Docker

IN PRACTICE

Ian Miell
Aidan Hobson Sayers

FOREWORD BY Ben Firshman

MANNING

SAMPLE CHAPTER

Docker in Practice

by Ian Miell
and Aidan Hobson Sayers

Chapter 6

brief contents

6 Continuous integration: speeding up your development pipeline

This chapter covers

- Using the Docker Hub workflow as a CI tool
- Speeding up your IO-heavy builds
- Using Selenium for automated testing
- Running Jenkins within Docker
- Using Docker as a Jenkins slave
- Scaling your available compute with your dev team

In this chapter we're going to look at various techniques that will use Docker to enable and improve your CI efforts.

By now you should understand how Docker is well suited to being used for automation. Its lightweight nature and the power it gives you to port environments from one place to another can make it a key enabler of continuous integration (CI). We've found the techniques in this chapter to be invaluable in making a CI process feasible within a business.

Making your build environment stable and reproducible, using testing tools requiring significant setup, and expanding your build capacity are all problems you may face, and Docker can help.

> **CONTINUOUS INTEGRATION** In case you don't know, *continuous integration* is a software lifecycle strategy used to speed up the development pipeline. By automatically rerunning tests every time a significant change is made to the codebase, you get faster and more stable deliveries because there's a base level of stability in the software being delivered.

6.1 Docker Hub automated builds

The Docker Hub automated build feature was mentioned in technique 9, though we didn't go into any detail on it. In short, if you point to a Git repository containing a Dockerfile, the Docker Hub will handle the process of building the image and making it available to download. An image rebuild will be triggered on any changes in the Git repository, making this quite useful as part of a CI process.

TECHNIQUE 55 Using the Docker Hub workflow

This technique introduces you to the Docker Hub workflow, which enables you to trigger rebuilds of your images.

> **DOCKER.COM ACCOUNT REQUIRED** For this section you'll need an account on docker.com linked to either a GitHub or a Bitbucket account. If you don't already have these set up and linked, instructions are available from the homepages of github.com and bitbucket.org.

PROBLEM

You want to automatically test and push changes to your image when the code changes.

SOLUTION

Set up a Docker Hub repository and link it to your code.

DISCUSSION

Although the Docker Hub build isn't complicated, there are a number of steps required, so we've broken them up into bite-sized chunks in table 6.1, which serves as an overview of the process.

Table 6.1 Setting up a linked Docker Hub repository

Number	Step
1	Create your repository on GitHub or Bitbucket
2	Clone the new Git repository
3	Add code to your Git repository
4	Commit the source
5	Push the Git repository

Table 6.1 **Setting up a linked Docker Hub repository *(continued)***

Number	Step
6	Create a new repository on the Docker Hub
7	Link the Docker Hub repository to the Git repository
8	Wait for the Docker Hub build to complete
9	Commit and push a change to the source
10	Wait for the second Docker Hub build to complete

GIT AND DOCKER REPOSITORIES Both Git and Docker use the term *repository* to refer to a project. This can confuse people. A Git repository and a Docker repository are not the same thing, even though here we're linking the two types of repositories.

1. CREATE YOUR REPOSITORY ON GITHUB OR BITBUCKET

Create a new repository on GitHub or Bitbucket. You can give it any name you want.

2. CLONE THE NEW GIT REPOSITORY

Clone your new Git repository to your host machine. The command for this will be available from the Git project's homepage.

Change directory into this repository.

3. ADD CODE TO YOUR GIT REPOSITORY

Now you need to add code to the project.

You can add any Dockerfile you like, but the following listing shows an example known to work. It consists of two files representing a simple dev tools environment. It installs some preferred utilities and outputs the bash version you have.

Listing 6.1 Dockerfile—simple dev tools container

```
FROM ubuntu:14.04
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update
RUN apt-get install -y curl
RUN apt-get install -y nmap
RUN apt-get install -y socat
RUN apt-get install -y openssh-client
RUN apt-get install -y openssl
RUN apt-get install -y iotop
RUN apt-get install -y strace
RUN apt-get install -y tcpdump
RUN apt-get install -y lsof
RUN apt-get install -y inotify-tools
RUN apt-get install -y sysstat
RUN apt-get install -y build-essential
RUN echo "source /root/bash_extra" >> /root/.bashrc
ADD bash_extra /root/bash_extra
CMD ["/bin/bash"]
```

Install useful packages.

Add a line to the root's bashrc to source bash_extra.

Add bash_extra from the source to the container.

Now you need to add the bash_extra file you referenced and give it the content shown in the next listing.

Listing 6.2 bash_extra—extra bash commands

```
bash --version
```

4. COMMIT THE SOURCE

To commit your source code source, use this command:

```
git commit -am "Initial commit"
```

5. PUSH THE GIT REPOSITORY

Now you can push the source to the Git server with this command:

```
git push origin master
```

6. CREATE A NEW REPOSITORY ON THE DOCKER HUB

Next you need to create a repository for this project on the Docker Hub. Go to https://hub.docker.com and ensure you're logged in. Then click on Add Repository and choose Automated Build.

7. LINK THE DOCKER HUB REPOSITORY TO THE GIT REPOSITORY

You'll see a screen with a choice of Git services. Pick the source code service you use (GitHub or Bitbucket) and select your new repository from the provided list. (If this step doesn't work for you, you may need to set up the link between your Docker Hub account and the Git service.)

You'll see a page with options for the build configuration. You can leave the defaults and click Create Repository at the bottom.

8. WAIT FOR THE DOCKER HUB BUILD TO COMPLETE

You'll see a page with a message explaining that the link worked. Click on the Build Details link.

Next, you'll see a page that shows the details of the builds. Under Builds History there will be an entry for this first build. If you don't see anything listed, you may need to press the button to trigger the build manually. The Status field next to the build ID will show Pending, Finished, Building, or Error. If all is well, you'll see one of the first three. If you see Error, then something has gone wrong and you'll need to click on the build ID to see what the error was.

> **BUILDING CAN TAKE TIME** It can take a while for the build to start, so seeing Pending for some time while waiting is perfectly normal.

Click Refresh periodically until you see that the build has completed. Once it's complete, you can pull the image with the `docker pull` command listed on the top of the same page.

9. COMMIT AND PUSH A CHANGE TO THE SOURCE

Now you decide that you want more information about your environment when you log in, so you want to output the details of the distribution you're running in. To achieve this, add these lines to your bash_extra file so that it now looks like this:

```
bash --version
cat /etc/issue
```

Then commit and push as in steps 4 and 5.

10. WAIT FOR THE (SECOND) DOCKER HUB BUILD TO COMPLETE

If you return to the build page, a new line should show up under the Builds History section, and you can follow this build as in step 8.

> **EMAIL RECEIVED ON ERROR ONLY** You'll be emailed if there's an error with your build (no email if all is OK), so once you're used to this workflow, you only need to check up on it if you receive an email.

You can now use the Docker Hub workflow! You'll quickly get used to this framework and find it invaluable for keeping your builds up to date and reducing the cognitive load of rebuilding Dockerfiles by hand.

6.2 More efficient builds

CI implies a more frequent rebuilding of your software and tests. Although Docker makes delivering CI easier, the next problem you may bump into is the resulting increased load on your compute resources.

We'll look at ways to alleviate this pressure in terms of disk I/O, network bandwidth, and automated testing.

