8 Continuous integration: Speeding up your development pipeline

This chapter covers

- Using the Docker Hub workflow as a CI tool
- Speeding up your I/O-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 continuous integration (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 CI. We've found the techniques in this chapter to be invaluable in making a CI process feasible within a business.

By the end of this chapter you'll understand how Docker can make the process of CI faster, more stable, and reproducible. By using test tools such as Selenium, and expanding your build capacity with the Jenkins Swarm plugin, you'll see how Docker can help you get even more out of your CI process.

> **NOTE** 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.

8.1 Docker Hub automated builds

The Docker Hub automated build feature was mentioned in technique 10, 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 61 Using the Docker Hub workflow

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

> **NOTE** 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.

Although the Docker Hub build isn't complicated, a number of steps are required:

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.
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.

NOTE 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.

CREATE YOUR REPOSITORY ON GITHUB OR BITBUCKET

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

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.

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 8.1 Dockerfile—simple dev tools container Dockerfile

```
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"]
```

Installs useful packages

Adds a line to the root's bashrc to source bash_extra

Adds bash_extra from the source to the container

Now you'll need to add the bash_extra file you referenced and give it the following content:

```
bash --version
```

This file is just for illustration. It shows that you can create a bash file that's sourced on startup. In this case it displays the version of bash you're using in your shell, but it could contain all manner of things that set up your shell to your preferred state.

COMMIT THE SOURCE

To commit your source code source, use this command:

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

PUSH THE GIT REPOSITORY

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

```
git push origin master
```

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 Create and choose Create Automated Build.

For the first time only, you'll need to go through the account-linking process. You'll see a prompt to link your account to a hosted Git service. Select your service and follow the instructions to link your account. You may be offered the choice to give full or more limited access to Docker Inc. for the integration. If you opt for the more limited access, you should read the official documentation for your specific service to identify what extra work you might need to do during the rest of the steps.

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.

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

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 click 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, something has gone wrong and you'll need to click on the build ID to see what the error was.

> **NOTE** 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.

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.

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.

> **TIP** 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'll 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.

DISCUSSION

Because the Docker Hub is the canonical source of images, pushing there during your CI process can make some things more straightforward (distributing images to third parties, for one). Not having to run the build process yourself is easier and gives you some additional benefits, like a checkmark against the listing on the Docker Hub indicating that the build was performed on a trusted server.

Having this additional confidence in your builds helps you comply with the *Docker contract* in technique 70—in technique 113 we'll look at how specific machines can sometimes affect Docker builds, so using a completely independent system is good for increasing confidence in the final results.

8.2 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 62 Speeding 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

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.

INSTALLING EATMYDATA

To install eatmydata in your container, 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 8.2 Compiling and installing eatmydata

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

Flamingspork.com is the website of the maintainer.

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

Builds the eatmydata executable

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

Installs 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 8.1 shows at a high level how eatmydata saves you processing time.

> **WARNING** 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!

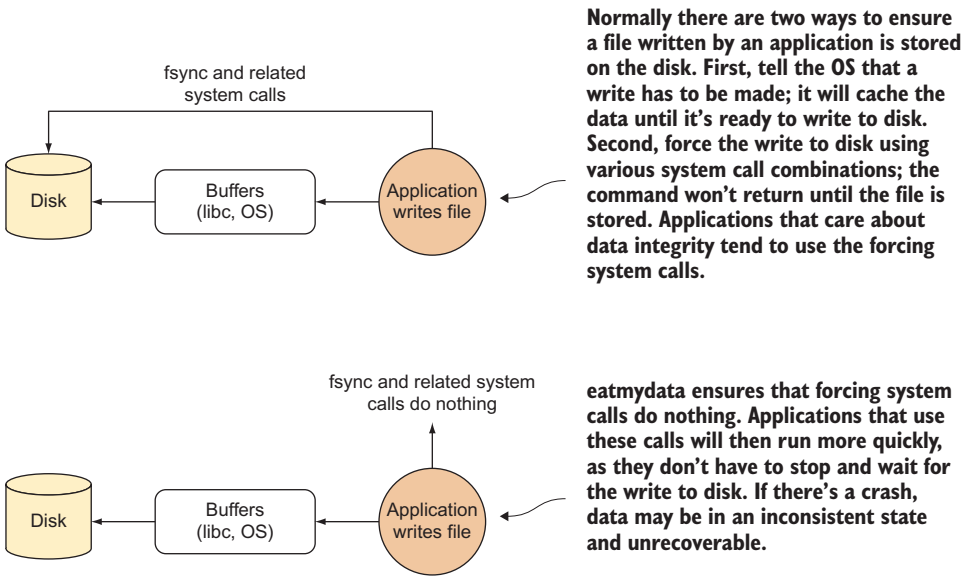


Figure 8.1 Application writes to disk without (top) and with (bottom) eatmydata

Be aware that running `eatmydata docker run ...` to start a Docker container, perhaps after installing eatmydata on your host or mounting the Docker socket, will not have the effect you may expect due to the Docker client/server architecture outlined in chapter 2. Instead, you need to install eatmydata inside each individual container you want to use it in.

DISCUSSION

Although precise use cases will vary, one place you should immediately be able to apply this is in technique 68. It's very rare for the data integrity on a CI job to matter—you're usually just interested in success or failure, and the logs in the case of failure.

