9 Continuous delivery: A perfect fit for Docker principles

This chapter covers

- The Docker contract between dev and ops
- Taking manual control over build availability across environments
- Moving builds between environments over low-bandwidth connections
- Centrally configuring all containers in an environment
- Achieving zero-downtime deployment with Docker

Once you're confident that all of your builds are being quality-checked with a consistent CI process, the logical next step is to start looking at deploying every good build to your users. This goal is known as continuous delivery (CD).

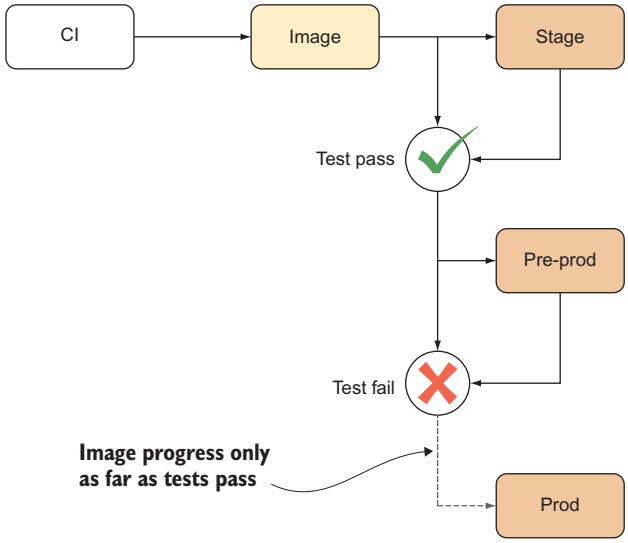

Figure 9.1 A typical CD pipeline

In this chapter we'll refer to your "CD pipeline"—the process your build goes through after it comes out of your "CI pipeline." The dividing line can sometimes be blurred, but think of the CD pipeline as starting when you have a final image that has passed your initial tests during the build process. Figure 9.1 demonstrates how the image might progress through a CD pipeline until it (hopefully) reaches production.

It's worth repeating that last point—the image that comes out of CI should be final and unmodified throughout your CD process! Docker makes this easy to enforce with immutable images and encapsulation of state, so using Docker takes you one step down the CD road already.

When this chapter is done, you'll fully understand why Docker's immutability makes it a perfect partner for your CD strategy. In this way, Docker can be a key enabler for any DevOps strategy in any organization.

9.1 Interacting with other teams in the CD pipeline

First we're going to take a little step back and look at how Docker changes the relationship between development and operations.

Some of the biggest challenges in software development aren't technical—splitting people up into teams based on their roles and expertise is a common practice, yet this can result in communication barriers and insularity. Having a successful CD pipeline requires involvement from the teams at all stages of the process, from development to testing to production. Having a single reference point for all teams can help ease this interaction by providing structure.

TECHNIQUE 70 The Docker contract: Reducing friction

One of Docker's aims is to allow easy expression of inputs and outputs as they relate to a container that contains a single application. This can provide clarity when working

with other people—communication is a vital part of collaboration, and understanding how Docker can ease things by providing a single reference point can help you win over Docker unbelievers.

PROBLEM

You want cooperating teams' deliverables to be clean and unambiguous, reducing friction in your delivery pipeline.

SOLUTION

Use the *Docker contract* to facilitate clean deliverables between teams.

As companies scale, they frequently find that the flat, lean organization they once had, in which key individuals "knew the whole system," gives way to a more structured organization within which different teams have different responsibilities and competencies. We've seen this firsthand in the organizations we've worked at.

If technical investment isn't made, friction can arise as growing teams deliver to each other. Complaints of increasing complexity, "throwing the release over the wall," and buggy upgrades all become familiar. Cries of "Well, it works on our machine!" will increasingly be heard, to the frustration of all concerned. Figure 9.2 gives a simplified but representative view of this scenario.