TECHNIQUE 56 Speed up I/O-intensive builds with eatmydata

Because Docker is a great fit for automated building, you'll likely perform a lot of disk-I/O-intensive builds as time goes on. Jenkins jobs, database rebuild scripts, and large code checkouts will all hit your disks hard. In these cases, you'll be grateful for any speed increases you can get, both to save time and to minimize the many overheads that result from resource contention.

This technique has been shown to give up to a 1/3 speed increase, and our experience backs this up. This is not to be sniffed at!

PROBLEM

You want to speed up your I/O-intensive builds.

SOLUTION

Install eatmydata on your image.

DISCUSSION

eatmydata is a program that takes your system calls to write data and makes them super-fast by bypassing work required to persist those changes. This entails some lack of safety, so it's not recommended for normal use, but it's quite useful for environments not designed to persist, such as in testing.

INSTALLATION

To install eatmydata, you have a number of options.

If you're running a deb-based distribution, you can `apt-get install` it.

If you're running an rpm-based distribution, you'll be able to `rpm --install` it by searching for it on the web and downloading it. Websites such as rpmfind.net are a good place to start.

As a last resort, and if you have a compiler installed, you can download and compile it directly as shown in the next listing.

Listing 6.3 Compile and install eatmydata

```
$ url=https://www.flamingspork.com/projects/libeatmydata/
➥ libeatmydata-105.tar.gz
$ wget -qO- $url | tar -zxf -
$ ./configure --prefix=/usr
$ make
$ sudo make install
```

If this version doesn't download, check on the website to see whether it's been updated to a number later than 105.

Flamingspork.com is the website of the maintainer.

Change the prefix directory if you want the eatmydata executable to be installed somewhere other than /usr/bin.

Build the eatmydata executable.

Install the software; this step requires root privileges.

USING EATMYDATA

Once libeatmydata is installed on your image (either from a package or from source), run the eatmydata wrapper script before any command to take advantage of it:

```
docker run -d mybuildautomation eatmydata /run_tests.sh
```

Figure 6.1 shows at a high level how eatmydata saves you processing time.

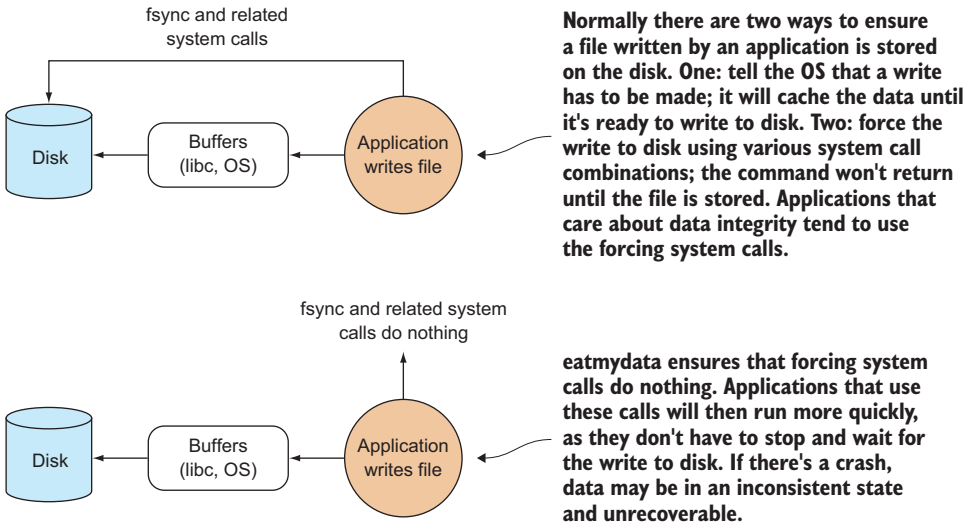


Figure 6.1 Application writes to disk without and with eatmydata

USE WITH CAUTION! eatmydata skips the steps to guarantee that data is safely written to disk, so there's a risk that data will not yet be on disk when the program thinks it is. For test runs, this usually doesn't matter, because the data is disposable, but don't use eatmydata to speed up any kind of environment where the data matters!

TECHNIQUE 57 Set up a package cache for faster builds

As Docker lends itself to frequent rebuilding of services for development, testing, and production, you can quickly get to a point where you're repeatedly hitting the network a lot. One major cause is downloading package files from the internet. This can be a slow (and costly) overhead, even on a single machine. This technique shows you how to set up a local cache for your package downloads, covering apt and yum.

PROBLEM

You want to speed up your builds by reducing network I/O.

SOLUTION

Install a Squid proxy for your package manager.

DISCUSSION

Figure 6.2 illustrates how this technique works in the abstract.

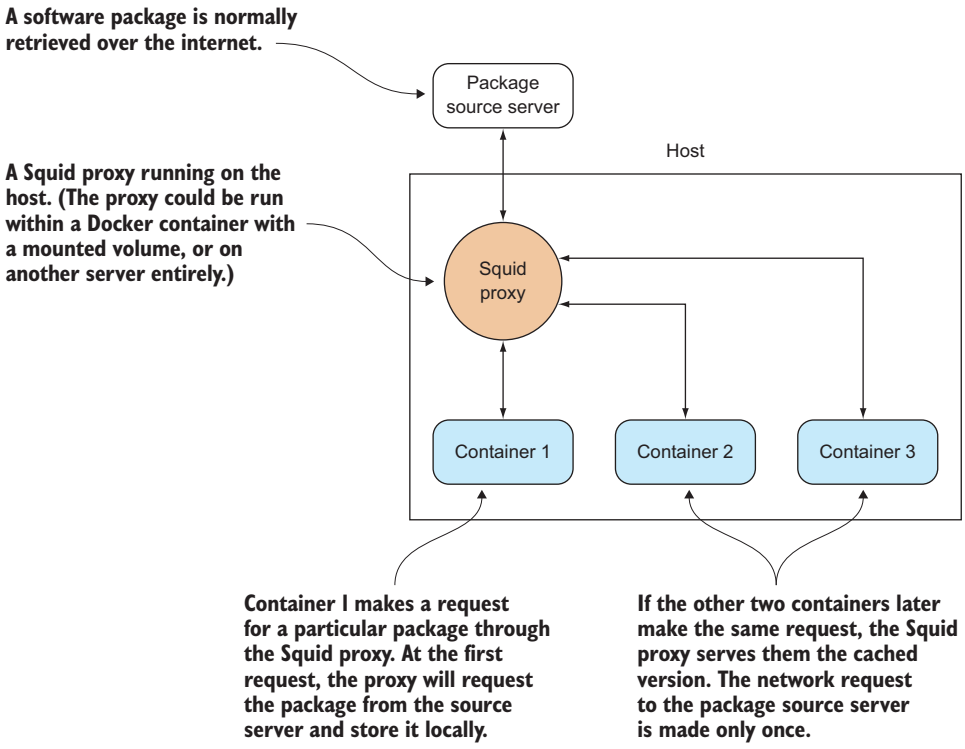


Figure 6.2 Using a Squid proxy to cache packages

Because the calls for packages go to the local Squid proxy first, and are only requested over the internet the first time, there should only be one request over the internet for each package. If you have hundreds of containers all pulling down the same large packages from the internet, this can save you a lot of time and money.