One other relevant technique is technique 77. A database is one place where data integrity really does matter a lot (any popular one will be designed to not lose data in the case of machine power loss), but if you're just running some tests or experiments, it's overhead that you don't need.

TECHNIQUE 63 Setting 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.

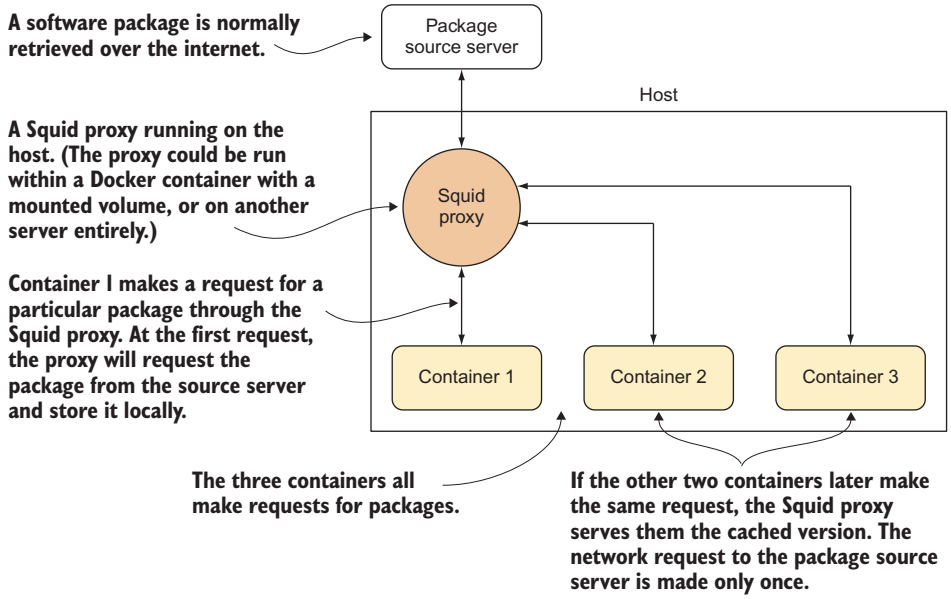


Figure 8.2 Using a Squid proxy to cache packages

SOLUTION

Install a Squid proxy for your package manager. Figure 8.2 illustrates how this technique works.

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.

> **NOTE** 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. For this reason, you may want to convert this Dockerfile to a script to be run from within the container before installing new software.

Listing 8.3 Configuring a Debian image to use an apt proxy

```
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"]
```

Ensures the route tool is installed

To determine the host's IP address from the point of view of the container, runs the route command and uses awk to extract the relevant IP address from the output (see technique 67).

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

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'll 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.

Listing 8.4 Configuring a CentOS image to use a yum proxy

```
FROM centos:centos7
RUN yum update -y && yum install -y net-tools
 RUN echo "proxy=http://$(route -n | \
 awk '/^0.0.0.0/ {print $2}'):3128" >> /etc/yum.conf
 RUN sed -i 's/^mirrorlist/#mirrorlist/' \
/etc/yum.repos.d/CentOS-Base.repo
 RUN sed -i 's/^#baseurl/baseurl/' \
/etc/yum.repos.d/CentOS-Base.repo
 RUN rm -f /etc/yum/pluginconf.d/fastestmirror.conf
 RUN yum update -y
 CMD ["/bin/bash"]
```

Ensures the route tool is installed

To determine the host's IP address from the point of view of the container, runs the route command and uses awk to extract the relevant IP address from the output

Port 3128 is used to connect to the Squid proxy on the host machine.

Removes the fastestmirror plugin, as it's no longer required.

Ensures the mirrors are checked. When running yum update, the mirrors listed in the config files may have out-of-date information, so the first update will be slow.

To avoid cache misses where possible, removes the mirror lists and uses the base URLs only. This ensures you only hit one set of URLs to fetch packages, and you're therefore more likely to hit the cached file.

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.

DISCUSSION

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 64 Headless Chrome in a container

Running tests is a crucial part of CI, and most unit test frameworks will run within Docker without any issues. But sometimes more involved testing is called for, from making sure multiple microservices cooperate correctly to ensuring that website frontend functionality still works. Visiting a website frontend requires a browser of some kind, so to solve this problem we need a way to start a browser inside a container, and then to control it programmatically.

PROBLEM

You want to test against the Chrome browser within a container, without needing a GUI.

SOLUTION

Use the Puppeteer Node.js library in an image to automate Chrome actions.

This library is maintained by the Google Chrome development team, and it allows you to write scripts against Chrome for testing purposes. It's "headless," which means you don't need a GUI to work against it.

NOTE This image is also maintained by us on GitHub at https://github.com/docker-in-practice/docker-puppeteer. It's also accessible as a Docker image with `docker pull dockerinpractice/docker-puppeteer`.

The following listing shows a Dockerfile that will create an image containing all you need to get started with Puppeteer.