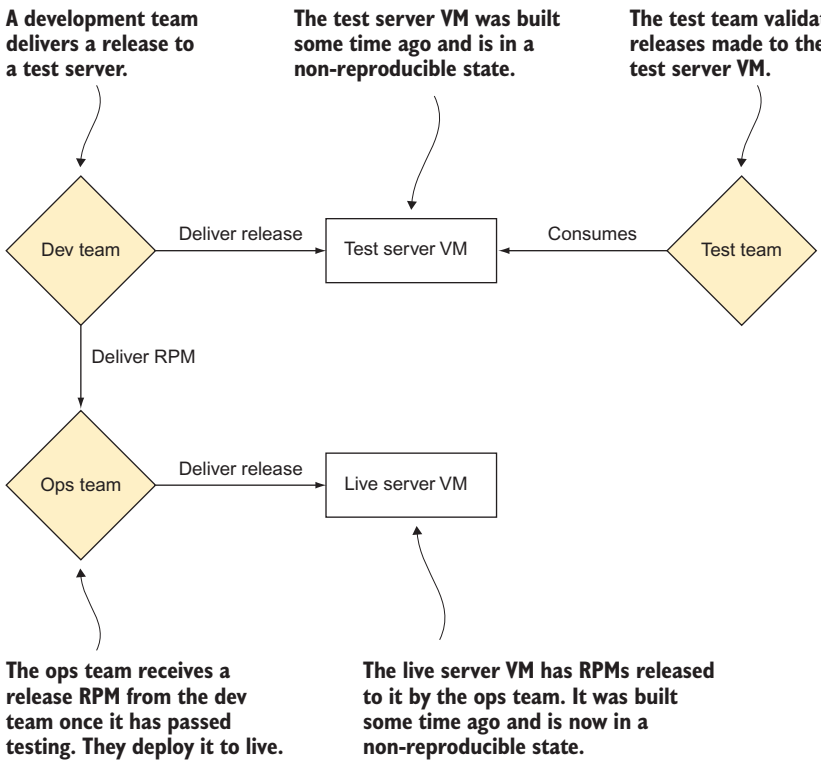

Figure 9.2 Before: a typical software workflow

The workflow in figure 9.2 has a number of problems that may well look familiar to you. They all boil down to the difficulties of managing state. The test team might test something on a machine that differs from what the operations team has set up. In theory, changes to all environments should be carefully documented, rolled back when problems are seen, and kept consistent. Unfortunately, the realities of commercial pressure and human behavior routinely conspire against this goal, and environmental drift is seen.

Existing solutions to this problem include VMs and RPMs. VMs can be used to reduce the surface area of environmental risk by delivering complete machine representations to other teams. The downside is that VMs are relatively monolithic entities that are difficult for teams to manipulate efficiently. At the other end, RPMs offer a standard way of packaging applications that helps define dependencies when rolling out software. This doesn't eliminate configuration management issues, though, and rolling out RPMs created by fellow teams is far more error-prone than using RPMs that have been battle-tested across the internet.

THE DOCKER CONTRACT

What Docker can do is give you a clean line of separation between teams, where the Docker image is both the borderline and the unit of exchange. We call this the *Docker contract*, illustrated in figure 9.3.

With Docker, the reference point for all teams becomes much cleaner. Rather than dealing with sprawling monolithic virtual (or real) machines in unreproducible states, all teams are talking about the same code, whether it's on test, live, or development. In addition, there's a clean separation of data from code, which makes it easier to reason about whether problems are caused by variations in data or code.

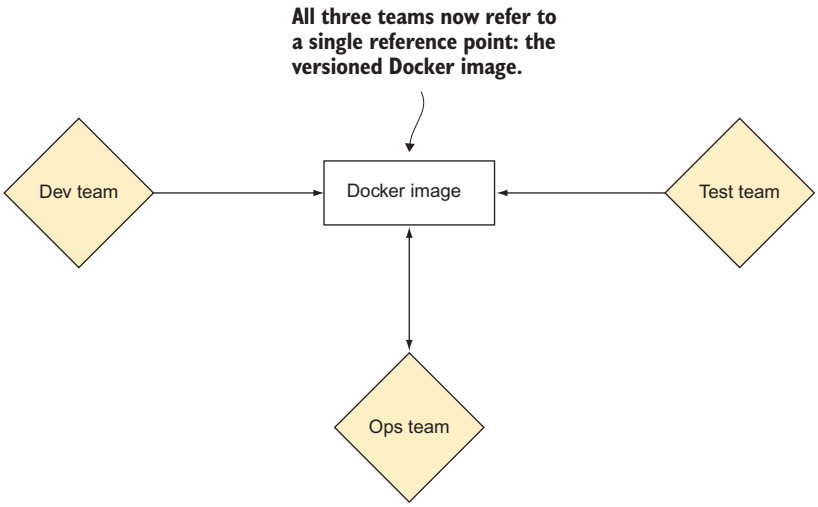

Figure 9.3 After: the Docker contract

Because Docker uses the remarkably stable Linux API as its environment, teams that deliver software have far more freedom to build software and services in whatever fashion they like, safe in the knowledge that it will run predictably in various environments. This doesn't mean that you can ignore the context in which it runs, but it does reduce the risk of environmental differences causing issues.

Various operational efficiencies result from having this single reference touchpoint. Bug reproduction becomes much easier, as all teams are able to describe and reproduce issues from a known starting point. Upgrades become the responsibility of the team delivering the change. In short, state is managed by those making the change. All these benefits greatly reduce the communications overhead and allow teams to get on with their jobs. This reduced communications overhead can also help encourage moves toward a microservices architecture.

This is no merely theoretical benefit: we've seen this improvement firsthand in a company of over 500 developers, and it's a frequent topic of discussion at Docker technical meetups.

DISCUSSION

This technique outlines a strategy, useful to keep in mind as you continue throughout the book, to identify how other techniques fit into this new world. For example, technique 76 describes a way to run a microservices-based application in the same cross-container way it would be run on production systems, eliminating a source of configuration file tweaking. When you do find yourself with external URLs or other unchanging factors on different environments, technique 85 will step up with information about service discovery—a good way to turn a sprawl of configuration files into a single source of truth.

9.2 Facilitating deployment of Docker images

The first problem when trying to implement CD is moving the outputs of your build process to the appropriate location. If you're able to use a single registry for all stages of your CD pipeline, it may seem like this problem has been solved. But it doesn't cover a key aspect of CD.

One of the key ideas behind CD is *build promotion*: each stage of a pipeline (user acceptance tests, integration tests, and performance tests) can only trigger the next stage if the previous one has been successful. With multiple registries you can ensure that only *promoted* builds are used by only making them available in the next registry when a build stage passes.

We'll look at a few ways of moving your images between registries, and even at a way of sharing Docker objects without a registry.