> **NETWORK SETUP!** You may have network configuration issues when setting this up on your host. Advice is given in the following sections to determine whether this is the case, but if you're unsure how to proceed, you may need to seek help from a friendly network admin.

DEBIAN

For Debian (otherwise known as apt or .deb) packages, the setup is simpler because there is a prepackaged version.

On your Debian-based host run this command:

```
sudo apt-get install squid-deb-proxy
```

Ensure that the service is started by telneting to port 8000:

```
$ telnet localhost 8000
Trying ::1...
Connected to localhost.
Escape character is '^]'.
```

Press Ctrl-] followed by Ctrl-d to quit if you see the preceding output. If you don't see this output, then Squid has either not installed properly, or it has installed on a non-standard port.

To set up your container to use this proxy, we've provided the following example Dockerfile. Bear in mind that the IP address of the host from the point of view of the container may change from run to run. You may want to convert this Dockerfile to a script to be run from within the container before installing new software:

Ensure the route tool is installed.

Port 8000 is used to connect to the Squid proxy on the host machine.

In order to determine the host's IP address from the point of view of the container, run the route command and use awk to extract the relevant IP address from the output (see technique 59).

```
FROM debian
RUN apt-get update -y && apt-get install net-tools
RUN echo "Acquire::http::Proxy \"http://$( \
route -n | awk '/^0.0.0.0/ {print $2}' \
):8000\";" \
> /etc/apt/apt.conf.d/30proxy
RUN echo "Acquire::http::Proxy::ppa.launchpad.net  DIRECT;" >> \
    /etc/apt/apt.conf.d/30proxy
CMD ["/bin/bash"]
```

The echoed lines with the appropriate IP address and configuration are added to apt's proxy configuration file.

YUM

On the host, ensure Squid is installed by installing the "squid" package with your package manager.

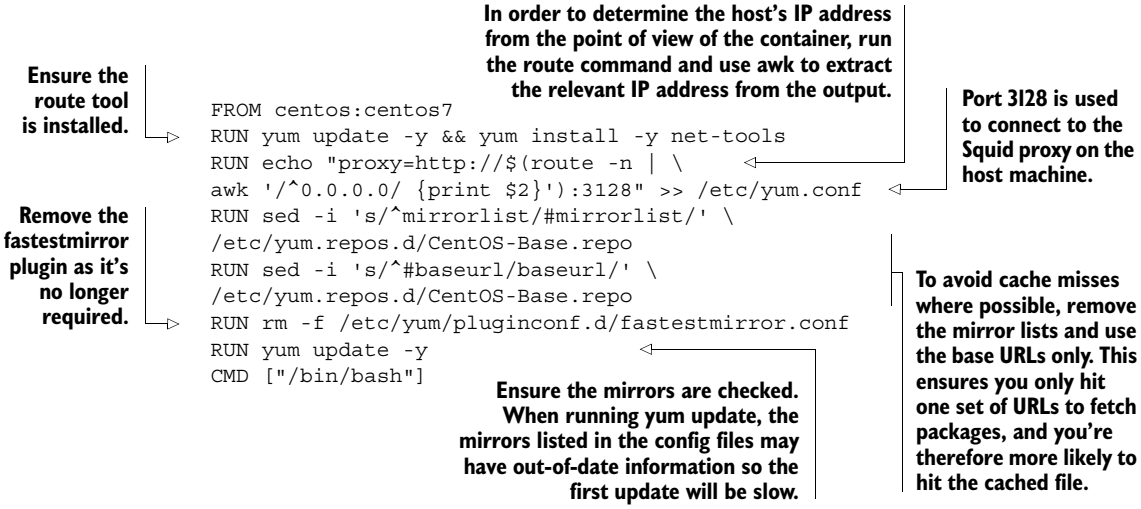
Then you need to change the Squid configuration to create a larger cache space. Open up the /etc/squid/squid.conf file and replace the commented line beginning with `#cache_dir ufs /var/spool/squid` with this: `cache_dir ufs /var/spool/squid 10000 16 256`. This creates a space of 10,000 MB, which should be sufficient.

Ensure the service is started by telneting to port 3128:

```
$ telnet localhost 3128
Trying ::1...
Connected to localhost.
Escape character is '^]'.
```

Press Ctrl-] followed by Ctrl-d to quit if you see the preceding output. If you don't see this output, then Squid has either not installed properly, or has installed on a nonstandard port.

To set up your container to use this proxy, we've provided the following example Dockerfile. Bear in mind that the IP address of the host from the point of view of the container may change from run to run. You may want to convert this Dockerfile to a script to be run from within the container before installing new software:



If you set up two containers this way and install the same large package on both, one after the other, you should notice that the second installation downloads its prerequisites much quicker than the first.

DOCKERIZE THE SQUID PROXY You may have observed that you can run the Squid proxy on a container rather than on the host. That option wasn't shown here to keep the explanation simple (in some cases, more steps are required to make Squid work within a container). You can read more about this, along with how to make containers automatically use the proxy, at https://github.com/jpetazzo/squid-in-a-can.

TECHNIQUE 58 Running Selenium tests inside Docker

One Docker use case we haven't yet examined in much detail is running graphical applications. In chapter 3, VNC was used to connect to containers during the "save game" approach to development (technique 14), but this can be clunky—windows are contained inside the VNC viewer window, and desktop interaction can be a little limited. We'll explore an alternative to this by demonstrating how to write graphical tests using Selenium and also show you how this image can be used to run the tests as part of your CI workflow.

PROBLEM

You want to be able to run graphical programs in your CI process while having the option to display those same graphical programs on your own screen.

SOLUTION

Share your X11 server socket to view the programs on your own screen, and use xvfb in your CI process.

DISCUSSION

No matter what other things you need to do to start your container, you must have the Unix socket that X11 uses to display your windows mounted as a volume inside the container, and you need to indicate which display your windows should be shown on. You can double-check whether these two things are set to their defaults by running the following commands:

```
~ $ ls /tmp/.X11-unix/
X0
~ $ echo $DISPLAY
:0
```

The first command checks that the X11 server Unix socket is running in the location assumed for the rest of the technique. The second command checks the environment variable applications used to find the X11 socket. If your output for these commands doesn't match the output here, you may need to alter some arguments to the commands in this technique.

Now that you've checked your machine setup, you want to get applications running inside a container to be seamlessly displayed outside the container. The main problem you need to overcome is the security that your computer puts in place to prevent other people from connecting to your machine, taking over your display, and potentially recording your keystrokes. In technique 26 you briefly saw how to do this, but we didn't talk about how it worked or look at any alternatives.

X11 has multiple ways of authenticating a container to use your X socket. First we'll look at the .Xauthority file—it should be present in your home directory. It contains hostnames along with the "secret cookie" each host must use to connect. By giving your Docker container the same hostname as your machine, you can use your existing X authority file:

```
$ ls $HOME/.Xauthority
/home/myuser/.Xauthority
$ docker run -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix \
    --hostname=$HOSTNAME -v $HOME/.Xauthority:/root/.Xauthority \
    -it ubuntu:14.04 bash
```

The second method of allowing Docker to access the socket is a much blunter instrument and it has security issues, as it disables all the protection X gives you. If nobody has access to your computer, this may be an acceptable solution, but you should always try to use the X authority file first. You can secure yourself again after you try the following steps by running `xhost -` (though this will lock out your Docker container):

```
$ xhost +
access control disabled, clients can connect from any host
$ docker run -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix \
    -it ubuntu:14.04 bash
```

The first line disables all access control to X, and the second runs the container. Note that you don't have to set the hostname or mount anything apart from the X socket.