Listing 8.5 Puppeteer Dockerfile

```
FROM ubuntu:16.04
 RUN apt-get update -y && apt-get install -y \
     npm python-software-properties curl git \
     libpangocairo-1.0-0 libx11-xcb1 \
     libxcomposite1 libxcursor1 libxdamage1 \
     libxi6 libxtst6 libnss3 libcups2 libxss1 \
     libxrandr2 libgconf-2-4 libasound2 \
     libatk1.0-0 libgtk-3-0 vim gconf-service \
     libappindicator1 libc6 libcairo2 libcups2 \
     libdbus-1-3 libexpat1 libfontconfig1 libgcc1 \
     libgdk-pixbuf2.0-0 libglib2.0-0 libnspr4 \
     libpango-1.0-0 libstdc++6 libx11-6 libxcb1 \
     libxext6 libxfixes3  libxrender1 libxtst6 \
     ca-certificates fonts-liberation lsb-release \
     xdg-utils wget
 RUN curl -sL https://deb.nodesource.com/setup_8.x | bash -
 RUN apt-get install -y nodejs
 RUN useradd -m puser
 USER puser
 RUN mkdir -p /home/puser/node_modules
 ENV NODE_PATH /home/puppeteer/node_modules
 WORKDIR /home/puser/node_modules
 RUN npm i webpack
 RUN git clone https://github.com/GoogleChrome/puppeteer
 WORKDIR /home/puser/node_modules/puppeteer
 RUN npm i .
 WORKDIR /home/puser/node_modules/puppeteer/examples
 RUN perl -p -i -e \
     "s/puppeteer.launch\(\)/puppeteer.launch({args: ['--no-sandbox']})/" *
 CMD echo 'eg: node pdf.js' && bash
```

Starts with an Ubuntu base image

Installs all the software required. This is most of the display libraries required to get Chrome to operate within a container.

Sets up the latest nodejs version

Installs the Ubuntu nodejs package

Creates a non-root user, "puser" (which the library requires to run)

Installs webpack (a dependency of Puppeteer)

Creates a node modules folder

Sets the NODE_PATH environment variable to the node_module folder

Goes to the Puppeteer code folder

Clones the Puppeteer module code

Sets the current working directory to the node module path

Goes to the Puppeteer examples folder

Installs the Puppeteer NodeJS library

Adds a no-sandbox argument to the Puppeteer launch arguments, to overcome a security setting when running within a container

Starts the container with bash, adding a helpful echo command

Build and run this Dockerfile with this command:

```
$ docker build -t puppeteer .
```

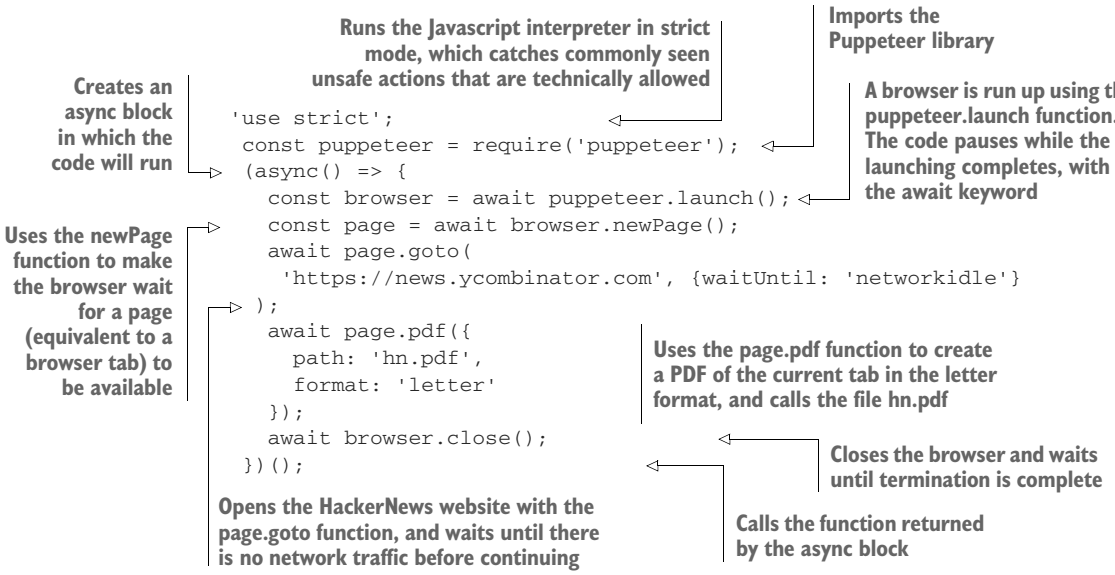
Then run it:

```
$ docker run -ti puppeteer
eg: node pdf.js
puser@03b9be05e81d:~/node_modules/puppeteer/examples$
```

You'll be presented with a terminal and the suggestion to run `node pdf.js`.

The pdf.js file contains a simple script that serves as an example of what can be done with the Puppeteer library.

Listing 8.6 pdf.js



A host of options are available to the Puppeteer user beyond this simple example. It's beyond the scope of this technique to explain the Puppeteer API in detail. If you want to look in more depth at the API and adapt this technique, take a look at the Puppeteer API documentation on GitHub: https://github.com/GoogleChrome/puppeteer/blob/master/docs/api.md.

DISCUSSION

This technique shows you how Docker can be used to test against a specific browser.

The next technique broadens this one in two ways: by using Selenium, a popular testing tool that can work against multiple browsers, and combining this with some exploration of X11 to allow you to see a browser running in an graphical window rather than in the headless fashion used in this technique.

