances and share traffic between them.

Smoke Testing Identidock

Throughout this book, we `curl` the identidock service to make sure it works. However, simply grabbing the front page isn't a great test; it only proves that the identidock container is up and running. A better test is to retrieve an identicon, which proves both the identidock and dnmonster containers are active and communicating. You can do this with a test such as:

```
$ curl localhost:5000/monster/gordon | head -c 4
PNG
```

Here we've used the Unix head utility to grab the first four characters of the image, which avoids dumping binary data to our terminal.

Using a Proxy

Let's start by creating a reverse proxy using nginx that our identidock service can sit behind. Create a new folder *identiproxy* for this and create the following Dockerfile:

```
FROM nginx:1.7

COPY default.conf /etc/nginx/conf.d/default.conf
```

Also create a file *default.conf* with the following contents:

```
server {
    listen       80;
    server_name  45.55.251.164; ❶

    location / {

      proxy_pass  http://identidock:9090; ❷
      proxy_next_upstream error timeout invalid_header http_500 http_502
                          http_503 http_504;
      proxy_redirect off;
      proxy_buffering off;
      proxy_set_header        Host            45.55.251.164; ❶
      proxy_set_header        X-Real-IP       $remote_addr;
      proxy_set_header        X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

❶ Replace this with the IP address of your Docker host or a domain name that points to it.

❷ Redirect all traffic to the identidock container. We'll use links to make this work.

If you still have Machine running and pointed to the cloud server, we can now build our image on the remote server:

```
$ docker build --no-cache -t identiproxy:0.1 .
Sending build context to Docker daemon 3.072 kB
Sending build context to Docker daemon
Step 0 : FROM nginx:1.7
 ---> 637d3b2f5fb5
Step 1 : COPY default.conf /etc/nginx/conf.d/default.conf
 ---> 2e82d9a1f506
Removing intermediate container 5383f47e3d1e
Successfully built 2e82d9a1f506
```

It's easy to forget that we're speaking to a remote Docker engine, but the image now exists on the remote server, not your local development machine.

Now we can return to the `identidock` folder and create a new Compose configuration file to test it out. Create a *prod.yml* with the following contents:

```
proxy:
  image: identiproxy:0.1 ❶
  links:
    - identidock
  ports:
   - "80:80"
identidock:
  image: amouat/identidock:1.0
  links:
   - dnmonster
   - redis
  environment:
    ENV: PROD ❷
dnmonster:
  image: amouat/dnmonster:1.0 ❶
redis:
  image: redis:3 ❶
```

❶ Note that I've used tags for the all the images. In production, you should be careful about the versions of containers you are running. Using `latest` is particularly bad, as it can be difficult or impossible to figure out what version of the application the container is running.

❷ Note that we're no longer exposing ports on the identidock container (only the proxy container needs to do that) and we've updated the environment variable to start the production version of the web server.

Using extends in Compose

For more verbose YAML files, you can use the `extends` keyword to share config details between environments. For example, we could define a file *common.yml* with the following contents:

```
identidock:
  image: amouat/identidock:1.0
  environment:
    ENV: DEV
dnmonster:
  image: amouat/dnmonster:1.0
redis:
  image: redis:3
```

We can then rewrite our *prod.yml* file as:

```
proxy:
  image: identiproxy:0.1
  links:
    - identidock
  ports:
   - "80:80"
identidock:
  extends:
    file: common.yml
    service: identidock
  environment:
    ENV: PROD
dnmonster:
  extends:
    file: common.yml
    service: dnmonster
redis:
  extends:
    file: common.yml
    service: redis
```

Where the `extends` keyword pulls in the appropriate config from the common file. Settings in the *prod.yml* will override settings in the *common.yml*. Values in `links` and `volumes-from` are *not* inherited to avoid unexpected breakages. Because of this, in our case, using `extends` actually results in a more verbose *prod.yml* file, although it would still have the important advantage of automatically inheriting any changes made to the base file. The main reason I've avoided using `extends` in the book is simply to keep the examples standalone.

Stop the old version and start the new:

```
$ docker-compose stop
Stopping identidock_identidock_1... done
Stopping identidock_redis_1... done
Stopping identidock_dnmonster_1... done
Starting identidock_dnmonster_1...
Starting identidock_redis_1...
Recreating identidock_identidock_1...
Creating identidock_proxy_1...
```

Now let's test it out; it should now answer on the default port 80 rather than port 9090:

```
$ curl $(docker-machine ip identihost-do)
<html><head><title>Hello...
```

Excellent! Now our container is sitting behind a proxy, which makes it possible to do things like load balance over a group of identidock instances or move identidock to a new host without breaking the IP address (as long as the proxy remains on the old host and is updated with the new value). In addition, security has increased because the application container can only be accessed via the proxy and is no longer exposing ports to the Internet at large.

We can do a bit better than this though. It's really annoying that the IP of the host and the container name are hardcoded into the proxy image; if we want to use a different name than "identidock" or use identiproxy for another service, we need to build a new image or overwrite the config with a volume. What we want is to have these parameters set as environment variables. We can't use environment variables directly in nginx, but we can write a script that will generate the config at runtime, then start nginx. We need to go back to our *identiproxy* folder and update the *default.conf* file so that we have placeholders instead of the hardcoded variables:

```
server {
    listen       80;
    server_name {{NGINX_HOST}};

    location / {

      proxy_pass  {{NGINX_PROXY}};
      proxy_next_upstream error timeout invalid_header http_500 http_502
                          http_503 http_504;
      proxy_redirect off;
      proxy_buffering off;
      proxy_set_header        Host            {{NGINX_HOST}};
      proxy_set_header        X-Real-IP       $remote_addr;
      proxy_set_header        X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

and create the following *entrypoint.sh*, which will do our replacement:

```
#!/bin/bash
set -e

sed -i "s|{{NGINX_HOST}}|$NGINX_HOST|;s|{{NGINX_PROXY}}|$NGINX_PROXY|" \
  /etc/nginx/conf.d/default.conf ❶
cat /etc/nginx/conf.d/default.conf ❷
exec "$@" ❸
```

❶ We're using the `sed` utility to do our replacement. This is a bit hacky, but it will be fine for our purposes. Note we've used bars (`|`) instead of slashes (`/`) to avoid confusion with slashes in URLs.

❷ Prints the final template into the logs, which is handy for debugging.

❸ Executes whatever `CMD` has been passed. By default, the Nginx container defines a `CMD` instruction that starts nginx in the foreground, but we could define a different `CMD` at runtime that runs different commands or starts a shell if required.