TECHNIQUE 71 Manually mirroring registry images

The simplest image-mirroring scenario is when you have a machine with a high-bandwidth connection to both registries. This permits the use of normal Docker functionality to perform the image copy.

PROBLEM

You want to copy an image between two registries.

SOLUTION

Manually use the standard pulling and pushing commands in Docker to transfer the image.

The solution for this involves:

- Pulling the image from the registry
- Retagging the image
- Pushing the retagged image

If you have an image at test-registry.company.com and you want to move it to stage-registry.company.com, the process is simple.

Listing 9.1 Transferring an image from a test to a staging registry

```
$ IMAGE=mygroup/myimage:mytag
$ OLDREG=test-registry.company.com
$ NEWREG=stage-registry.company.com
$ docker pull $OLDREG/$MYIMAGE
[...]
$ docker tag -f $OLDREG/$MYIMAGE $NEWREG/$MYIMAGE
$ docker push $NEWREG/$MYIMAGE
$ docker rmi $OLDREG/$MYIMAGE
$ docker image prune -f
```

There are three important points to note about this process:

1. The new image has been force tagged. This means that any older image with the same name on the machine (left there for layer-caching purposes) will lose the image name, so the new image can be tagged with the desired name.
2. All dangling images have been removed. Although layer caching is extremely useful for speeding up deployment, leaving unused image layers around can quickly use up disk space. In general, old layers are less likely to be used as time passes and they become more out-of-date.
3. You may need to log into your new registry with `docker login`.

The image is now available in the new registry for use in subsequent stages of your CD pipeline.

DISCUSSION

This technique illustrates a simple point about Docker tagging: the tag itself contains information about the registry it belongs to.

Most of the time this is hidden from users because they normally pull from the default registry (the Docker Hub at docker.io). When you're starting to work with registries, this issue comes to the fore because you have to explicitly tag the registry with the registry location in order to push it to the correct endpoint.

TECHNIQUE 72 Delivering images over constrained connections

Even with layering, pushing and pulling Docker images can be a bandwidth-hungry process. In a world of free large-bandwidth connections, this wouldn't be a problem, but sometimes reality forces us to deal with low-bandwidth connections or costly bandwidth metering between data centers. In this situation you need to find a more efficient way of transferring differences, or the CD ideal of being able to run your pipeline multiple times a day will remain out of reach.

The ideal solution is a tool that will reduce the average size of an image so it's even smaller than classic compression methods can manage.

PROBLEM

You want to copy an image between two machines with a low-bandwidth connection between them.

SOLUTION

Export the image, split it up, transfer the chunks, and import the recombined image on the other end.

To do all this, we must first introduce a new tool: bup. It was created as a backup tool with extremely efficient deduplication—*deduplication* being the ability to recognize where data is used repeatedly and only store it once. It works particularly well on archives containing a number of similar files, which is conveniently a format Docker allows you to export images as.

For this technique we've created an image called dbup (short for "Docker bup"), which makes it easier to use bup to deduplicate images. You can find the code behind it at https://github.com/docker-in-practice/dbup.

As a demonstration, let's see how much bandwidth we could save when upgrading from the ubuntu:14.04.1 image to ubuntu:14.04.2. Bear in mind that in practice you'd have a number of layers on top of each of these, which Docker would want to completely retransfer after a lower layer change. By contrast, this technique would recognize the significant similarities and give you much greater savings than you'll see in the following example.

The first step is to pull both of those images so we can see how much is transferred over the network.

Listing 9.2 Examining and saving two Ubuntu images

```
$ docker pull ubuntu:14.04.1 && docker pull ubuntu:14.04.2
[...]
$ docker history ubuntu:14.04.1
IMAGE        CREATED     CREATED BY                                      SIZE
ab1bd63e0321 2 years ago /bin/sh -c #(nop) CMD [/bin/bash]               0B
<missing>    2 years ago /bin/sh -c sed -i 's/^#\s*\(deb.*universe\... 1.9kB
<missing>    2 years ago /bin/sh -c echo '#!/bin/sh' > /usr/sbin/po... 195kB
<missing>    2 years ago /bin/sh -c #(nop) ADD file:62400a49cced0d7... 188MB
<missing>    4 years ago                                                 0B
$ docker history ubuntu:14.04.2
```

```
IMAGE          CREATED       CREATED BY                                      SIZE
44ae5d2a191e 2 years ago /bin/sh -c #(nop) CMD ["/bin/bash"]                  0B
<missing>      2 years ago /bin/sh -c sed -i 's/^#\s*\(deb.*universe\... 1.9kB
<missing>      2 years ago /bin/sh -c echo '#!/bin/sh' > /usr/sbin/po... 195kB
<missing>      2 years ago /bin/sh -c #(nop) ADD file:0a5fd3a659be172... 188MB
$ docker save ubuntu:14.04.1 | gzip | wc -c
65973497
$ docker save ubuntu:14.04.2 | gzip | wc -c
65994838
```

The bottom layer on each image (the `ADD`) is the majority of the size, and you can see that the file being added is different, so you can treat the whole image size as the amount that would be transferred when pushing the new image. Also note that the Docker registry uses gzip compression to transfer layers, so we've included that in our measurement (instead of taking the size from `docker history`). About 65 MB is being transferred in both the initial deployment and the subsequent deployment.

In order to get started, you'll need two things—a directory to store the pool of data bup uses as storage, and the dockerinpractice/dbup image. You can then go ahead and add your image to the bup data pool.