TECHNIQUE 65 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 19), 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 you can write graphical tests using Selenium. We'll 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.

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 on your host:

```
~ $ 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 use 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 need to get the 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 29 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 and using the same username as outside the container, you can use your existing .Xauthority file.

Listing 8.7 Starting a container with an Xauthority-enabled display

```
$ ls $HOME/.Xauthority
/home/myuser/.Xauthority
$ docker run -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix \
    --hostname=$HOSTNAME -v $HOME/.Xauthority:$HOME/.Xauthority \
    -it -e EXTUSER=$USER ubuntu:16.04 bash -c 'useradd $USER && exec bash'
```

The second method of allowing Docker to access the socket is a much blunter instrument and it has security issues, because 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 .Xauthority file first. You can secure yourself again after you try the following steps by running `xhost -` (though this will lock out your Docker container):

Listing 8.8 Starting a container with xhost-enabled display

```
$ xhost +
access control disabled, clients can connect from any host
$ docker run -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix \
    -it ubuntu:16.04 bash
```

The first line in the preceding listing 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 if going the .Xauthority route:

```
root@myhost:/# apt-get update && apt-get install -y x11-apps
[...]
root@myhost:/# su - $EXTUSER -c "xeyes"
```

Alternatively you can use these slightly different commands if you're going the xhost route, because you don't need to run the command as a specific user:

```
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 it's 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.

The following listing first installs Python, Firefox, and a Python package manager, and then it uses the Python package manager to install the Selenium Python package. It also downloads the "driver" binary that Selenium uses to control Firefox. A Python REPL is then started, and the Selenium library is used to create a Firefox instance.

For simplicity, this will only cover the xhost route—to go the Xauthority route, you'll need to create a home directory for the user so Firefox has somewhere to save its profile settings.

Listing 8.9 Installing the Selenium requirements and starting a browser

```
root@myhost:/# apt-get install -y python2.7 python-pip firefox wget
[...]
root@myhost:/# pip install selenium
Collecting selenium
[...]
Successfully installed selenium-3.5.0
root@myhost:/# url=https://github.com/mozilla/geckodriver/releases/download
➥ /v0.18.0/geckodriver-v0.18.0-linux64.tar.gz
root@myhost:/# wget -qO- $url | tar -C /usr/bin -zxf -
root@myhost:/# 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.

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('https://github.com/search')
>>> searchselector = '#search_form input[type="text"]'
>>> searchbox = b.find_element_by_css_selector(searchselector)
>>> searchbox.send_keys('docker-in-practice')
>>> searchbox.submit()
>>> import time
>>> time.sleep(2) # wait for page JS to run
>>> 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 = '.meta-item a'
>>> mlink = b.find_element_by_css_selector(mlinkselector)
>>> mlink.click()
```

The details here aren't important, though you can get an idea of what's going on by switching to Firefox between commands—we're navigating to the docker-in-practice organization on GitHub, and clicking the organization link. The main takeaway is 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 it doesn't require a monitor.

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

Listing 8.10 Creating a Selenium test script

```
>>> exit()
root@myhost:/# apt-get install -y xvfb
[...]
root@myhost:/# exit
$ docker commit ef351febcee4 selenium
d1cbfbc76790cae5f4ae95805a8ca4fc4cd1353c72d7a90b90ccfb79de4f2f9b
$ cat > myscript.py << EOF
from selenium import webdriver
b = webdriver.Firefox()
print 'Visiting github'
b.get('https://github.com/search')
print 'Performing search'
searchselector = '#search_form input[type="text"]'
searchbox = b.find_element_by_css_selector(searchselector)
searchbox.send_keys('docker-in-practice')
searchbox.submit()
print 'Switching to user search'
import time
time.sleep(2) # wait for page JS to run
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 = '.meta-item a'
mlink = b.find_element_by_css_selector(mlinkselector)
mlink.click()
print 'Done!'
EOF
```

Note the subtle difference in the assignment of the `dlink` variable (indexing to position `99` rather than `0`). 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 nonzero 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 nonzero exit code.

> **NOTE** The `sh -c "command string here"` is an unfortunate result of how Docker treats `CMD` values by default. If you built this image with a Dockerfile, you'd be able to remove the `sh -c` and make `xvfb-run -s '-screen 0 1024x768x24 -extension RANDR'` the entrypoint, allowing you to pass the test command as image arguments.

DISCUSSION

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 (technique 40 does look at doing this for at least your developer tools) but we've found that re-examining assumptions about Docker can lead to some surprising use cases. For example, appendix A talks about running graphical Linux applications on Windows after installing Docker for Windows.

8.3 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 isn't essential to the techniques here.

TECHNIQUE 66 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 (see the next technique), 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 the official Jenkins Docker image to run your server.

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.

NOTE The Jenkins-related examples from this book are available on GitHub: `git clone https://github.com/docker-in-practice/jenkins.git`.

NOTE This Jenkins image and its `run` command will be used as the server in Jenkins-related techniques in this book.