Once you've started up your container, it's time to check that it works. You can do this by running the following commands:

```
root@ef351febcee4:/# apt-get update && apt-get install -y x11-apps
[...]
root@ef351febcee4:/# xeyes
```

This will start up a classic application that tests whether X is working—xeyes. You should see the eyes follow your cursor as you move it around the screen. Note that (unlike VNC) the application is integrated into your desktop—if you were to start xeyes multiple times, you'd see multiple windows.

It's time to get started with Selenium. If you've never used it before, it's a tool with the ability to automate browser actions and is commonly used to test website code—it needs a graphical display for the browser to run in. Although it's most commonly used with Java, we're going to use Python to allow more interactivity:

```
root@ef351febcee4:/# apt-get install -y python2.7 python-pip firefox
[...]
root@ef351febcee4:/# pip install selenium
Downloading/unpacking selenium==2.47.3
[...]
Successfully installed selenium==2.47.3
Cleaning up...
root@ef351febcee4:/# python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from selenium import webdriver
>>> b = webdriver.Firefox()
```

As you may have noticed, Firefox has launched and appeared on your screen! All the preceding code does is install Python, Firefox, and a Python package manager. It then uses the Python package manager to install the Selenium Python package.

You can now experiment with Selenium. An example session running against GitHub follows—you'll need a basic understanding of CSS selectors to understand what's going on here. Note that websites frequently change, so this particular snippet may need modifying to work correctly:

```
>>> b.get('http://github.com')
>>> searchselector = '.js-site-search-form input[type="text"]'
>>> searchbox = b.find_element_by_css_selector(searchselector)
>>> searchbox.send_keys('docker-in-practice\n')
>>> usersxpath = '//nav//a[contains(text(), "Users")]'
>>> userslink = b.find_element_by_xpath(usersxpath)
>>> userslink.click()
>>> dlinkselector = '.user-list-info a'
>>> dlink = b.find_elements_by_css_selector(dlinkselector)[0]
>>> dlink.click()
>>> mlinkselector = '.org-header a.meta-link'
>>> mlink = b.find_element_by_css_selector(mlinkselector)
>>> mlink.click()
```

The details here aren't important. Just note that we're writing commands in Python in our container and seeing them take effect in the Firefox window running inside the container, but visible on the desktop.

This is great for debugging tests you write, but how would you integrate them into a CI pipeline with the same Docker image? A CI server typically doesn't have a graphical display, so you need to make this work without mounting your own X server socket, but Firefox still needs an X server to run on. There's a useful tool created for situations like this called xvfb, which pretends to have an X server running for applications to use, but doesn't require a monitor.

To see how this works, let's install xvfb, commit the container, tag it as `selenium`, and create a test script:

```
>>> exit()
root@ef351febcee4:/# apt-get install -y xvfb
[...]
root@ef351febcee4:/# exit
$ docker commit ef351febcee4 selenium
d1cbfbc76790cae5f4ae95805a8ca4fc4cd1353c72d7a90b90ccfb79de4f2f9b
$ cat > myscript.py << EOF
from selenium import webdriver
b = webdriver.Firefox()
print 'Visiting github'
b.get('http://github.com')
print 'Performing search'
searchselector = '.js-site-search-form input[type="text"]'
searchbox = b.find_element_by_css_selector(searchselector)
searchbox.send_keys('docker-in-practice\n')
```

```
print 'Switching to user search'
usersxpath = '//nav//a[contains(text(), "Users")]'
userslink = b.find_element_by_xpath(usersxpath)
userslink.click()
print 'Opening docker in practice user page'
dlinkselector = '.user-list-info a'
dlink = b.find_elements_by_css_selector(dlinkselector)[99]
dlink.click()
print 'Visiting docker in practice site'
mlinkselector = '.org-header a.meta-link'
mlink = b.find_element_by_css_selector(mlinkselector)
mlink.click()
print 'Done!'
EOF
```

Note the subtle difference in the assignment of the dlink variable. By attempting to get the hundredth result containing the text "Docker in Practice," you'll trigger an error, which will cause the Docker container to exit with a non-zero status and trigger failures in the CI pipeline.

Time to try it out:

```
$ docker run --rm -v $(pwd):/mnt selenium sh -c \
"xvfb-run -s '-screen 0 1024x768x24 -extension RANDR'\
python /mnt/myscript.py"
Visiting github
Performing search
Switching to user search
Opening docker in practice user page
Traceback (most recent call last):
  File "myscript.py", line 15, in <module>
      dlink = b.find_elements_by_css_selector(dlinkselector)[99]
      IndexError: list index out of range
$ echo $?
1
```

You've run a self-removing container that executes the Python test script running under a virtual X server. As expected, it failed and returned a non-zero exit code.

> **CMD VS. ENTRYPOINT** The `sh -c "command string here"` is an unfortunate result of how Docker treats `CMD` values by default. If you were to put this in a Dockerfile, you'd be able to remove the `sh -c` and make `xvfb-run` the entrypoint, allowing you to run whatever test scripts you'd like.

As has been demonstrated, Docker is a flexible tool and can be put to some initially surprising uses (graphical apps in this case). Some people run *all* of their graphical apps inside Docker, including games! We wouldn't go that far, but we've found that re-examining assumptions about Docker can lead to some surprising use cases.

6.3 Containerizing your CI process

Once you have a consistent development process across teams, it's important to also have a consistent build process. Randomly failing builds defeat the point of Docker.

As a result, it makes sense to *containerize* your entire CI process. This not only makes sure your builds are repeatable, it allows you to move your CI process anywhere without fear of leaving some vital piece of configuration behind (likely discovered with much frustration later).

In these techniques, we'll use Jenkins (as this is the most widely used CI tool), but the same techniques should apply to other CI tools. We don't assume a great deal of familiarity with Jenkins here, but we won't cover setting up standard tests and builds. That information is not essential to the techniques here.

TECHNIQUE 59 Containing a complex development environment

Docker's portability and lightweight nature make it an obvious choice for a CI slave (a machine the CI master connects to in order to carry out builds). A Docker CI slave is a step change from a VM slave (and is even more of a leap from bare-metal build machines). It allows you to perform builds on a multitude of environments with a single host, to quickly tear down and bring up clean environments to ensure uncontaminated builds, and to use all your familiar Docker tooling to manage your build environments.

Being able to treat the CI slave as just another Docker container is particularly interesting. Have mysterious build failures on one of your Docker CI slaves? Pull the image and try the build yourself.

PROBLEM

You want to scale and modify your Jenkins slave.

SOLUTION

Encapsulate the configuration of your slave in a Docker image, and deploy.

DISCUSSION

Many organizations set up a heavyweight Jenkins slave (often on the same host as the server), maintained by a central IT function, that serves a useful purpose for a time. As time goes on, and teams grow their codebases and diverge, requirements grow for more and more software to be installed, updated, or altered so that the jobs will run.

Figure 6.3 shows a simplified version of this scenario. Imagine hundreds of software packages and multiple new requests all giving an overworked infrastructure team headaches.

> **ILLUSTRATIVE, NON-PORTABLE EXAMPLE** This technique has been constructed to show you the essentials of running a Jenkins slave in a container. This makes the result less portable but the lesson easier to grasp. Once you understand all the techniques in this chapter, you'll be able to make a more portable setup.