Now we just need to update our Dockerfile to include our new script:

```
FROM nginx:1.7

COPY default.conf /etc/nginx/conf.d/default.conf
COPY entrypoint.sh /entrypoint.sh

ENTRYPOINT ["/entrypoint.sh"]
CMD ["nginx", "-g", "daemon off;"] ❶
```

❶ This command starts our proxy and will be passed as an argument to our *entrypoint.sh* script if no command is specified in `docker run`.

Make it executable and rebuild. This time we'll just call it proxy, as we've abstracted out the identidock details:

```
$ chmod +x entrypoint.sh
$ docker build -t proxy:1.0 .
...
```

To use our new image, go back to the *identidock* folder and update our *prod.yml* to use the new image:

```
proxy:
  image: proxy:1.0
  links:
    - identidock
  ports:
   - "80:80"
  environment:
    - NGINX_HOST=45.55.251.164 ❶
    - NGINX_PROXY=http://identidock:9090
identidock:
```

```
    image: amouat/identidock:1.0
    links:
     - dnmonster
     - redis
    environment:
      ENV: PROD
dnmonster:
    image: amouat/dnmonster:1.0
redis:
    image: redis:3
```

❶ Set this variable to the IP or name of your host.

So now if you bring down the old version and restart the app, we'll be using the new, generic image. For our simple web app, this is all we need, but due to the use of Docker links, we are currently stuck with a single-host configuration—we can't move to a multihost architecture (which would be necessary for fault tolerance and scaling) without using more advanced networking and service discovery features that we will see in Chapters 11 and 12.

Once you've finished with the application, you can stop it as follows:

```
$ docker-compose -f prod.yml stop
...
$ docker-compose -f prod.yml rm
...
```

When you're ready to shut down the cloud resource, just do this:

```
$ docker-machine stop identihost-do
$ docker-machine rm identihost-do
```

It's worth making sure the resources have been correctly freed in the cloud provider's web interface.

Next, let's take a look at some of the alternatives to using Compose.

Setting the COMPOSE_FILE Variable

Rather than explicitly specifying `-f prod.yml` to compose each time, you can also set the `COMPOSE_FILE` environment variable. For example:

```
$ export COMPOSE_FILE=prod.yml
$ docker-compose up -d
...
```

This will use the file *prod.yml* rather than the default *docker-compose.yml*.

Supercharged Config File Generation

The technique of using templates to build configuration files for Docker containers is fairly common when Dockerizing applications, especially when they don't natively support environment variables. When moving beyond simple examples like the one here, you will want to use a proper template processor, such as Jinga2 or Go templates, in order to avoid strange errors due to regexp clashes.

The problem is common enough that there is now a utility to help automate this process: Jason Wilder's dockerize. Dockerize will generate configuration files from a template file and environment variables, then call the normal application. In this way, it can be used to wrap application startup scripts called from a `CMD` or `ENTRYPOINT` instruction.

However, Jason took this one step further with docker-gen, which can use values from container metadata (such as IP address) as well as environment variables. It can also run continuously, responding to Docker events such as new container creation to update configuration files appropriately. A great example of this is his nginx-proxy container, which will automatically add containers with the `VIRTUAL_HOST` environment variable to a load-balanced group.

Execution Options

Now that we've got a production ready system,[2] how should we go about starting the system on the server?[3] So far we've looked at Compose and Machine, but because both these projects are relatively new and in rapid development, it's wise to be wary of using them in production for anything except small side projects (and, at the time of writing, there are warnings to this extent on the Docker website). Both the projects

2 Well, not really. It's important to think about how to secure your application before inviting Joe Public to take a look. See ??? for more information.

3 Oh, and you'll want to think about how to handle monitoring and logging, too. Don't forget those. See ???.

are quickly maturing and developing production features; to get an idea of where they are going, you can find roadmap documents in the GitHub repositories, which are great for feeling out how close the projects are to production-ready.

So, if Compose isn't an option, what is? Let's take a look at some of the other possibilities. All of the following code assumes that images are available on Docker Hub, rather than building them on the server. If you want to follow along, either push your own images to a registry or use my images from the Docker Hub (`amouat/identidock:1.0`, `amouat/dnmonster:1.0` and `amouat/proxy:1.0`).

Shell Scripts

The easiest answer to running without Compose is just to write a short shell script that executes Docker commands to bring up the containers. This will work well enough for a lot of simple use cases, and if you add in some monitoring, you can make sure you know about it if anything goes wrong that requires your attention. However, in the long run, it is far from perfect; you will likely end up maintaining a messy and unstructured script that evolves over time to grow features of other solutions.

We can ensure containers that exit prematurely are automatically restarted by using the `--restart` argument to `docker run`. The argument specifies the restart policy, which can be `no`, `on-failure`, or `always`. The default is `no`, which will never automatically restart containers. The `on-failure` policy will only restart containers that exit with a nonzero exit code and can also specify a maximum number of retries (e.g., `docker run --restart on-failure:5` will attempt to restart the container five times before giving up).

The following script (named *deploy.sh*) will get our identidock service up and running:

```
#!/bin/bash
set -e

echo "Starting identidock system"

docker run -d --restart=always --name redis redis:3
docker run -d --restart=always --name dnmonster amouat/dnmonster:1.0
docker run -d --restart=always \
  --link dnmonster:dnmonster \
  --link redis:redis \
  -e ENV=PROD \
  --name identidock amouat/identidock:1.0
docker run -d --restart=always \
  --name proxy \
  --link identidock:identidock \
  -p 80:80 \
  -e NGINX_HOST=45.55.251.164 \
```

```
    -e NGINX_PROXY=http://identidock:9090 \
    amouat/proxy:1.0

echo "Started"
```

Note that we're really just converting our *docker-compose.yml* file into the equivalent shell commands. But unlike Compose, there is no logic for cleaning up after failures, or to check for already running containers.

In the case of Digital Ocean, I can now use the following `ssh` and `scp` commands to start identidock using the shell script:

```
$ docker-machine scp deploy.sh identihost-do:~/deploy.sh
deploy.sh                                     100%  575     0.6KB/s   00:00
$ docker-machine ssh identihost-do
...
$ chmod +x deploy.sh
$ ./deploy.sh
Starting identidock system
3b390441b16eaece94df7e0e07d1edcb4c11ce7232108849d691d153330c6dfb
57459e4c0c2a75d2fbcef978aca9344d445693d2ad6d9efe70fe87bf5721a8f4
5da04a34302b400ec08e9a1d59c3baeec14e3e65473533c165203c189ad58364
d1839d8de1952fca5c41e0825ebb27384f35114574c20dd57f8ce718ed67e3f5
Started
```

We could also have just run these commands directly in the shell. The main reason to prefer the script is for documentation and portability reasons—if I want to start identidock on a new host, I can easily find the instructions to bring up an identical version of the application.