Listing 9.3 Saving two Ubuntu images into the bup data pool

```
$ mkdir bup_pool
$ alias dbup="docker run --rm \
    -v $(pwd)/bup_pool:/pool -v /var/run/docker.sock:/var/run/docker.sock \
    dockerinpractice/dbup"
$ dbup save ubuntu:14.04.1
Saving image!
Done!
$ du -sh bup_pool
74M     bup_pool
$ dbup save ubuntu:14.04.2
Saving image!
Done!
$ du -sh bup_pool
96M     bup_pool
```

Adding the second image to the bup data pool has only increased the size by about 20 MB. Assuming you synced the folder to another machine (possibly with rsync) after adding ubuntu:14.04.1, syncing the folder again will only transfer 20 MB (as opposed to the 65 MB before).

You then need to load the image at the other end.

Listing 9.4 Loading an image from the bup data pool

```
$ dbup load ubuntu:14.04.1
Loading image!
Done!
```

The process for transferring between registries would look something like this:

1. `docker pull` on host1
2. `dbup save` on host1
3. `rsync` from host1 to host2
4. `dbup load` on host2
5. `docker push` on host2

This technique opens up a number of possibilities that may not have been possible previously. For example, you can now rearrange and consolidate layers without having to worry about how long it will take to transfer all of the new layers over the low-bandwidth connection.

DISCUSSION

Even when following best practices and adding your application code as the last stage, bup may be able to help—it will recognize that most of the code is unchanged and only add the difference to the data pool.

The data pools can be very large, such as database files, and bup will likely perform very well (which is useful if you've decided to use technique 77 with the database inside the container, meaning there's no volume). This is actually somewhat unusual—database exports and backups are typically very efficient to incrementally transfer, but the actual on-disk storage of databases can vary significantly and occasionally defeat tools like rsync. On top of this, dbup puts a full history of your images at your fingertips—there's no need to store three full copies of images to roll back to. You can pull them out of the pool at your leisure. Unfortunately there's currently no way to clean the pool of images you don't want any more, so you'll probably need to clear the pool every now and again.

Although you may not see an immediate need for dbup, keep it in mind, in case your bandwidth bills start growing.

TECHNIQUE 73 Sharing Docker objects as TAR files

TAR files are a classic method of moving files around on Linux. Docker allows you to create TAR files and ship them around manually when there's no registry available and no possibility of setting one up. Here we're going to show you the ins and outs of these commands.

PROBLEM

You want to share images and containers with others, with no available registry.

SOLUTION

Use `docker export` or `docker save` to create TAR file artifacts, and then consume them with `docker import` or `docker load` over SSH.

The distinctions between the commands can be difficult to grasp if you're using them casually, so let's take a moment to quickly go over what they do. Table 9.1 outlines the inputs and outputs of the commands.

Table 9.1 Export and import vs. save and load

Command	Creates?	Of what?	From what?
`export`	TAR file	Container filesystem	Container
`import`	Docker image	Flat filesystem	TAR file
`save`	TAR file	Docker image (with history)	Image
`load`	Docker image	Docker image (with history)	TAR file

The first two commands work with flat filesystems. The command `docker export` outputs a TAR file of the files that make up the state of the container. As always with Docker, the state of running processes isn't stored—only the files. The command `docker import` creates a Docker image—with no history or metadata—from a TAR file.

These commands aren't symmetrical—you can't create a container from an existing container using only `import` and `export`. This asymmetry can be useful because it allows you to `docker export` an image to a TAR file, and then `docker import` it to "lose" all the layer history and metadata. This is the image-flattening approach described in technique 52.

If you're exporting or saving to a TAR file, the file is sent to `stdout` by default, so make sure you save it to a file like this:

```
docker pull debian:7:3
[...]
docker save debian:7.3 > debian7_3.tar
```

A TAR file like the one just created can be flung around the network safely (though you may want to compress it with `gzip` first), and other people can use them to import images intact. They can be sent by email or `scp` if you have access:

```
$ scp debian7_3.tar example.com:/tmp/debian7_3.tar
```

You can take this one step further and deliver images to other users' Docker daemons directly—assuming you have the permission.

Listing 9.5 Sending an image directly over SSH

```
docker save debian:7.3 | \
 ssh example.com \
 docker load -
```

Extracts the Debian version 7.3 image and pipes it to the ssh command

Runs a command on a remote machine, example.com

Takes the TAR file it's given and creates an image with all the history. The dash indicates that the TAR file is being delivered over standard input.

If you want to discard the history of the image, you can use `import` instead of `load`.

Listing 9.6 Transferring a Docker image directly over SSH, discarding layers

```
docker export $(docker run -d debian:7.3 true) | \
    ssh example.com docker import
```

NOTE Unlike `docker import`, `docker load` doesn't require a dash at the end to indicate that the TAR file is being delivered through standard input.

DISCUSSION

You may remember the export and import process from technique 52, where you saw how flattening images can be used to remove secrets that may be hidden in lower layers. The fact that secrets may be accessible in lower layers is worth bearing in mind if you're transferring images to other people—realizing that you deleted your public key on the top image layer but that it's available lower down can be a real hassle, because you should then treat it as compromised and change it everywhere.

If you find yourself doing the image transfer in this technique a lot, it may be worth putting a little time into technique 9 to set up your own registry and make things less ad hoc.