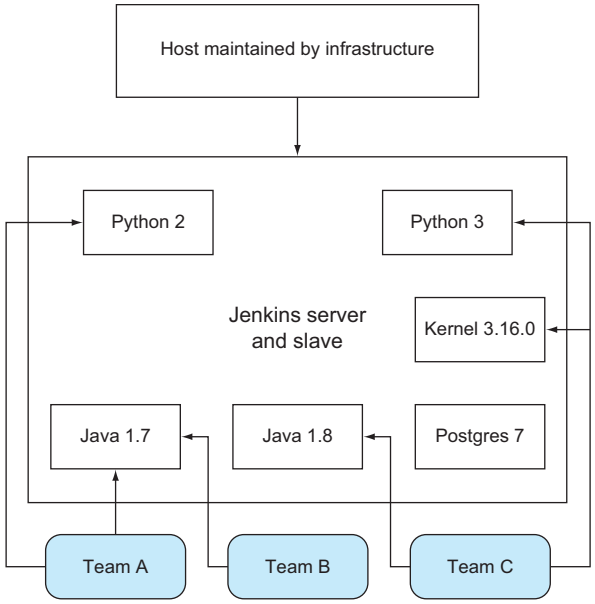


Figure 6.3 An overloaded Jenkins server

Stalemate has been known to ensue, because sysadmins may be reluctant to update their configuration management scripts for one group of people as they fear breaking another's build, and teams get increasingly frustrated over the slowness of change.

Docker (naturally) offers a solution by allowing multiple teams to use a base image for their own personal Jenkins slave, while using the same hardware as before. You can create an image with the required shared tooling on it, and allow teams to alter it to meet their own needs.

Some contributors have uploaded their own reference slaves on the Docker Hub; you can find them by searching for "jenkins slave" on the Docker Hub. The following listing is a minimal Jenkins slave Dockerfile.

Listing 6.4 Bare-bones Jenkins slave Dockerfile

```
FROM ubuntu
ENV DEBIAN_FRONTEND noninteractive
RUN groupadd -g 1000 jenkins_slave
RUN useradd -d /home/jenkins_slave -s /bin/bash \
-m jenkins_slave -u 1000 -g jenkins_slave
RUN echo jenkins_slave:jpass | chpasswd
RUN apt-get update && \
apt-get install -y openssh-server openjdk-7-jre wget
RUN mkdir -p /var/run/sshd
CMD ip route | grep "default via" \
| awk '{print $3}' && /usr/sbin/sshd -D
```

Create the Jenkins slave user and group.

Set the Jenkins user password to jpass. In a more sophisticated setup, you'd likely want to use other authentication methods.

Install the required software to function as a Jenkins slave.

On startup, output the IP address of the host machine from the point of view of the container, and start the SSH server.

Build the slave image, tagging it as `jenkins_slave`:

```
$ docker build -t jenkins_slave .
```

Run it with this command:

```
$ docker run --name jenkins_slave -ti -p 2222:22 jenkins_slave
172.17.42.1
```

Jenkins server needs to be running

If you don't have a Jenkins server already running on your host, ensure you have the Jenkins server running as in the previous technique. If you're in a hurry, run this command:

```
$ docker run --name jenkins_server -p 8080:8080 -p 50000:50000 \
dockerinpractice/jenkins:server
```

This will make the Jenkins server available at http://localhost:8080 if you've run it on your local machine.

If you navigate to the Jenkins server, you'll be greeted with the page in figure 6.4.

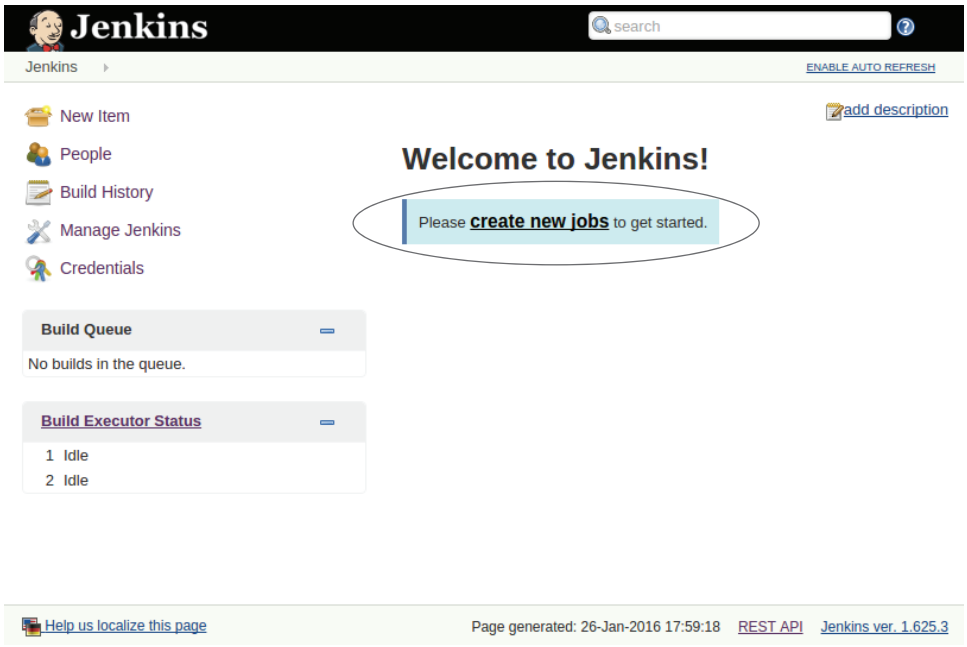


Figure 6.4 Jenkins' welcome page

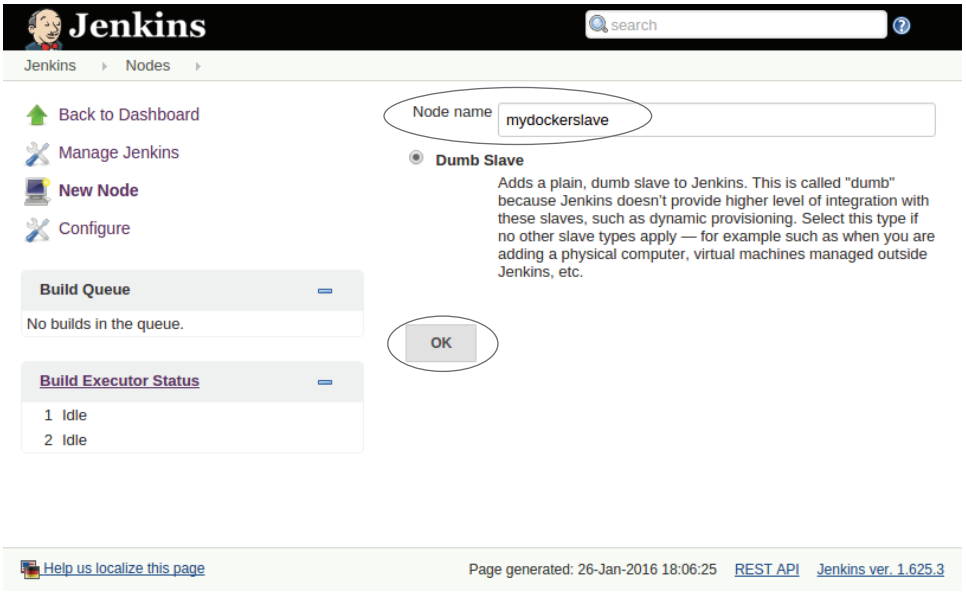


Figure 6.5 Naming a new node

You can add a slave by clicking on Build Executor Status > New Node and adding the node name as a dumb slave, as shown in figure 6.5. Call it `mydockerslave`.