When we need to update images or make changes, we can either use Machine to connect our local client to the remote Docker server or log directly into the remote server and use the client there. To perform a zero-downtime update of a container, you will need to have a load balancer or reverse proxy in front of the container and do something like:

1. Bring a up a new container with the updated image (it's best to avoid trying to update images in place).
2. Point the load balancer at the new image, for some or all of the traffic.
3. Test the new container is working.
4. Turn off the old container.

Also, refer to "Testing in Production", which describes various techniques for deploying updates without breaking services.

Breaking Links on Restart

Older versions of Docker had problems with links breaking when containers restarted. If you see similar issues, make sure you are running an up-to-date version of Docker. At the time of writing, I am using Docker version 1.8, which works correctly; any changes to a container's IP address are automatically propagated to linked containers. Also note that only */etc/hosts* is updated, and environment variables are *not* updated on changes to linked containers.

In the rest of this section, we'll look at how you can control the initialization and deployment of containers using existing technology you may already be familiar with. In ???, we will look at some of the newer, Docker-specific tooling that also addresses this issue.

Using a Process Manager (or systemd to Rule Them All)

Instead of relying on a shell script and the Docker restart functionality, you can use a process manager or init system such as systemd or upstart to bring up your containers. This can be particularly useful if you have host services that don't run in a container, but are dependent on one or more containers. If you want to do this, be aware that there are some issues:

- You will need to make sure you don't use Docker's automatic container restarting functionality-that is, don't use `--restart=always` in your `docker run` commands.
- Normally, your process manager will end up monitoring the `docker client` process, rather than the processes inside the container. This works most of the time, but if the network connection drops or something else goes wrong, the Docker client will exit but leave the container running, which can cause problems. Instead, it would be much better if the process manager monitored the main process inside the container. This situation may change in the future, but until then, be aware of the systemd-docker project, which works around this by taking control of the container's cgroup. (For more information on the problem, see this GitHub issue.)

To give you an example of how to manage containers with systemd, the following service files can be used to start our identidock service on a systemd host. For this example, I've used CentOS 7, but other systemd-based distributions should be very similar. I haven't included an upstart example, as all major distributions seem to be moving to systemd. All of the files should be stored under */etc/systemd/system/*.

Let's start by looking at the service file for the Redis container, *identidock.redis.service*, which isn't dependent on any other containers:

```
[Unit]
Description=Redis Container for Identidock
After=docker.service
Requires=docker.service ❶

[Service]
TimeoutStartSec=0 ❷
Restart=always
ExecStartPre=-/usr/bin/docker stop redis ❸
ExecStartPre=-/usr/bin/docker rm redis
ExecStartPre=/usr/bin/docker pull redis ❹
ExecStart=/usr/bin/docker run --rm --name redis redis

[Install]
WantedBy=multi-user.target
```

❶ We need to make sure Docker is running before starting the container.

❷ As the Docker commands may take some time to run, it's easiest to turn the timeout off.

❸ Before starting the container, we first remove any old container with the same name, which means we will destroy the Redis cache on restart. But in the case of identidock, it's not an issue. The use of - at the start of the command tells systemd not to abort if the command returns a nonzero return code.

❹ Doing a pull ensures we are running the newest version.

The identidock service *identidock.identidock.service* is similar but requires other services:

```
[Unit]
Description=identidock Container for Identidock
After=docker.service
Requires=docker.service
After=identidock.redis.service ❶
Requires=identidock.redis.service
After=identidock.dnmonster.service
Requires=identidock.dnmonster.service

[Service]
TimeoutStartSec=0
Restart=always
ExecStartPre=-/usr/bin/docker stop identidock
ExecStartPre=-/usr/bin/docker rm identidock
ExecStartPre=/usr/bin/docker pull amouat/identidock
ExecStart=/usr/bin/docker run --name identidock \
  --link dnmonster:dnmonster \
  --link redis:redis \
  -e ENV=PROD \
```

```
  amouat/identidock

[Install]
WantedBy=multi-user.target
```

❶ In addition to Docker, we need to declare that we are dependent on the other containers used in identidock-in this case, the Redis and dnmonster containers. Both `After` and `Requires` are needed to avoided race conditions.

The proxy service (called *identidock.proxy.service*) looks like:

```
[Unit]
Description=Proxy Container for Identidock
After=docker.service
Requires=docker.service
Requires=identidock.identidock.service

[Service]
TimeoutStartSec=0
Restart=always
ExecStartPre=-/usr/bin/docker stop proxy
ExecStartPre=-/usr/bin/docker rm proxy
ExecStartPre=/usr/bin/docker pull amouat/proxy
ExecStart=/usr/bin/docker run --name proxy \
  --link identidock:identidock \
  -p 80:80 \
  -e NGINX_HOST=0.0.0.0 \
  -e NGINX_PROXY=http://identidock:9090 \
  amouat/proxy

[Install]
WantedBy=multi-user.target
```

And finally, the dnmonster service (called *identidock.dnmonster.service*) looks like:

```
[Unit]
Description=dnmonster Container for Identidock
After=docker.service
Requires=docker.service

[Service]
TimeoutStartSec=0
Restart=always
ExecStartPre=-/usr/bin/docker stop dnmonster
ExecStartPre=-/usr/bin/docker rm dnmonster
ExecStartPre=/usr/bin/docker pull amouat/dnmonster
ExecStart=/usr/bin/docker run --name dnmonster amouat/dnmonster

[Install]
WantedBy=multi-user.target
```

We can now start identidock with `systemctl start identidock.*`. The major difference between using this system and the Docker restart functionality is that restart-

ing a stopped container will kick off a chain of restarts in systemd; if the Redis container goes down, both the identidock and proxy containers will also be restarted. This isn't the case in Docker, as it knows how to update links without restarting the container completely.

Despite the previously mentioned issues, it is worth noting that both CoreOS and the Giant Swarm PaaS use systemd to control containers. At the moment, it seems fair to say that there is unresolved tension between Docker and systemd, both of which want to be in charge of the lifecycle of services running on the host.

Using a Configuration Management Tool

If your organization is responsible for more than a handful of hosts, chances are that you're using some sort of configuration management (CM) tool (and if not, you probably should be). All projects need to consider how they are going to ensure the operating system on the Docker host is up to date, especially with regard to security patches. In turn, you also want to make sure the Docker images you are running are up to date and you aren't mixing multiple versions of your software. CM solutions such as Puppet, Chef, Ansible, and Salt are designed to help manage these issues.