9.3 Configuring your images for environments

As mentioned in the introduction to this chapter, one of the keystones of CD is the concept of "doing the same thing everywhere." Without Docker, this would mean building a deployment artifact once and using the same one everywhere. In a Dockerized world, this means using the same image everywhere.

But environments aren't all the same—there may be different URLs for external services, for example. For "normal" applications you'd be able to use environment variables to solve this problem (with the caveat that they're not easy to apply to numerous machines). The same solution can work for Docker (explicitly passing the variables in), but there's a better way of doing it with Docker that comes with some additional benefits.

TECHNIQUE 74 Informing your containers with etcd

Docker images are designed so they can be deployed anywhere, but you'll often want to add some extra information after deployment to affect the behavior of the application while it's running. In addition, machines running Docker may need to remain unaltered, so you may need an external source of information (making environment variables less suitable).

PROBLEM

You need an external source of configuration when running containers.

SOLUTION

Set up etcd, a distributed key/value store, to hold your container configuration.

etcd holds pieces of information and can be part of a multinode cluster for resiliency. In this technique you'll create an etcd cluster to hold your configuration and use an etcd proxy to access it.

NOTE Each value held by etcd should be kept small—under 512 KB is a good rule of thumb; past this point you should consider doing benchmarking to verify that etcd is still performing as you'd expect. This limit is not unique to etcd. You should bear it in mind for other key/value stores like Zookeeper and Consul.

Because etcd cluster nodes need to talk to each other, the first step is to identify your external IP address. If you were going to run the nodes on different machines, you'd need the external IP for each of them.

Listing 9.7 Identifying IP addresses of the local machine

```
$ ip addr | grep 'inet ' | grep -v 'lo$\|docker0$'
    inet 192.168.1.123/24 brd 192.168.1.255 scope global dynamic wlp3s0
    inet 172.18.0.1/16 scope global br-0c3386c9db5b
```

Here we've looked for all IPv4 interfaces and excluded LoopBack and Docker. The top line (the first IP address on that line) is the one you need, and it represents the machine on the local network—try pinging it from another machine if you're not sure.

We can now get started with the three-node cluster, all running on the same machine. Be careful with the following arguments—the ports being exposed and advertised change on each line, as do the names of the cluster nodes and containers.

Listing 9.8 Setting up a three-node etcd cluster

The external IP address of your machine

Uses the external IP address of the machine in the cluster definition, giving the nodes a way to communicate with others. Because all nodes will be on the same host, the cluster ports (for connecting to other nodes) must be different.

```
$ IMG=quay.io/coreos/etcd:v3.2.7
$ docker pull $IMG
[...]
$ HTTPIP=http://192.168.1.123
 $ CLUSTER="etcd0=$HTTPIP:2380,etcd1=$HTTPIP:2480,etcd2=$HTTPIP:2580"
 $ ARGS="etcd"
$ ARGS="$ARGS -listen-client-urls http://0.0.0.0:2379"
 $ ARGS="$ARGS -listen-peer-urls http://0.0.0.0:2380"
 $ ARGS="$ARGS -initial-cluster-state new"
$ ARGS="$ARGS -initial-cluster $CLUSTER"
$ docker run -d -p 2379:2379 -p 2380:2380 --name etcd0 $IMG \
    $ARGS -name etcd0 -advertise-client-urls $HTTPIP:2379 \
    -initial-advertise-peer-urls $HTTPIP:2380
912390c041f8e9e71cf4cc1e51fba2a02d3cd4857d9ccd90149e21d9a5d3685b
$ docker run -d -p 2479:2379 -p 2480:2380 --name etcd1 $IMG \
    $ARGS -name etcd1 -advertise-client-urls $HTTPIP:2479 \
    -initial-advertise-peer-urls $HTTPIP:2480
446b7584a4ec747e960fe2555a9aaa2b3e2c7870097b5babe65d65cffa175dec
$ docker run -d -p 2579:2379 -p 2580:2380 --name etcd2 $IMG \
    $ARGS -name etcd2 -advertise-client-urls $HTTPIP:2579 \
    -initial-advertise-peer-urls $HTTPIP:2580
3089063b6b2ba0868e0f903a3d5b22e617a240cec22ad080dd1b497ddf4736be
```

The port for handling requests from clients

The port to listen on for talking to other nodes in the cluster, corresponding to the ports specified in $CLUSTER

```
$ curl -L $HTTPIP:2579/version
{"etcdserver":"3.2.7","etcdcluster":"3.2.0"}
$ curl -sSL $HTTPIP:2579/v2/members | python -m json.tool | grep etcd
            "name": "etcd0",
             "name": "etcd1",
             "name": "etcd2",
```

Currently connected nodes in the cluster

You've now started up the cluster and have a response from one node. In the preceding commands, anything referring to "peer" is controlling how the etcd nodes find and talk to each other, and anything referring to "client" defines how other applications can connect to etcd.

Let's see the distributed nature of etcd in action.