BUILDING THE SERVER

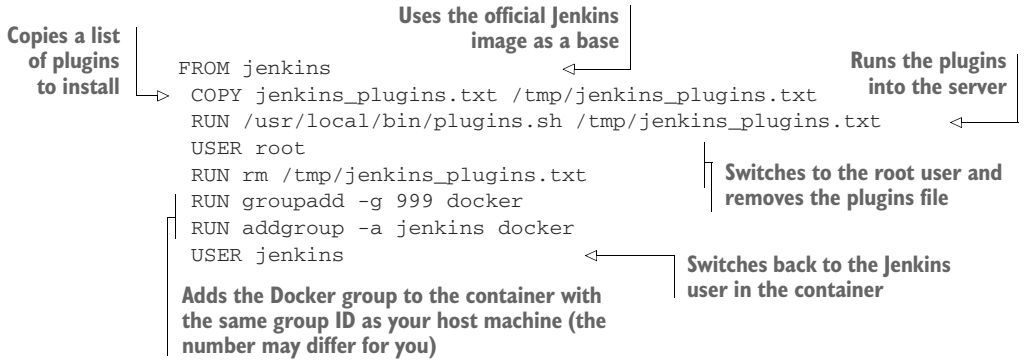
We'll first prepare a list of plugins we want for the server and place it in a file called jenkins_plugins.txt:

```
swarm:3.4
```

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 8.11 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 999 docker
RUN addgroup -a jenkins docker
USER jenkins
```

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:999:imiell
```

Replace the value if it differs from 999.

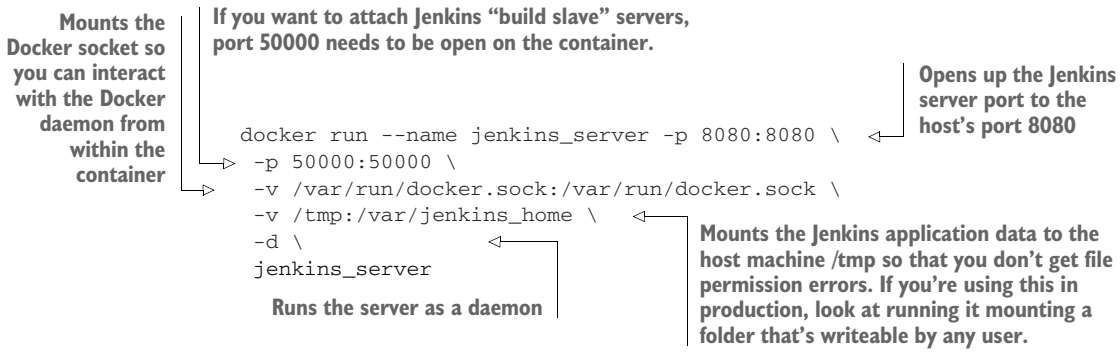
WARNING The group ID must match on the Jenkins server environment and your slave environment if you plan to run Docker from within the Jenkins Docker container. There will also be a potential portability issue if you choose to move the server (you'd encounter the same issue on a native server install). 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 configuration interface—follow the process to your linking, probably using `docker exec` (described in technique 12) to retrieve the password you'll be prompted for at the first step.

Once complete, your Jenkins server will be ready to go, with your plugins already installed (along with some others, depending on the options you selected during the setup process). To check this, go to Manage Jenkins > Manage Plugins > Installed, and look for Swarm to verify that it's installed.

DISCUSSION

You'll see that we've mounted the Docker socket with this Jenkins master as we did in technique 45, providing access to the Docker daemon. This allows you to perform Docker builds with the built-in master slave by running the containers on the host.

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

TECHNIQUE 67 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. Do you 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

Use Docker to encapsulate the configuration of your slave in a Docker image, and deploy.

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

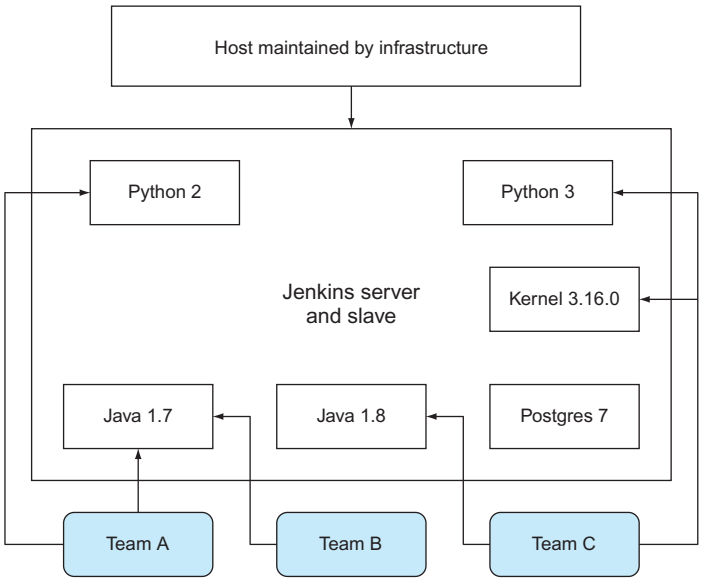


Figure 8.3 An overloaded Jenkins server

NOTE 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.

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 8.12 Bare-bones Jenkins slave Dockerfile

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

```
FROM ubuntu:16.04
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-8-jre wget iproute2
 RUN mkdir -p /var/run/sshd
 CMD ip route | grep "default via" \
  | awk '{print $3}' && /usr/sbin/sshd -D
```

Creates the Jenkins slave user and group

Installs the required software to function as a Jenkins slave.