There are two main ways we can use CM tools with containers:

- We can treat our containers as VMs and use CM software to manage and update the software inside them.
- We can use the CM software to manage the Docker host and ensure containers are running the correct version of images, but view the containers themselves as black boxes that can be replaced, but not modified.

The first approach is feasible, but is not the Docker way. You'll be working against Dockerfiles and the small-container-with-a-single-process philosophy that Docker is built around. In the rest of this section, we'll focus on the second alternative, which is much more in line with the Docker philosophy and microservices approach.

In this approach, the containers themselves are similar to *golden images* in VM parlance and shouldn't be modified once running. When you need to update them, you replace the entire container with one running the new image rather than try to change anything running inside the image. This has the major advantage that you know exactly what is running in your container by just looking at the image tag (assuming you are using a proper tagging system and aren't reusing tags).

Let's look at an example of how you can do this.

Ansible

For this example, we'll use Ansible—it's popular, easy to get started with, and open source. This isn't to say it is better or worse than other tools!

Unlike many other configuration management solutions, Ansible doesn't require the installation of agents on hosts. Instead, it mainly relies on SSH to configure hosts.

Ansible has a Docker module, which has functionality for both building and orchestrating containers. It is possible to use Ansible inside Dockerfiles to install and configure software, but here we will just consider using Ansible to set up a VM with our identidock image. We're only running on a single host so we're not really making the most the Ansible here, but it does demonstrate how well Ansible and Docker can be used together.

Rather than install the Ansible client, we can just use an Ansible client image from the Hub. There isn't an official image available, but the `generik/ansible` image will work for testing.

Start by creating a *hosts* file that contains a list of all the servers we want Ansible to manage (make sure to include the IP address of your remote host or VM here):

```
$ cat hosts
[identidock]
46.101.162.242
```

Now we need to create the "playbook" for installing identidock. Create a file *identidock.yml* with the following contents, replacing the image names if you want to use your own:

```
---
- hosts: identidock
  sudo: yes
  tasks:
  - name: easy_install
    apt: pkg=python-setuptools
  - name: pip
    easy_install: name=pip
  - name: docker-py
    pip: name=docker-py
  - name: redis container
    docker:
      name: redis
      image: redis:3
      pull: always
      state: reloaded
      restart_policy: always
  - name: dnmonster container
    docker:
      name: dnmonster
      image: amouat/dnmonster:1.0
```

```
      pull: always
      state: reloaded
      restart_policy: always
  - name: identidock container
    docker:
      name: identidock
      image: amouat/identidock:1.0
      pull: always
      state: reloaded
      links:
        - "dnmonster:dnmonster"
        - "redis:redis"
      env:
        ENV: PROD
      restart_policy: always
  - name: proxy container
    docker:
      name: proxy
      image: amouat/proxy:1.0
      pull: always
      state: reloaded
      links:
        - "identidock:identidock"
      ports:
        - "80:80"
      env:
        NGINX_HOST: www.identidock.com
        NGINX_PROXY: http://identidock:9090
      restart_policy: always
```

Most of the configuration is very similar to Docker Compose, but note that:

- We have to install `docker-py` on the host in order to use the Ansible Docker module. This in turn requires us to install some Python dependencies.
- The `pull` variable determines when Docker images are checked for updates. Setting it to `always` ensures Ansible will check for a new version of the image each time the task executes.
- The `state` variable determines what state the container should be in. Setting it to `reloaded` will restart the container whenever a change is made to the configuration.

There are many more configuration options available, but this config will get us something very similar to the other setups described in this chapter.

All that's left to do is to run the playbook:

```
$ docker run -it \
    -v ${HOME}/.ssh:/root/.ssh:ro \ ❶
    -v $PWD/identidock.yml:/ansible/identidock.yml \
    -v $PWD/hosts:/etc/ansible/hosts \
```

```
    --rm=true generik/ansible ansible-playbook identidock.yml

PLAY [identidock] ************************************************************

GATHERING FACTS ***************************************************************
The authenticity of host '46.101.41.99 (46.101.41.99)' can't be established.
ECDSA key fingerprint is SHA256:R0LfM7Kf3OgRmQmgxINko7SonsGAC0VJb27LTotGEds.
Are you sure you want to continue connecting (yes/no)? yes
Enter passphrase for key '/root/.ssh/id_rsa':
ok: [46.101.41.99]

TASK: [easy_install] **********************************************************
changed: [46.101.41.99]

TASK: [pip] *******************************************************************
changed: [46.101.41.99]

TASK: [docker-py] *************************************************************
changed: [46.101.41.99]

TASK: [redis container] *******************************************************
changed: [46.101.41.99]

TASK: [dnmonster container] ***************************************************
changed: [46.101.41.99]

TASK: [identidock container] **************************************************
changed: [46.101.41.99]

TASK: [proxy container] *******************************************************
changed: [46.101.41.99]

PLAY RECAP ********************************************************************
46.101.41.99               : ok=8    changed=7    unreachable=0    failed=0
$ curl 46.101.41.99
<html><head><title>Hello...
```

❶ This command is needed to map in the SSH key pair used to access the remote server.

This will take some time, as Ansible will need to pull the images. But once it's finished, our identidock application should be running.

This brief example has merely scratched the surface of Ansible's full power. There are many more things you can do, especially in terms of defining processes to perform rolling updates of containers without breaking dependencies or significant downtime.

Host Configuration

So far, this chapter has assumed that containers are being run on the stock Docker droplet (Digital Ocean's term for preconfigured VMs) provided by Digital Ocean (which, at the time of writing, runs Ubuntu 14.04). But there are many other choices for the host operating system and infrastructure with different trade offs and advantages. In particular, if you are responsible for running an on-premise resource, you should consider your options carefully.

Although it is possible to provision bare-metal machines for running Docker hosts (both on-premise and in the cloud), currently the most practical option is to use VMs. Most organizations will already have some sort of VM service you can use to provision hosts for your containers and provides strong guarantees of isolation and security between users.

Choosing an OS

There are already a few choices in this space, with different advantages and disadvantages. If you want to run a small- to medium-sized application, you will probably find it easiest to stick to what you know—if you use Ubuntu or Fedora and you or your organization is familiar with it with that OS, use it (but be aware of the storage driver issues discussed shortly). If, on the other hand, you want to run a very large application or cluster (hundreds or thousands of containers across many hosts), you will want to look at more specialized options such as CoreOS, Project Atomic, or RancherOS, as well as the orchestration solutions we discuss in ???.

If you're running on a cloud host, most of them will already have a Docker image ready to use, which will have been tried and tested to work on their infrastructure.