Listing 9.9 Testing the resilience of an etcd cluster

```
$ curl -L $HTTPIP:2579/v2/keys/mykey -XPUT -d value="test key"
{"action":"set","node": >
{"key":"/mykey","value":"test key","modifiedIndex":7,"createdIndex":7}}
$ sleep 5
$ docker kill etcd2
etcd2
$ curl -L $HTTPIP:2579/v2/keys/mykey
curl: (7) couldn't connect to host
$ curl -L $HTTPIP:2379/v2/keys/mykey
{"action":"get","node": >
{"key":"/mykey","value":"test key","modifiedIndex":7,"createdIndex":7}}
```

In the preceding code, you add a key to your etcd2 node and then kill it. But etcd has automatically replicated the information to the other nodes and is able to provide you with the information anyway. Although the preceding code paused for five seconds, etcd will typically replicate in under a second (even across different machines). Feel free to `docker start etcd2` now to make it available again—any changes you've made in the meantime will replicate back to it.

You can see that the data is still available, but it's a little unfriendly to have to manually choose another node to connect to. Fortunately etcd has a solution for this—you can start a node in "proxy" mode, which means it doesn't replicate any data; rather, it forwards the requests to the other nodes.

Listing 9.10 Using an etcd proxy

```
$ docker run -d -p 8080:8080 --restart always --name etcd-proxy $IMG \
    etcd -proxy on -listen-client-urls http://0.0.0.0:8080 \
    -initial-cluster $CLUSTER
037c3c3dba04826a76c1d4506c922267885edbfa690e3de6188ac6b6380717ef
$ curl -L $HTTPIP:8080/v2/keys/mykey2 -XPUT -d value="t"
{"action":"set","node": >
{"key":"/mykey2","value":"t","modifiedIndex":12,"createdIndex":12}}
$ docker kill etcd1 etcd2
$ curl -L $HTTPIP:8080/v2/keys/mykey2
```

```
{"action":"get","node": >
{"key":"/mykey2","value":"t","modifiedIndex":12,"createdIndex":12}}
```

This now gives you some freedom to experiment with how etcd behaves when over half the nodes are offline.

Listing 9.11 Using etcd with more than half the nodes down

```
$ curl -L $HTTPIP:8080/v2/keys/mykey3 -XPUT -d value="t"
{"errorCode":300,"message":"Raft Internal Error", >
"cause":"etcdserver: request timed out","index":0}
$ docker start etcd2
etcd2
$ curl -L $HTTPIP:8080/v2/keys/mykey3 -XPUT -d value="t"
{"action":"set","node": >
{"key":"/mykey3","value":"t","modifiedIndex":16,"createdIndex":16}}
```

Etcd permits reading but prevents writing when half or more of the nodes are not available.

You can now see that it would be possible to start an etcd proxy on each node in a cluster to act as an "ambassador container" for retrieving centralized configuration, as follows.

Listing 9.12 Using an etcd proxy inside an ambassador container

```
$ docker run -it --rm --link etcd-proxy:etcd ubuntu:14.04.2 bash
root@8df11eaae71e:/# apt-get install -y wget
root@8df11eaae71e:/# wget -q -O- http://etcd:8080/v2/keys/mykey3
{"action":"get","node": >
{"key":"/mykey3","value":"t","modifiedIndex":16,"createdIndex":16}}
```

> **TIP** An ambassador is a so-called "Docker pattern" that has some currency among Docker users. An ambassador container is placed between your application container and some external service and handles the request. It's similar to a proxy, but it has some intelligence baked into it to handle the specific requirements of the situation—much like a real-life ambassador.

Once you have an etcd running in all environments, creating a machine in an environment is just a matter of starting it up with a link to an etcd-proxy container—all CD builds to the machine will then use the correct configuration for the environment. The next technique shows how to use etcd-provided configuration to drive zero-downtime upgrades.

DISCUSSION

The ambassador container shown in the previous section draws on the link flag introduced in technique 8. As noted there, linking has fallen somewhat out of favor in the Docker world, and a more idiomatic way to achieve the same thing now is with named containers on a virtual network, covered in technique 80.

Having a cluster of key/value servers providing a consistent view of the world is a big step forward from managing configuration files on many machines, and it helps you push toward fulfilling the Docker contract described in technique 70.

9.4 Upgrading running containers

In order to achieve the ideal of multiple deployments to production every day, it's important to reduce downtime during the final step of the deployment process—turning off the old applications and starting up the new ones. There's no point deploying four times a day if the switchover is an hour-long process each time!

Because containers provide an isolated environment, a number of problems are already mitigated. For example, you don't need to worry about two versions of an application using the same working directory and conflicting with each other, or about rereading some configuration files and picking up new values without restarting using the new code.

Unfortunately there are some downsides to this—it's no longer simple to change files in-place, so soft-restarts (required to pick up configuration file changes) become harder to achieve. As a result, we've found it a best practice to always perform the same upgrade process regardless of whether you're changing a few configuration files or thousands of lines of code.

Let's look at an upgrade process that will achieve the gold standard of zero-downtime deployment for web-facing applications.

TECHNIQUE 75 Using confd to enable zero-downtime switchovers

Because containers can exist side by side on a host, the simple switchover approach of removing a container and starting a new one can be performed in as little as a few seconds (and it permits a similarly fast rollback).

For most applications, this may well be fast enough, but applications with a long startup time or high availability requirements need an alternative approach. Sometimes this is an unavoidably complex process requiring special handling with the application itself, but web-facing applications have an option you may wish to consider first.

PROBLEM

You need to be able to upgrade web-facing applications with zero downtime.

SOLUTION

Use confd with nginx on your host to perform a two-stage switchover.