On startup, outputs the IP address of the host machine from the point of view of the container, and starts 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
172.17.0.1
```

Jenkins server needs to be running

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

(continued)

```
$ 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. You'll need to go through the setup process before using it.

If you navigate to the Jenkins server, you'll be greeted with the page in figure 8.4.

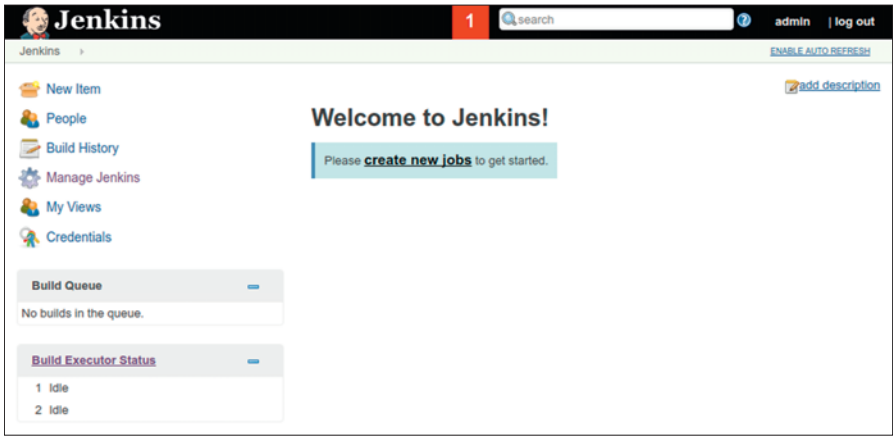


Figure 8.4 The Jenkins homepage

You can add a slave by clicking on Build Executor Status > New Node and adding the node name as a Permanent Agent, as shown in figure 8.5. Call it `mydockerslave`.

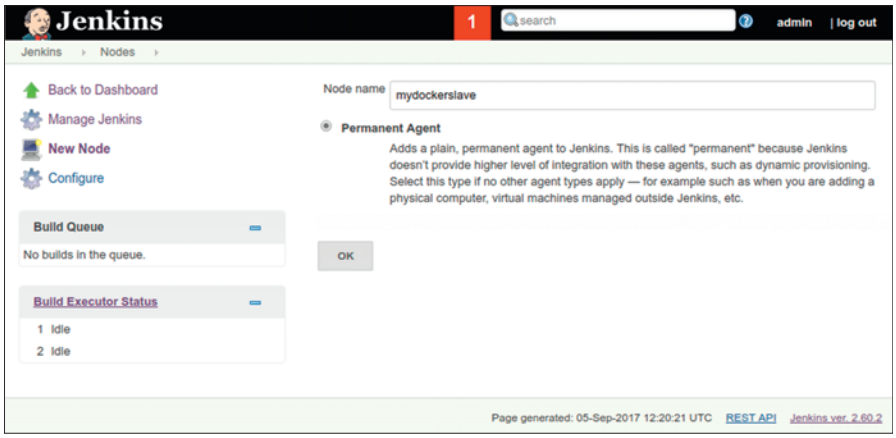


Figure 8.5 Naming a new node page

Click OK and configure it with these settings, as shown in figure 8.6:

- Set Remote Root Directory to /home/jenkins_slave.
- Give it a Label of "dockerslave".
- Make sure the Launch Slave Agents Via SSH option is selected.
- Set the host to the route IP address seen from within the container (output with the `docker run` command earlier).
- Click Add to add credentials, and set the username to "jenkins_slave" and the password to "jpass". Now select those credentials from the drop-down list.
- Set Host Key Verification Strategy to either Manually Trusted Key Verification Strategy, which will accept the SSH key on first connect, or Non Verifying Verification Strategy, which will perform no SSH host key checking.
- Click Advanced to expose the Port field, and set it to 2222.
- Click Save.

Now click through to the new slave, and click Launch Slave Agent (assuming this doesn't happen automatically). After a minute you should see that the slave agent is marked as online.

Go back to the homepage by clicking on Jenkins at the top left, and click on New Item. Create a Freestyle Project called "test", and under the Build section, click 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, meaning the job is now linked to the Docker slave. Click Save to create the job.