Choosing a Storage Driver

There are currently several storage drivers supported by Docker, with more on the way. Choosing an appropriate storage driver is essential to ensuring reliability and efficiency in production. Which driver is best depends on your use case and operational experience. The current options are:

AUFS
: The first storage driver for Docker. To date, this is probably the most tested and commonly used driver. Along with Overlay, it has the major advantage of supporting sharing of memory pages between containers—if two containers load libraries or data from the same underlying layer, the OS will be able to use the same memory page for both containers. The major problem with AUFS is that it is not in the mainline kernel, although it has been used by Debian and Ubuntu for some time. Also, AUFS operates on the file level, so if you make a small change to a large file, the whole file will be copied into the container's read/write

layer. In contrast, BTRFS and Device mapper operate on the block level and are therefore more space efficient with large files. If you currently use an Ubuntu or Debian host, you will most likely be using the AUFS driver.

Overlay
: Very similar to AUFS and was merged into the Linux kernel in version 3.18. Overlay is very likely to be the main storage driver going forward and should have slightly better performance than AUFS. Currently, the main drawbacks are the need to have an up-to-date kernel (which will require patching for most distros) and that it has seen less testing than AUFS and some of the other options.

BTRFS
: A copy-on-write filesystem[4] focused on supporting fault tolerance and very large files sizes and volumes. Because BTRFS has several quirks and gotchas (especially regarding chunks), it's recommended only for organizations that have experience with BTRFS or require a particular feature of BTRFS that is not supported by the other drivers. It may be a good choice if your containers read and write to very large files due to the block-level support.

ZFS
: This much-loved filesystem was originally developed by Sun Microsystems. Similar to BTRFS in many regards, but arguably with better performance and reliability. Running ZFS on Linux isn't trivial, as it can't be included in the kernel because of licensing issues. For this reason, it's only likely to be used by organizations with substantial existing experience with ZFS.

Device mapper
: Used by default on Red Hat systems. Device mapper is a kernel driver that is used as a foundation to several other technologies, including RAID, device encryption, and snapshots. Docker uses Device mapper's thin provisioning[5] (sometimes called thinp) target to do copy-on-write on the level of blocks, rather than files. The "thin pool" is allocated from a sparse file that defaults to 100 GB. Containers are allocated a filesystem backed by the pool when created whose size defaults to 100 GB (as of Docker 1.8). As the files are sparse, the actual disk usage is much less, but a container won't be able to grow past 100 GB without changing the defaults. Device mapper is arguably the most complex of the Docker storage drivers and is a common source of problems and support requests. If possible, I would recommend using one of the alternatives. But if you do use device mapper,

4 Don't ask me how to pronounce BTRFS: some people say "ButterFS" and some say "BetterFS." I say "FSCK."

5 In thin provisioning, rather than allocating all the resources a client asks for immediately, resources are only allocated on demand. This contrasts with thick provisioning, where the requested resources are immediately set aside for the client, even though the client may only use a fraction of the resources.

be aware that there are a lot of options that can be tuned to provide better performance (in particular, it's a good idea to move storage off the default "loopback" device and onto a real device).

VFS
: The default Linux Virtual Filesystem. This does not implement CoW and requires making a full copy of the image when starting a container. This slows down starting containers significantly and massively increases the amount of disk space required. The advantages are that it is simple and doesn't require any special kernel features. VFS may be a reasonable choice if you have problems with other drivers and don't mind taking the performance hit (e.g., if you have a small number of long-lived containers).

Unless you have a specific reason to choose an alternative, I would suggest running either AUFS or Overlay, even if it means applying kernel updates.

Switching storage driver

Switching storage driver is pretty easy, assuming you have the requisite dependencies installed. Just restart the Docker daemon, passing the appropriate value for `--storage-driver` (`-s` for short). For example, use `docker daemon -s overlay` to start the daemon with the overlay storage driver if your kernel supports it. It's also important to note the `--graph` or `-g` argument, which sets the root of the Docker runtime—you may need to move this to a partition running the appropriate filesystem (e.g., `docker daemon -s btrfs -g /mnt/btrfs_partition` for the BTRFS driver).

To make the change permanent, you'll need to edit the startup script or config file for the `docker` service. On Ubuntu 14.04, this means editing the variable `DOCKER_OPTS` in the file */etc/default/docker*.

Moving Images Between Storage Drivers

When you switch storage driver, you will lose access to all your old containers and images. Switching back to the old storage driver will restore access. To move an image to a new storage driver, just save the image to a TAR file and then load in the new filesystem. For example:

```
$ docker save -o /tmp/debian.tar debian:wheezy
$ sudo stop docker
$ docker daemon -s vfs
...
```

From a new terminal:

```
$ docker images
REPOSITORY  TAG     IMAGE ID      CREATED     VIRTUAL SIZE
$ docker load -i /tmp/debian.tar
$ docker images
REPOSITORY  TAG     IMAGE ID      CREATED     VIRTUAL SIZE
debian      wheezy b3d362b23ec1  2 days ago  84.96 MB
```

Specialist Hosting Options

There are already some specialist container hosting options that don't require you to manage hosts directly, including Triton, Google Container Engine, Amazon EC2 Container Service, and Giant Swarm. The following subsections take a closer look at each of these options.

Triton

Triton from Joyent is perhaps the most interesting of the options, as it doesn't use VMs internally. This gives Triton a significant performance benefit over VM-based solutions and allows for provisioning on a per-container basis.

Triton doesn't use the Docker engine but has its own container engine running on the SmartOS hypervisor (which has its roots in Solaris) using Linux Virtualization. By implementing the Docker remote API, Triton is fully compatible with the normal Docker client, which is used as the standard interface to Triton. Images from the Docker Hub work as normal.

Triton is open source and available in both a hosted version that runs on the Joyent cloud and an on-premise version. We can quickly get identidock running using the public Joyent public cloud. After setting up a Triton account and pointing the Docker client at Triton, try running a `docker info`:

```
$ docker info
Containers: 0
Images: 0
Storage Driver: sdc
```

```
 SDCAccount: amouat
Execution Driver: sdc-0.3.0
Logging Driver: json-file
Kernel Version: 3.12.0-1-amd64
Operating System: SmartDataCenter
CPUs: 0
Total Memory: 0 B
Name: us-east-1
ID: 92b0cf3a-82c8-4bf2-8b74-836d1dd61003
Username: amouat
Registry: https://index.docker.io/v1/
```

Note the values for the OS and execution driver, which indicate we aren't running on a normal Docker engine. We can use Compose and the following *triton.yml* file to launch identidock, as Triton supports the majority of the Docker engine API:

```
proxy:
  image: amouat/proxy:1.0
  links:
    - identidock
  ports:
   - "80:80"
  environment:
    - NGINX_HOST=www.identidock.com
    - NGINX_PROXY=http://identidock:9090
  mem_limit: "128M"
identidock:
  image: amouat/identidock:1.0
  links:
   - dnmonster
   - redis
  environment:
    ENV: PROD
  mem_limit: "128M"
dnmonster:
  image: amouat/dnmonster:1.0
  mem_limit: "128M"
redis:
  image: redis
  mem_limit: "128M"
```

This is almost the same as the *prod.yml* from before, with the addition of memory settings that tell Triton the size of container to launch. We're also using public images rather than building our own (Triton doesn't currently support `docker build`).