Click OK and configure it with these settings, as shown in figure 6.6:

- Set Remote Root Directory to /home/jenkins_slave.
- Click Advanced to expose the port field, and set it to 2222.
- Click Add to add credentials, and set the username to `jenkins_slave` and the password to `jpass`.
- Make sure the Launch Slave Agents on Unix Machines Via SSH option is selected.
- Set the host to the route IP seen from within the container (output with the `docker run` earlier).
- Give it a Label of `dockerslave`.
- Click Save.

Now click on Launch Slave Agent (assuming this doesn't happen automatically) and you should see that the slave agent is now marked as online.

Go back to the home page by clicking on Jenkins in the top left, and click on New Item. Create a Freestyle Project called `test`, and under the Build section click on Add Build Step > Execute Shell, with the command `echo done`. Scroll up and select Restrict Where Project Can Be Run and enter the Label Expression `dockerslave`. You should see that Slaves In Label is set as `1`.

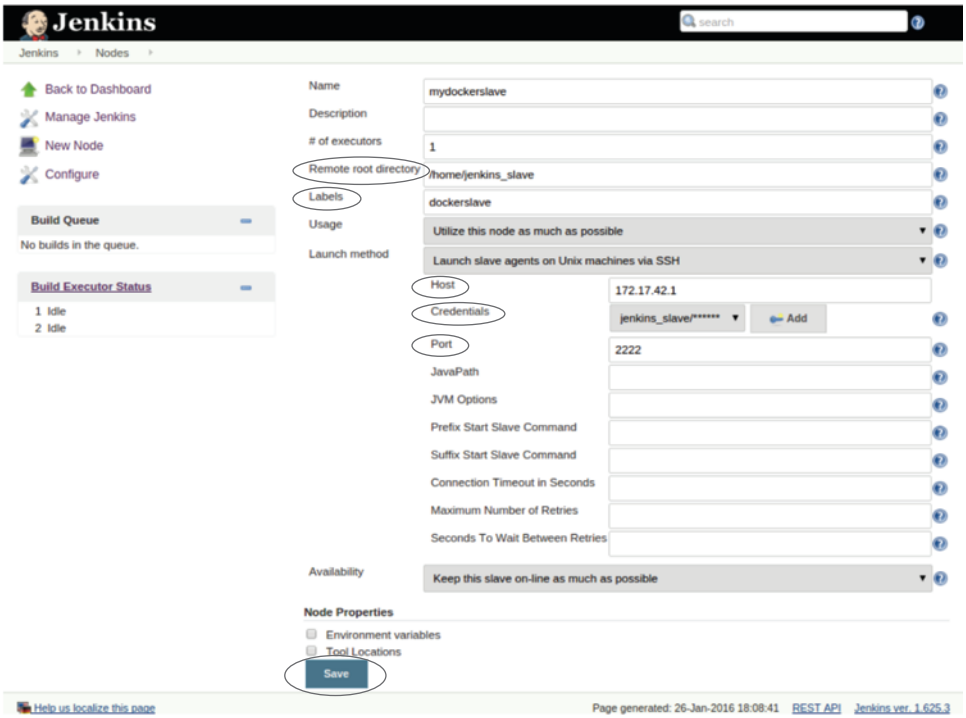


Figure 6.6 Configuring the new node

The job is now linked to the Docker slave. Click Build Now, then click the build that appears below on the left, and then click Console Output, and you should see output like this in the main window:

```
Started by user anonymous
Building remotely on testslave (dockerslave) in workspace
/home/jenkins_slave/workspace/ls
[ls] $ /bin/sh -xe /tmp/hudson4490746748063684780.sh
+ echo done
done
Finished: SUCCESS
```

Well done! You've successfully created your own Jenkins slave.

Now if you want to create your own bespoke slave, all you need to do is alter the slave image's Dockerfile to your taste, and run that instead of the example one.

AVAILABLE ON GITHUB The code for this technique and related ones is available on GitHub at https://github.com/docker-in-practice/jenkins.

TECHNIQUE 60 Running the Jenkins master within a Docker container

Putting the Jenkins master inside a container doesn't have as many benefits as doing the same for a slave, but it does give you the normal Docker win of immutable images.

We've found that being able to commit known-good master configurations and plugins eases the burden of experimentation significantly.

PROBLEM

You want a portable Jenkins server.

SOLUTION

Use a Jenkins Docker image.

DISCUSSION

Running Jenkins within a Docker container gives you some advantages over a straightforward host install. Cries of "Don't touch my Jenkins server configuration!" or, even worse, "Who touched my Jenkins server?" aren't unheard of in our office, and being able to clone the state of a Jenkins server with a `docker export` of the running container to experiment with upgrades and changes helps silence these complaints. Similarly, backups and porting become easier.

In this technique we'll take the official Jenkins Docker image and make a few changes to facilitate some later techniques that require the ability to access the Docker socket, like doing a Docker build from Jenkins.

> **DIRECT FROM THE SOURCE** The Jenkins-related examples from this book are available on GitHub: git clone https://github.com/docker-in-practice/jenkins.git.

> **A COMMON BASELINE** This Jenkins image and its `run` command will be used as the server in Jenkins-related techniques in this book.

BUILDING THE SERVER

We'll first prepare a list of plugins we want for the server and place it in a file called jenkins_plugins.txt:

```
swarm:1.22
```

This very short list consists of the Jenkins Swarm plugin (no relation to Docker Swarm), which we'll use in a later technique.

The following listing shows the Dockerfile for building the Jenkins server.

Listing 6.5 Jenkins server build

```
FROM jenkins
COPY jenkins_plugins.txt /tmp/jenkins_plugins.txt
RUN /usr/local/bin/plugins.sh /tmp/jenkins_plugins.txt
USER root
RUN rm /tmp/jenkins_plugins.txt
RUN groupadd -g 142 docker
RUN addgroup -a jenkins docker
USER jenkins
```

Use the official Jenkins image as a base.

Copy a list of plugins to install.

Run the plugins into the server.

Switch to the root user and remove the plugins file.

Switch back to the Jenkins user in the container.

Add the Docker group to the container with the same group ID as your host machine (the number may differ for you).

No CMD or ENTRYPOINT instruction is given because we want to inherit the startup command defined in the official Jenkins image.

The group ID for Docker may be different on your host machine. To see what the ID is for you, run this command to see the local group ID:

```
$ grep -w ^docker /etc/group
docker:x:142:imiell
```

Replace the value if it differs from 142.