Nginx is an extremely popular web server with a crucial built-in ability—it can reload configuration files without dropping connections to the server. By combining this with confd, a tool that can retrieve information from a central datastore (like etcd) and alter configuration files accordingly, you can update etcd with the latest settings and watch everything else be handled for you.

> **NOTE** The Apache HTTP server and HAProxy both also offer zero-downtime reloading and can be used instead of nginx if you have existing configuration expertise.

The first step is to start an application that will serve as an old application that you'll eventually update. Python comes with Ubuntu and has a built-in web server, so we'll use it as an example.

Listing 9.13 Starting a simple fileserver in a container

```
$ ip addr | grep 'inet ' | grep -v 'lo$\|docker0$'
    inet 10.194.12.221/20 brd 10.194.15.255 scope global eth0
$ HTTPIP=http://10.194.12.221
$ docker run -d --name py1 -p 80 ubuntu:14.04.2 \
  sh -c 'cd / && python3 -m http.server 80'
e6b769ec3efa563a959ce771164de8337140d910de67e1df54d4960fdff74544
$ docker inspect -f '{{.NetworkSettings.Ports}}' py1
map[80/tcp:[{0.0.0.0 32768}]]
$ curl -s localhost:32768 | tail | head -n 5
<li><a href="sbin/">sbin/</a></li>
<li><a href="srv/">srv/</a></li>
<li><a href="sys/">sys/</a></li>
<li><a href="tmp/">tmp/</a></li>
<li><a href="usr/">usr/</a></li>
```

The HTTP server has started successfully, and we used the filter option of the `inspect` command to pull out information about what port on the host is mapped to point inside the container.

Now make sure you have etcd running—this technique assumes you're still in the same working environment as the previous technique. This time you're going to use etcdctl (short for "etcd controller") to interact with etcd (rather than `curl`ing etcd directly) for simplicity.

Listing 9.14 Download and use the etcdctl Docker image

```
$ IMG=dockerinpractice/etcdctl
$ docker pull dockerinpractice/etcdctl
[...]
$ alias etcdctl="docker run --rm $IMG -C \"$HTTPIP:8080\""
$ etcdctl set /test value
value
$ etcdctl ls
/test
```

This has downloaded an etcdctl Docker image that we prepared, and it has set up an alias to always connect the etcd cluster set up previously. Now start up nginx.

Listing 9.15 Start an nginx + confd container

```
$ IMG=dockerinpractice/confd-nginx
$ docker pull $IMG
[...]
$ docker run -d --name nginx -p 8000:80 $IMG $HTTPIP:8080
ebdf3faa1979f729327fa3e00d2c8158b35a49acdc4f764f0492032fa5241b29
```

This is an image we prepared earlier, which uses confd to retrieve information from etcd and automatically update configuration files. The parameter that we pass tells the container where it can connect to the etcd cluster. Unfortunately we haven't told it where it can find our apps yet, so the logs are filled with errors.

Let's add the appropriate information to etcd.

Listing 9.16 Demonstrating the auto-configuration of the nginx container

```
$ docker logs nginx
Using http://10.194.12.221:8080 as backend
2015-05-18T13:09:56Z ebdf3faa1979 confd[14]: >
ERROR 100: Key not found (/app) [14]
2015-05-18T13:10:06Z ebdf3faa1979 confd[14]: >
ERROR 100: Key not found (/app) [14]
$ echo $HTTPIP
http://10.194.12.221
$ etcdctl set /app/upstream/py1 10.194.12.221:32768
10.194.12.221:32768
$ sleep 10
$ docker logs nginx
Using http://10.194.12.221:8080 as backend
2015-05-18T13:09:56Z ebdf3faa1979 confd[14]: >
ERROR 100: Key not found (/app) [14]
2015-05-18T13:10:06Z ebdf3faa1979 confd[14]: >
ERROR 100: Key not found (/app) [14]
2015-05-18T13:10:16Z ebdf3faa1979 confd[14]: >
ERROR 100: Key not found (/app) [14]
2015-05-18T13:10:26Z ebdf3faa1979 confd[14]: >
INFO Target config /etc/nginx/conf.d/app.conf out of sync
2015-05-18T13:10:26Z ebdf3faa1979 confd[14]: >
INFO Target config /etc/nginx/conf.d/app.conf has been updated
$ curl -s localhost:8000 | tail | head -n5
<li><a href="sbin/">sbin/</a></li>
<li><a href="srv/">srv/</a></li>
<li><a href="sys/">sys/</a></li>
<li><a href="tmp/">tmp/</a></li>
<li><a href="usr/">usr/</a></li>
```

The update to etcd has been read by confd and applied to the nginx configuration file, allowing you to visit your simple file server. The `sleep` command is included because confd has been configured to check for updates every 10 seconds. Behind the scenes, a confd daemon running in the confd-nginx container polls for changes in the etcd cluster, using a template within the container to regenerate the nginx configuration only when changes are detected.

Let's say we've decided we want to serve /etc rather than /. We'll now start up our second application and add it to etcd. Because we'll then have two backends, we'll end up getting responses from each of them.