Launch the application:

```
$ docker-compose -f triton.yml up -d
...
Creating triton_proxy_1...
$ docker inspect -f {{.NetworkSettings.IPAddress}} triton_proxy_1
165.225.128.41
```

```
$ curl 165.225.128.41
<html><head><title>Hello...
```

Triton automatically uses a publicly accessible IP when it sees a published port.

After running containers on Triton, make sure to stop and remove them; Triton charges for stopped but not removed containers.

Using the native Docker tools to interact with Triton is a great experience, but there are some rough edges; not all API calls are supported, and there are some issues surrounding how Compose handles volumes, but these should be worked out in time.

Until mainstream cloud providers are convinced that the isolation guarantees of the Linux kernel are strong enough that containers can be run without security concerns, Triton is one of the most attractive solutions for running containerized systems.

Google Container Engine

Google Container Engine (GKE) takes a more opinionated approach to running containers, building on top of the Kubernetes orchestration system.

Kubernetes is an open source project designed by Google, using some of the lessons learned from running containers internally with their Borg cluster manager.[6]

Deploying an application to GKE requires a basic understanding of Kubernetes and the creation of some Kubernetes-specific configuration files (this will be more fully discussed in ???).

In return for this extra work in configuring your application, you get services such as automatic replication and load balancing. These may sound like services that are only needed for large services with high traffic and many distributed parts, but they quickly become important for any service that wants to have any guarantees about up-time.

I'd strongly recommend Kubernetes, and GKE in particular, for deploying container systems, but be aware that this will tie to you to the Kubernetes model, making it more difficult to move your system between providers.

Amazon EC2 Container Service

Amazon's EC2 Container Service (ECS) helps you run containers on Amazon's EC2 infrastructure. ECS provides a web interface and an API for launching containers and managing the underlying EC2 cluster.

6 See the paper, "Large-Scale Cluster Management at Google with Borg", for a fascinating look at how to run a cluster handling hundreds of thousands of jobs.

On each node of the cluster, ECS will start a container agent, which communicates with the ECS service and is responsible for starting, stopping, and monitoring containers.

It's relatively quick to get identidock running on ECS, although it does involve a typical AWS interface with dozens of configuration options. Once you are registered with ECS and have created a cluster, we need to upload a "Task Definition" for identidock. The following JSON can be used as the definition for identidock:

```
{
  "family": "identidock",
  "containerDefinitions": [
    {
      "name": "proxy",
      "image": "amouat/proxy:1.0",
      "cpu": 100,
      "memory": 100,
      "environment": [
        {
          "name": "NGINX_HOST",
          "value": "www.identidock.com"
        },
        {
          "name": "NGINX_PROXY",
          "value": "http://identidock:9090"
        }
      ],
      "portMappings": [
        {
          "hostPort": 80,
          "containerPort": 80,
          "protocol": "tcp"
        }
      ],
      "links": [
        "identidock"
      ],
      "essential": true
    },
    {
      "name": "identidock",
      "image": "amouat/identidock:1.0",
      "cpu": 100,
      "memory": 100,
      "environment": [
        {
          "name": "ENV",
          "value": "PROD"
        }
      ],
      "links": [
        "dnmonster",
```

```
        "redis"
      ],
      "essential": true
    },
    {
      "name": "dnmonster",
      "image": "amouat/dnmonster:1.0",
      "cpu": 100,
      "memory": 100,
      "essential": true
    },
    {
      "name": "redis",
      "image": "redis:3",
      "cpu": 100,
      "memory": 100,
      "essential": false
    }
  ]
}
```

Each container needs to specify an amount of memory (in megabytes) and number of CPU units. The `essential` key defines whether or not the task should be stopped if that container fails. In our case, the Redis container can be considered as nonessential, as the application will still work without it. The other fields should be self-explanatory.

Once the task has been successfully created, it needs to be started on the cluster. Identidock should be started as a *service*, rather than a one-off task. Running as a service means that ECS will monitor the containers to ensure availability and provides the option to connect to Amazon's Elastic Load Balancer to spread traffic between instances. When creating the service, ECS will ask for a name and the number of task instances it should ensure are running. After creating the service and waiting for the task to start, you should be able to access identidock via the IP address of the EC2 instance. This can be found on the task instance details page, in the expanded information for the proxy container.

Stopping the service and associated resources takes several steps. First, the service needs to be updated and the number of tasks changed to 0, to avoid ECS trying to bring up replacement tasks when shutting down. At this point, the service can be deleted. Before the cluster can be deleted, you will also need to deregister the container instances. Be careful to also stop any associated resources you may have started, such as Elastic Load Balancers or EBS storage.

There is a lot of engineering work going on behind the scenes in ECS. It's easy to launch hundreds or thousands of containers with a few clicks, providing serious capabilities for scaling. The scheduling of containers onto hosts is highly configurable, allowing users to optimize for their own needs, such as maximum efficiency or maxi-

mum reliability. Users can replace the default ECS scheduler with their own or use a third-party solution such as Marathon (see ???).

ECS also integrates with existing Amazon features such as Elastic Load Balancer for spreading load over multiple instances and the Elastic Block Store for persistent storage.

Giant Swarm

Giant Swarm bills itself as "an opinionated solution for microservice architectures," which really means it's a fast and easy way to launch a Docker-based system using a specialized configuration format. Giant Swarm offers a hosted version on a shared cluster as well as a dedicated offering (where Giant Swarm will provision and maintain bare-metal hosts for you) and an on-premise solution. At the time of writing, the shared offering is still in alpha, but the dedicated offering is production ready.

Giant Swarm is a rarity in that it makes minimal-to-no use of VMs. Users with strict security requirements have separate bare-metal hosts, but the shared cluster has containers from separate users running next to each other.