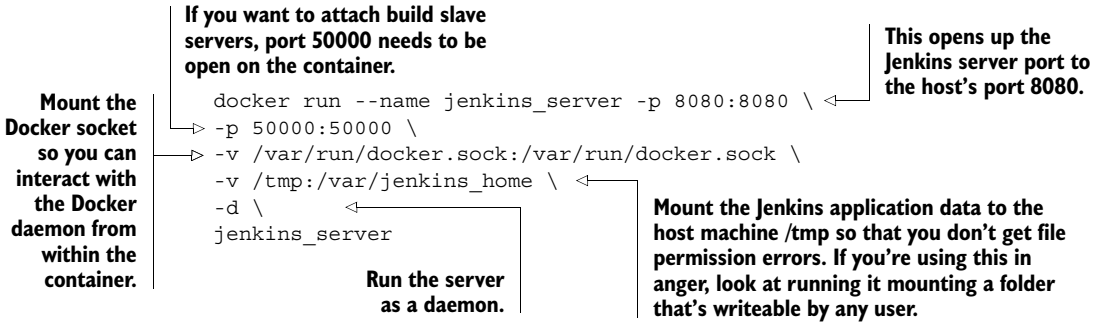
MATCHING GROUP IDS ACROSS ENVIRONMENTS The group ID must match on the Jenkins server environment and your slave environment if you plan to run Docker from within the Jenkins Docker container. If you do, there will be a potential portability issue if you choose to move the server (you'd encounter the same issue on a native server install). Environment variables won't help here by themselves, as the group needs to be set up at build time rather than being dynamically configured.

To build the image in this scenario, run this command:

```
docker build -t jenkins_server .
```

RUNNING THE SERVER

Now you can run the server under Docker with this command:



If you access http://localhost:8080, you'll see the Jenkins server ready to go with your plugins already installed. To check this, go to Manage Jenkins > Manage Plugins > Installed and look for Swarm to verify that it's installed.

AVAILABLE ON GITHUB The code for this technique and related ones is available on GitHub at https://github.com/docker-in-practice/jenkins.

TECHNIQUE 61 Scale your CI with Jenkins' Swarm plugin

Being able to reproduce environments is a big win, but your build capacity is still constrained by the number of dedicated build machines you have available. If you want to do experiments on different environments with the newfound flexibility of Docker

slaves, this may become frustrating. Capacity can also become a problem for more mundane reasons—the growth of your team!

PROBLEM

You want your CI compute to scale up with your development work rate.

SOLUTION

Use Jenkins' Swarm plugin and a Docker swarm slave to dynamically provision Jenkins slaves.

DISCUSSION

Many small- to medium-sized businesses have a model for CI where there are one or more Jenkins servers devoted to supplying the resources required to run Jenkins jobs. This is illustrated in figure 6.7.

This works fine for a time, but as the CI processes become more embedded, the capacity limits are often reached. Most Jenkins workloads are triggered off check-ins to source control, so as more developers check in, the workload increases. The number of complaints to the ops team then explodes as busy developers impatiently wait for their build results.

One neat solution is to have as many Jenkins slaves as there are people checking in code, as illustrated in figure 6.8.

The Dockerfile shown in listing 6.6 creates an image with the Jenkins Swarm client plugin installed, allowing a Jenkins master with the appropriate Jenkins Swarm server

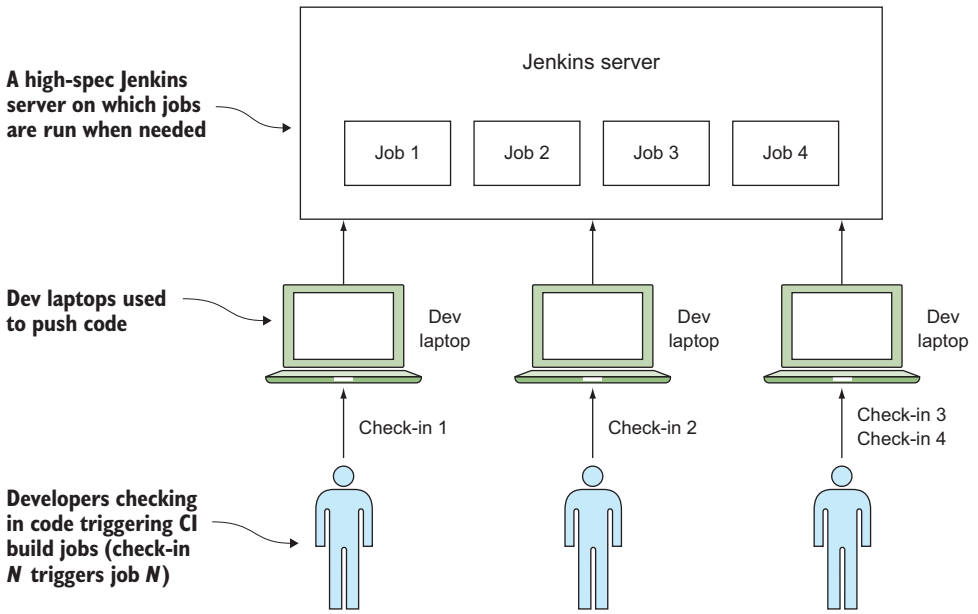


Figure 6.7 Before: Jenkins server—OK with one dev, but doesn't scale

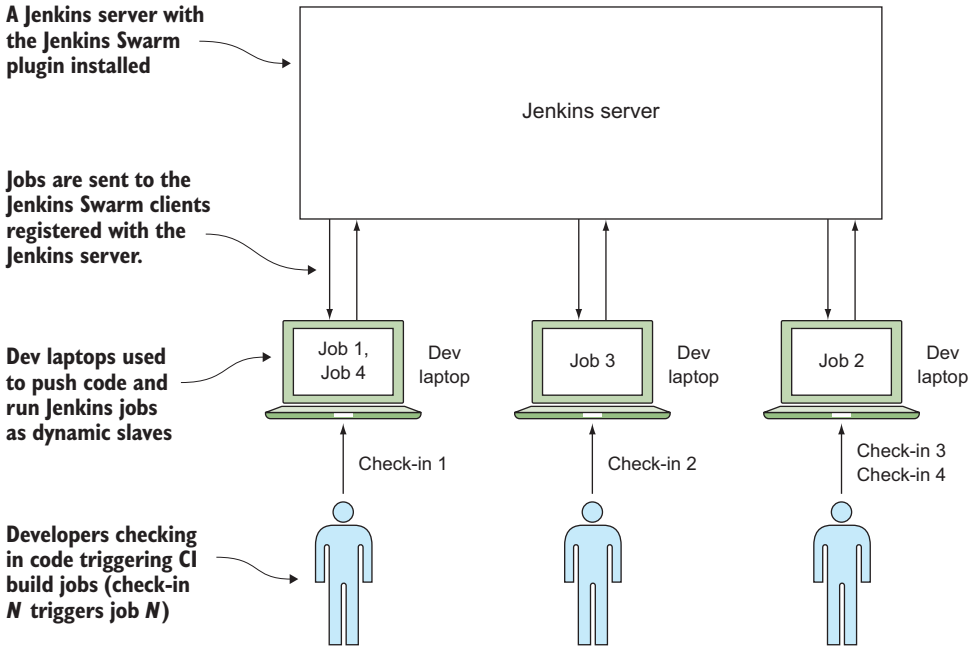


Figure 6.8 **After: compute scales with team**

plugin to connect and run jobs. It begins in the same way as the normal Jenkins slave Dockerfile in the last technique.