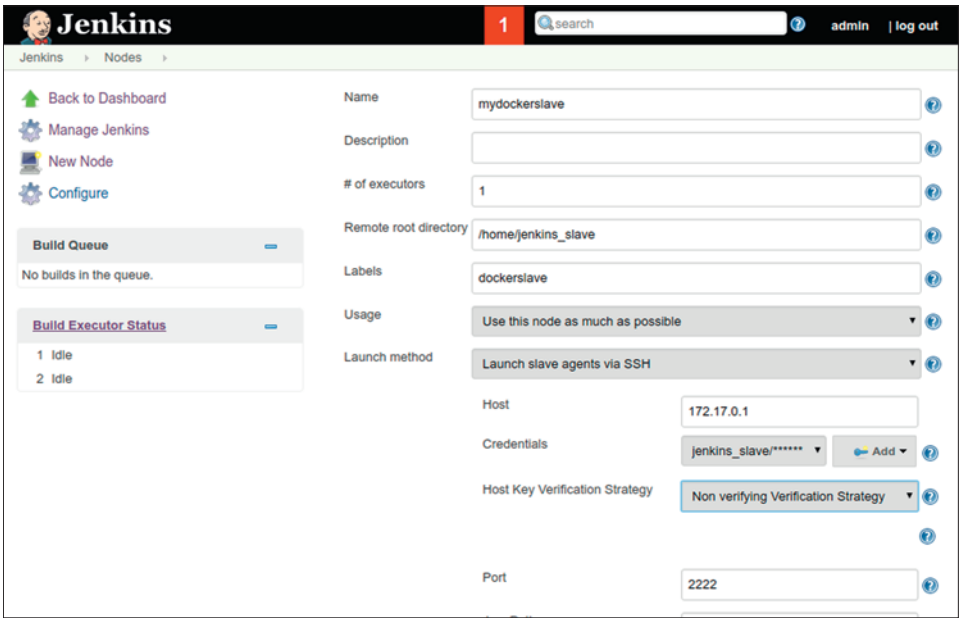


Figure 8.6 The Jenkins node settings page

Click Build Now, and then click the build "#1" link that appears below on the left. Then click Console Output, and you should see output like this in the main window:

```
Started by user admin
Building remotely on mydockerslave (dockerslave)
➥ in workspace /home/jenkins_slave/workspace/test
[test] $ /bin/sh -xe /tmp/jenkins5620917016462917386.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.

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

DISCUSSION

This technique walks you down the road of creating a container to act like a virtual machine, much like technique 12 but with the added complexity of Jenkins integration. One particularly useful strategy is to also mount the Docker socket inside the container and install the Docker client binary so you can perform Docker builds. See technique 45 for information on mounting the Docker socket (for a different purpose) and appendix A for installation details.

TECHNIQUE 68 Scaling 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.

> **NOTE** It's been mentioned before, but it's worth repeating here: the Jenkins' Swarm plugin is not at all related to Docker's Swarm technology. They are two entirely unrelated things that happen to use the same word. The fact that they can be used together here is pure coincidence.

Many small- to medium-sized businesses have a model for CI where one or more Jenkins servers are devoted to supplying the resources required to run Jenkins jobs. This is illustrated in figure 8.7.

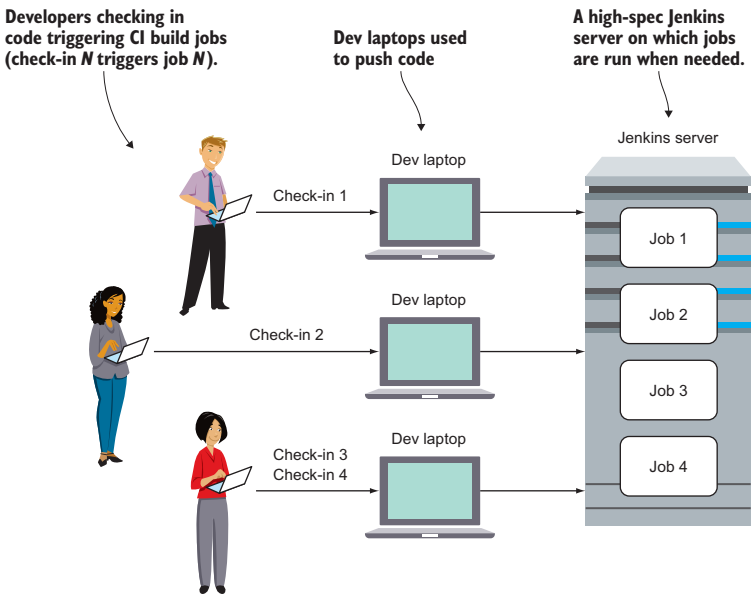


Figure 8.7 Before: Jenkins server—OK with one dev, but doesn't scale

This works fine for a time, but as the CI processes become more embedded, capacity limits are often reached. Most Jenkins workloads are triggered by 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 8.8.

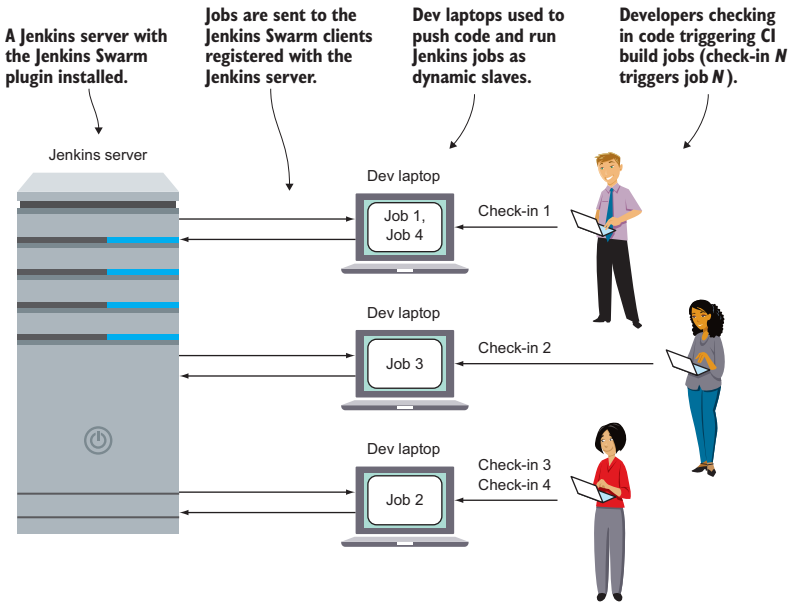


Figure 8.8 After: compute scales with team

The Dockerfile shown in listing 8.13 creates an image with the Jenkins Swarm client plugin installed, allowing a Jenkins master with the appropriate Jenkins Swarm server plugin to connect and run jobs. It begins in the same way as the normal Jenkins slave Dockerfile in the last technique.

Listing 8.13 Dockerfile