Let's see how to run identidock on the Giant Swarm shared cluster. Assuming you've got access to Giant Swarm and installed the Swarm CLI,[7] start by creating the following configuration file and saving it as *swarm.json*:

```
{
    "name": "identidock_svc",
    "components": {
        "proxy": {
            "image": "amouat/proxy:1.0",
            "ports": [80],
            "env": {
                "NGINX_HOST": "$domain",
                "NGINX_PROXY": "http://identidock:9090"
            },
            "links": [ {
                "component": "identidock",
                "target_port": 9090
            }],
            "domains": { "80": "$domain" }
        },
        "identidock": {
            "image": "amouat/identidock:1.0",
            "ports": [9090],
            "links": [
            {
                "component": "dnmonster",
```

7 No relation to Docker's clustering solution, which is also called Swarm.

```
                "target_port": 8080
            },
            {
                "component": "redis",
                "target_port": 6379
            }
            ]
        },
        "redis": {
            "image": "redis:3",
            "ports": [6379]
        },
        "dnmonster": {
            "image": "amouat/dnmonster:1.0",
            "ports": [8080]
        }
    }
}
```

Now it's time to kick identidock into action:

```
$ swarm up --var=domain=identidock-$(swarm user).gigantic.io
Starting service identidock_svc...
Service identidock_svc is up.
You can see all components using this command:

    swarm status identidock_svc

$ swarm status identidock_svc
Service identidock_svc is up

component   image                  instanceid    created              status
dnmonster   amouat/dnmonster:1.0   m6eyoilfiei1  2015-09-04 09:50:40  up
identidock  amouat/identidock:1.0  r22ut7h0vx39  2015-09-04 09:50:40  up
proxy       amouat/proxy:1.0       6dr38cmrg3nx  2015-09-04 09:50:40  up
redis       redis:3                jvcf15d6lpz4  2015-09-04 09:50:40  up
$ curl identidock-amouat.gigantic.io
<html><head><title>Hello...
```

Here we've shown off one of the features that distinguishes Giant Swarm configuration files from Docker Compose—the ability use template variables. In this case, we've passed in the hostname we want on the command line, and Swarm has gone ahead and replaced the `$domain` in the *swarm.json* with this value. Other features provided by *swarm.json* include the ability to define *pods*—groups of containers that are scheduled together—as well as the ability to define how many instances of a container should be running.

Finally, in addition to the Swarm CLI, there is a web UI for monitoring services and viewing logs and a REST API for automating interaction with Giant Swarm.

Persistent Data and Production Containers

Arguably, the data storage story hasn't changed much under Docker, at least at the larger end of the scale. If you run your own databases, you have the choice of using Docker container, VMs, or raw metal. Whenever you have a large amount of data, your VM or container will end up effectively pinned to the host machine due to the difficulties of moving the data around. This means the portability benefits normally associated with containers won't be of help here, but you may still want to use containers to keep a consistent platform and for isolation benefits. If you have concerns about performance, using `--net=host` and `--privileged` will ensure the container is effectively as efficient as the host VM or box, but be aware of the security implications. If you don't run your own databases, but use a hosted service such as Amazon RDS, things continue much as before.

At the smaller end of the scale, where containers have configuration files and moderate amounts of data, you may find volumes limiting, as they tie you to a host machine, making scaling and migrating containers more difficult. You may want to consider moving such data to separate key-value stores or DBs, which you can also run in a container. An interesting alternative approach is to use Flocker to manage your data volumes. Flocker leverages the features of the ZFS filesystem to support the migration of data with containers. If you're trying to take a microservices approach, you will find things a lot simpler if you strive to keep your containers stateless where possible.

Sharing Secrets

You will probably have some sensitive data, such as passwords and API keys, that needs to be securely shared with your containers. The following subsections describe the various approaches to doing this, along with their advantages and disadvantages.[8]

Saving Secrets in the Image

Never do this. It's a bad idea.™

It might be the easiest solution, but it means the secret is now available to anyone with access to the image. It can't be deleted because it will still exist in previous layers. Even if you're using a private registry or not using a registry at all, it would be far too easy for someone to accidentally share the image, and there is no need for everyone who can access the image to know the secret. Also, it ties your image to a specific deployment.

8 If you're using configuration management software such as Ansible to manage container deployment, it may come with or prescribe a solution to this problem.

You *could* store secrets encrypted in images, but then you still need a way of passing the decryption key, and you are unnecessarily giving attackers something to work with.

Just forget about this idea. I only included it here so I can point at this section when someone does it and it goes horribly wrong.

Passing Secrets in Environment Variables

Using environment variables to pass secrets is a very straightforward solution and is considerably better than baking secrets into the image. It's simple to do: just pass the secrets as arguments to `docker run`. For example:

```
$ docker run -d -e API_TOKEN=my_secret_token myimage
```

A better method is to pass the variables in via a file, which has the advantage of keeping them from appearing in shell history or the output of the `ps` command (which will be visible to other users on a shared host):

```
$ cat pass.txt
API_TOKEN=my_secret_token
$ docker run -d --env-file ./pass.txt myimage
```

Many applications and configuration files will support using environment variables directly. For the rest, you may need some scripting similar to what we did in "Using a Proxy".

This is the method recommended by The Twelve-Factor App, a popular and respected methodology for building software-as-a-service applications.[9] While I would strongly recommend reading this document and implementing most of the advice, storing secrets in the environment has some serious drawbacks, including:

- Environment variables are visible to all child processes, `docker inspect`, and any linked containers. None of these has a good reason for being able to see these secrets.
- The environment is often saved for logging and debugging purposes. There is a large risk of secrets appearing in debug logs and issue trackers.
- They can't be deleted. Ideally we would overwrite or wipe the secret after using it, but this isn't possible with Docker containers.

For these reasons, I would advise against using this method.

9 It's worth pointing out that the Twelve-Factor methodology predates Docker containers, so some advice needs to be adapted.

Passing Secrets in Volumes

A slightly better—but still far from perfect—solution is to use volumes to share secrets. For example:

```
$ docker run -d -v $PWD:/secret-file:/secret-file:ro myimage
```

Unless you map in whole configuration files with secrets, you will probably require some scripting to handle secrets passed this way. If you're feeling really clever, it is possible to create a temporary file with the secret and delete the file after reading it (be careful not to delete the original though!).

For configuration files that use environment variables, you can also create a script that sets up the environment variables and can be sourced prior to running the appropriate application. For example:

```
$ cat /secret/env.sh
export DB_PASSWORD=s3cr3t
$ source /secret/env.sh && run_my_app.sh
...
```

This has the important advantage of not exposing the variables to `docker inspect` or linked containers.