Listing 9.17 Using confd to set up two backend web services for nginx

```
$ docker run -d --name py2 -p 80 ubuntu:14.04.2 \
  sh -c 'cd /etc && python3 -m http.server 80'
9b5355b9b188427abaf367a51a88c1afa2186e6179ab46830715a20eacc33660
$ docker inspect -f '{{.NetworkSettings.Ports}}' py2
map[80/tcp:[{0.0.0.0 32769}]]
$ curl -s $HTTPIP:32769 | tail | head -n 5
<li><a href="udev/">udev/</a></li>
<li><a href="update-motd.d/">update-motd.d/</a></li>
<li><a href="upstart-xsessions">upstart-xsessions</a></li>
<li><a href="vim/">vim/</a></li>
<li><a href="vtrgb">vtrgb@</a></li>
$ echo $HTTPIP
http://10.194.12.221
$ etcdctl set /app/upstream/py2 10.194.12.221:32769
10.194.12.221:32769
$ etcdctl ls /app/upstream
/app/upstream/py1
/app/upstream/py2
$ curl -s localhost:8000 | tail | head -n 5
<li><a href="sbin/">sbin/</a></li>
<li><a href="srv/">srv/</a></li>
<li><a href="sys/">sys/</a></li>
<li><a href="tmp/">tmp/</a></li>
<li><a href="usr/">usr/</a></li>
$ curl -s localhost:8000 | tail | head -n 5
<li><a href="udev/">udev/</a></li>
<li><a href="update-motd.d/">update-motd.d/</a></li>
<li><a href="upstart-xsessions">upstart-xsessions</a></li>
<li><a href="vim/">vim/</a></li>
<li><a href="vtrgb">vtrgb@</a></li>
```

In the preceding process, we checked that the new container came up correctly before adding it to etcd (see figure 9.4). We could have performed the process in one step by overwriting the `/app/upstream/py1` key in etcd—this is also useful if you need only one backend to be accessible at a time.

With the two-stage switchover, the second stage is to remove the old backend and container.

Listing 9.18 Removing the old upstream address

```
$ etcdctl rm /app/upstream/py1
PrevNode.Value: 192.168.1.123:32768
$ etcdctl ls /app/upstream
/app/upstream/py2
$ docker rm -f py1
py1
```

The new application is up and running by itself! At no point has the application been inaccessible to users, and there has been no need to manually connect to web server machines to reload nginx.

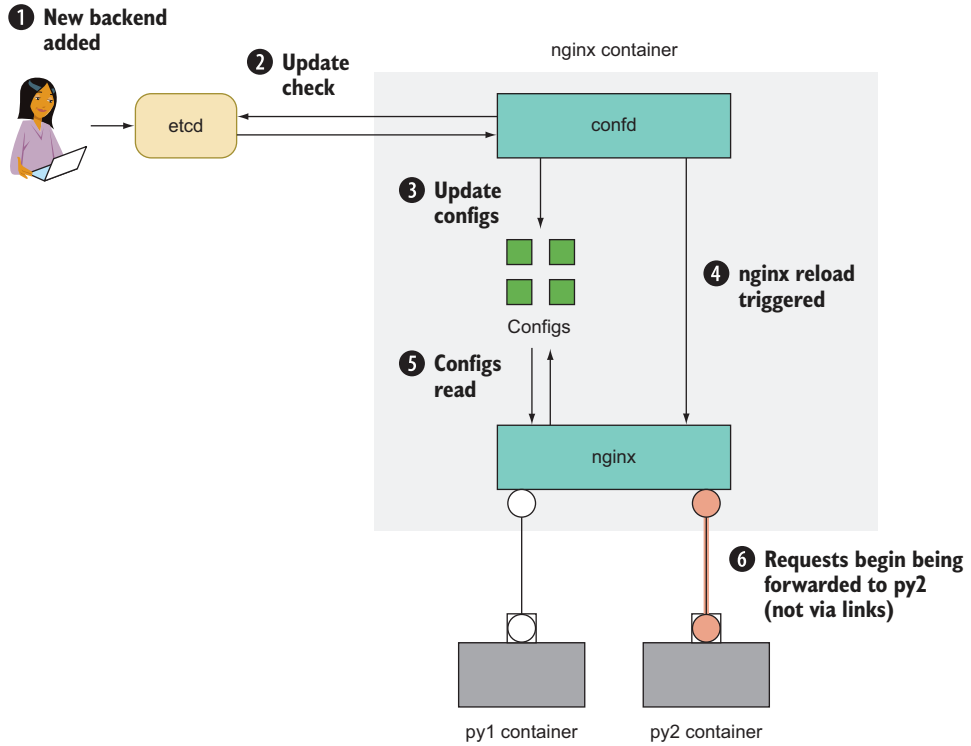

Figure 9.4 Adding the py2 container to etcd

DISCUSSION

The uses of confd extend to more than configuring web servers: if you have any file containing text that needs updating based on external values, confd is there to step in—a useful connector between configuration files stored on disk and a single-point-of-truth etcd cluster.

As noted in the previous technique, etcd is not designed for storing large values. There's also no reason you must use etcd with confd—there are a number of integrations available for the most popular key/value stores, so you might not need to add another moving part if you've already got something that works for you.

Later on, in technique 86, when we look at using Docker in production, you'll see a method that avoids having to manually alter etcd at all if you want to update the backend servers for a service.

Summary

- Docker provides a great basis for a contract between development and operations teams.
- Moving images between registries can be a good way to control how far builds progress through your CD pipeline.

- Bup is good at squeezing image transfers even more than layers can.
- Docker images can be moved and shared as TAR files.
- etcd can act as a central configuration store for an environment.
- Zero-downtime deployment can be achieved by combining etcd, confd, and nginx.