Listing 6.6 Dockerfile

```
FROM ubuntu
ENV DEBIAN_FRONTEND noninteractive
RUN groupadd -g 1000 jenkins_slave
RUN useradd -d /home/jenkins_slave -s /bin/bash \
-m jenkins_slave -u 1000 -g jenkins_slave
RUN echo jenkins_slave:jpass | chpasswd
RUN apt-get update && apt-get install -y openjdk-7-jre wget unzip
RUN wget -O /home/jenkins_slave/swarm-client-1.22-jar-with-dependencies.jar \
http://maven.jenkins-ci.org/content/repositories/releases/org/jenkins-ci/
➥ plugins/swarm-client/1.22/swarm-client-1.22-jar-with-dependencies.jar
COPY startup.sh /usr/bin/startup.sh
RUN chmod +x /usr/bin/startup.sh
ENTRYPOINT ["/usr/bin/startup.sh"]
```

Retrieve the Jenkins Swarm plugin.

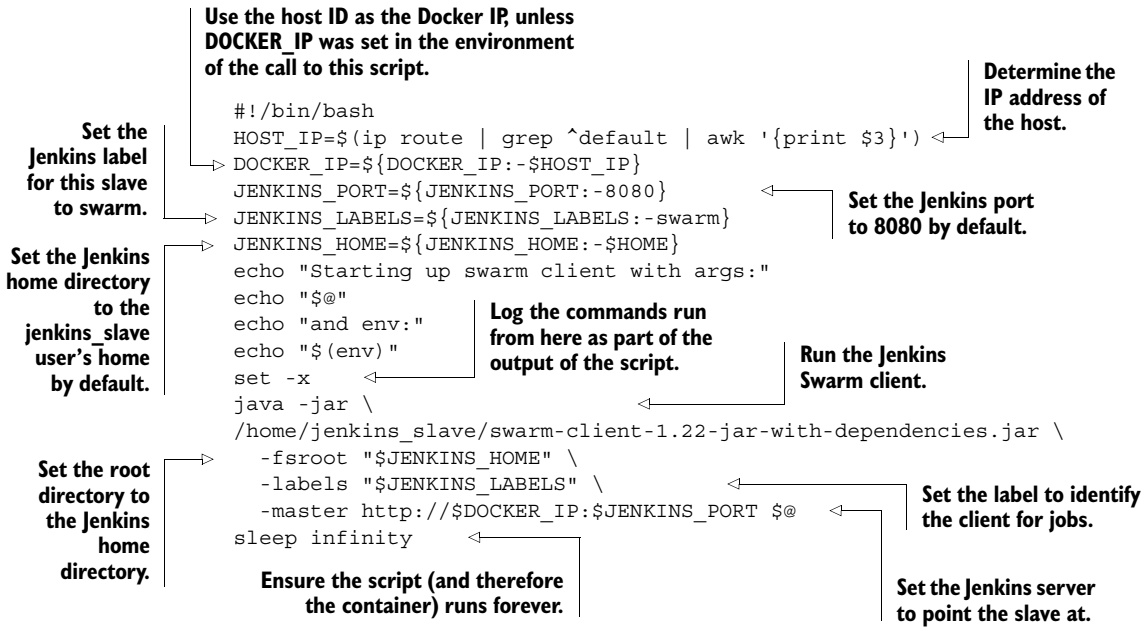
Copy the startup script to the container.

Mark the startup script as executable.

Make the startup script the default command run.

The following listing is the startup script copied into the preceding Dockerfile.

Listing 6.7 startup.sh



Most of the preceding script sets up and outputs the environment for the Java call at the end. The Java call runs the Swarm client, which turns the machine on which it's run into a dynamic Jenkins slave rooted in the directory specified in the `-fsroot` flag, running jobs labeled with the `-labels` flag and pointed at the Jenkins server specified with the `-master` flag. The lines with `echo` just provide some debugging information about the arguments and environment setup.

Building and running the container is a simple matter of running what should be the now-familiar pattern:

```
$ docker build -t jenkins_swarm_slave .
$ docker run -d --name \
jenkins_swarm_slave jenkins_swarm_slave
```

Now that you have a slave set up on this machine, you can run Jenkins jobs on them. Set up a Jenkins job as normal, but add `swarm` as a label expression in the Restrict Where This Project Can Be Run section (see technique 59).

SET UP A SYSTEM SERVICE TO SPREAD THIS AROUND You can automate this process by setting it up as a supervised system service on all of your estate's PCs (see technique 75).

PERFORMANCE IMPACT ON SLAVE MACHINES Jenkins jobs can be onerous processes, and it's quite possible that their running will negatively affect the laptop. If the job is a heavy one, you can set the labels on jobs and Swarm clients appropriately. For example, you might set a label on a job as 4CPU8G and match it to Swarm containers run on 4 CPU machines with 8 GB of memory.

This technique gives some indication of the Docker concept. A predictable and portable environment can be placed on multiple hosts, reducing the load on an expensive server and reducing the configuration required to a minimum.

Although this is not a technique that can be rolled out without considering performance, we think there's a lot of scope here to turn contributing developer computer resources into a form of game, increasing efficiencies with a development organization without needing expensive new hardware.

AVAILABLE ON GITHUB The code for this technique and related ones is available on GitHub at https://github.com/docker-in-practice/jenkins.

6.4 Summary

In this chapter we've shown how Docker can be used to enable and facilitate CI within your organization. You've seen how many of the barriers to CI, such as the availability of raw compute and sharing resources with others, can be overcome with Docker's help.

In this chapter you learned that

- Builds can be sped up significantly by using eatmydata and package caches.
- You can run GUI tests (like Selenium) inside Docker.
- A Docker CI slave lets you keep complete control over your environment.
- You can farm out build processes to your whole team using Docker and Jenkins' Swarm plugin.

In the next chapter we'll move away from CI to deployment and cover techniques related to continuous delivery, another key component of the DevOps picture.

PROGRAMMING/SYSTEM ADMINISTRATION

Docker IN PRACTICE

Miell • Hobson Sayers

An open source container system, Docker makes deploying applications painless and flexible. Docker is powerful and simple to use, and it makes life easier for developers and administrators alike providing shorter build times, fewer production bugs, and effortless application roll out.

Docker in Practice is a hands-on guide that covers 101 specific techniques you can use to get the most out of Docker. Following a cookbook-style Problem/Solution/Discussion format, this practical handbook gives you instantly useful solutions for important problems like effortless server maintenance and configuration, deploying microservices, creating safe environments for experimentation, and much more. As you move through this book, you'll advance from basics to Docker best practices like using it with your Continuous Integration process, automating complex container creation with Chef, and orchestration with Kubernetes.

What's Inside

- Speeding up your DevOps pipeline
- Cheaply replacing VMs
- Streamlining your cloud workflow
- Using the Docker Hub
- Navigating the Docker ecosystem

For anyone interested in *real-world* Docker.

Ian Miell and Aidan Hobson Sayers have contributed to Docker and have extensive experience building and maintaining commercial Docker-based infrastructures in large-scale environments.

"A deluge of practical advice about applying Docker to problems you have right now."
—From the Foreword by Ben Firshman, Docker, Inc.

"Filled with 4-star recipes!"
—Chad Davis, SolidFire

"You'll love Docker after reading this book."
—José San Leandro, OSOCO

"Packed with Docker tricks of the developer trade."
—Kirk Brattkus
Net Effect Technologies

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

ISBN-13: 978-1-61729-272-9
ISBN-10: 1-61729-272-9
54499
9 781617 292729

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]