The major drawback with this approach is that it requires you to keep your secrets in files, which are all too easy to check into version control. It can also be a more fiddly solution that typically requires scripting.

Using a Key-Value Store

Arguably the best solution is to use a key-value store to keep secrets and retrieve them from the container at runtime. This allows a level of control over the secrets that isn't possible with the previous options, but also requires more set up and putting your trust in the key-value store.

Some solutions in this area include:

KeyWhiz
: Stores secrets encrypted in memory and provides access via a REST API and a CLI. Developed and used by Square (a payment-processing company).

Vault
: Can store secrets encrypted in a variety of backends, including file and Consul. Also has a CLI and API. Has several features not currently present in KeyWhiz, but is arguably less mature. Developed by HashiCorp, which is also behind the Consul service discovery tool and the Terraform infrastructure configuration tool.

Crypt
: Stores values encrypted in the etcd or Consul key-value stores. The major advantage with this approach is that it allows a degree of control over the secrets that wasn't previously possible. It becomes easy to change and delete secrets, apply "leases" to secrets so they expire after a given time period, or to lock down access to secrets in case of a security alert.

However, there is still a problem here: how does the container authenticate itself to the store? Typically, you will still need to pass the container either a private key using a volume or a token via an environment variable. The previous objections to using an environment variable can be mitigated by creating a *one-use token* that is revoked immediately after use. Another solution currently in development is to use a volume plugin for the store that mounts secrets from the store as a file inside the container. GitHub has more information on this approach with regard to the KeyWhiz store.

This type of solution will be the future. The level of control it provides over sensitive data is more than worth any complications in implementation, which should be reduced as tooling improves. However, you may wish to wait and see how the sector evolves before making a decision. In the meantime, use volumes to share your secrets, but be very careful not to check them into SCM.

Networking

Networking is discussed in depth in ???. However, it is worth noting that if you're using the stock Docker networking in production, you are taking a considerable performance hit—setting up the Docker bridge and using veth[10] means that a lot of network routing is happening in user space, which is a lot slower than being handled by routing hardware or the kernel.

Production Registry

With identidock, we've just been using the Docker Hub to retrieve our images. Most production setups will include a registry (or multiple registries) to provide fast access to images and avoid relying on a third party for crucial infrastructure (some organizations will also be uneasy about storing their code with a third party, whether it's in a private repository or not). For details on setting up a registry, refer back to ???.

Keeping the images inside the registry up to date and correct is important—you don't want hosts to be able to pull old and potentially vulnerable images. For this reason, it's a good idea to run regular audits on registries, as discussed in ???. However,

10 Virtual Ethernet, or veth, is a virtual network device with its own MAC address that was developed for use in VMs.

remember that each Docker host will also maintain its own cache of images, which also needs to be checked.

The Docker distribution project is currently working on supporting highly available and scalable registry deployments using techniques such as mirroring.

Continuous Deployment/Delivery

Continuous delivery is the extension of continuous integration to production; engineers should be able to make changes in development, have them run through testing, and then have them be available for deployment at the touch of a button. Continuous deployment takes this a step further and automatically pushes changes that pass testing to deployment.

We saw in Chapter 1 how to set up a continuous integration system using Jenkins. Extending this to continuous deployment can be achieved by pushing images to the production registry and migrating running containers to the new image. Migrating images without downtime requires bringing up new containers and rerouting traffic before stopping the old containers. As discussed in "Testing in Production", there are several possible ways to achieve this in a safe manner, such as blue/green deployments and ramped deployments. Implementing these techniques is often done with in-house tooling, although frameworks such as Kubernetes offer built-in solutions, and I expect to see specialist tools arrive on the market.

Conclusion

We've covered a great deal of information in this chapter-there are a lot of different aspects to consider when deploying containers to production, even with something as simple as identidock.

Although the container space is still very young, there are already several production-grade options for hosting containers. The best option to choose is dependent on the size and complexity of your system and how much effort and money you are willing to expend on deployment and maintenance. Small deployments can be managed by simply running a Docker Engine on a VM in the cloud, but this incurs a large maintenance burden with larger deployments. This can be mitigated by using systems such as Kubernetes and Mesos, which are discussed in ???, or by using a specialist hosting service such as Giant Swarm, Triton, or ECS.

In this chapter, we looked at some of the issues commonly faced in production, from tasks as seemingly simple as launching containers to thorny issues such as passing secrets, handling data volumes, and continuous deployment. Some of these issues require new approaches in a containerized system, especially when it is comprised of dynamic microservices. New patterns and best practices will be developed to deal

with these issues, leading to new tooling and frameworks. Containers can already be used reliably in production, but the future is even brighter.

About the Author

Adrian Mouat is the chief scientist for Container Solutions, a pan-European services company that specializes in Docker and Mesos. Previously, he was an applications consultant at EPCC, part of the University of Edinburgh.

Colophon

The animal on the cover of *Using Docker* is a bowhead whale (*Balaena mysticetus*). It is a dark-colored, stocky whale, notable for its lack of dorsal fin. Bowhead whales live their lives in Arctic and sub-Arctic waters, unlike other whales that migrate to low-latitude waters to feed or reproduce.

Bowhead whales are large and robust, growing up to 53 feet (males) and 59 feet (females). They have massive, triangular skulls that they use to break through Arctic ice in order to breathe. Bowhead whales have strongly bowed, white lower jaws and narrow upper jaws, which house the longest baleen of any whale (at 9.8 feet) and is used to strain tiny prey from the water. Paired blowholes are found at the highest point of the whale's head; they can spout water 20 feet high. It boasts the thickest blubber of any animal, ranging from 17–20 inches thick.

Bowhead whales travel alone or in small pods of six. They can remain underwater for up to an hour, but tend to limit their single dives to 4–15 minutes. These whales typically travel about 2–5 kilometers per hour—slow for a whale—but when in danger, they can reach speeds of 10 km/hr. Despite not being very social, bowhead whales are the most vocal of large whales. They communicate using underwater sounds while traveling, socializing, and feeding. During mating season, bowheads make long, complex songs as mating calls.

These whales are known as the longest-living mammals, with an average lifespan of over 200 years. In 2007, a 49-foot bowhead whale was caught off the coast of Alaska with an explosive harpoon head found embedded in its neck blubber. The weapon was traced back to a major whaling center in New Bedford, Massachusetts, and determined to have been manufactured in 1890. Other bowhead whales have been aged between 135 and 172 years old. Once in danger of extinction, bowhead whales have increased since commercial whaling ceased. Small numbers (25–40) are still hunted during subsistence hunts by Alaska natives, but this level of hunt is not expected to affect the population's recovery.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from *Braukhaus Lexicon*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.