```
FROM ubuntu:16.04
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-8-jre wget iproute2
RUN wget -O /home/jenkins_slave/swarm-client-3.4.jar \
 https://repo.jenkins-ci.org/releases/org/jenkins-ci/plugins/swarm-client
➥ /3.4/swarm-client-3.4.jar
COPY startup.sh /usr/bin/startup.sh
 RUN chmod +x /usr/bin/startup.sh
 ENTRYPOINT ["/usr/bin/startup.sh"]
```

Retrieves the Jenkins Swarm plugin

Copies the startup script to the container

Marks the startup script as executable

Makes the startup script the default command run

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

Listing 8.14 startup.sh

```
#!/bin/bash
export HOST_IP=$(ip route | grep ^default | awk '{print $3}')
 export JENKINS_IP=${JENKINS_IP:-$HOST_IP}
 export JENKINS_PORT=${JENKINS_PORT:-8080}
 export JENKINS_LABELS=${JENKINS_LABELS:-swarm}
 export JENKINS_HOME=${JENKINS_HOME:-$HOME}
 echo "Starting up swarm client with args:"
echo "$@"
echo "and env:"
echo "$(env)"
set -x
 java -jar \
 /home/jenkins_slave/swarm-client-3.4.jar \
  -sslFingerprints '[]' \
  -fsroot "$JENKINS_HOME" \
   -labels "$JENKINS_LABELS" \
   -master http://$JENKINS_IP:$JENKINS_PORT "$@"
```

Uses the host IP as the Jenkins server IP, unless JENKINS_IP was set in the environment of the call to this script

Determines the IP address of the host

Sets the Jenkins port to 8080 by default

Sets the Jenkins label for this slave to "swarm"

Sets the Jenkins home directory to the jenkins_slave user's home by default

Logs the commands run from here as part of the output of the script

Runs the Jenkins Swarm client

Sets the root directory to the Jenkins home directory

Sets the label to identify the client for jobs

Sets the Jenkins server to point the slave at

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 \
-username admin -password adminpassword
```

The `username` and `password` should be an account on your Jenkins instance with permission to create slaves—the `admin` account will work, but you can also create another account for this purpose.

Now that you have a slave set up on this machine, you can run Jenkins jobs on it. 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 67).

> **WARNING** 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 4CPU 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 isn't a technique that can be rolled out without considering performance, we think there's a lot of scope here to turn contributing developers' computer resources into a form of game, increasing efficiency in a development organization without needing expensive new hardware.

DISCUSSION

You can automate this process by setting it up as a supervised system service on all of your estate's PCs (see technique 82).

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

TECHNIQUE 69 Upgrading your containerized Jenkins server safely

If you've used Jenkins for a while in production, you'll be aware that Jenkins frequently publishes updates to its server for security and functionality changes.

On a dedicated, non-dockerized host, this is generally managed for you through package management. With Docker, it can get slightly more complicated to reason about upgrades, as you've likely separated out the context of the server from its data.

PROBLEM

You want to reliably upgrade your Jenkins server.

SOLUTION

Run a Jenkins updater image that will handle the upgrade of a Jenkins server.

This technique is delivered as a Docker image composed of a number of parts.

First we'll outline the Dockerfile that builds the image. This Dockerfile draws from the library Docker image (which contains a Docker client) and adds a script that manages the upgrade.

The image is run in a Docker command that mounts the Docker items on the host, giving it the ability to manage any required Jenkins upgrade.

DOCKERFILE

We start with the Dockerfile.

Listing 8.15 Dockerfile for Jenkins updater

```
FROM docker
 ADD jenkins_updater.sh /jenkins_updater.sh
 RUN chmod +x /jenkins_updater.sh
 ENTRYPOINT /jenkins_updater.sh
```

Uses the docker standard library image

Adds in the jenkins_updater.sh script (discussed next)

Ensures that the jenkins_updater.sh script is runnable

Sets the default entrypoint for the image to be the jenkins_updater.sh script

The preceding Dockerfile encapsulates the requirements to back up Jenkins in a runnable Docker image. It uses the `docker` standard library image to get a Docker client to run within a container. This container will run the script in listing 8.16 to manage any required upgrade of Jenkins on the host.

> **NOTE** If your docker daemon version differs from the version in the `docker` Docker image, you may run into problems. Try to use the same version.

JENKINS_UPDATER.SH

This is the shell script that manages the upgrade within the container.

Listing 8.16 Shell script to back up and restart Jenkins

```
#!/bin/sh
 set -e
 set -x
 if ! docker pull jenkins | grep up.to.date
 then
    docker stop jenkins
     docker rename jenkins jenkins.bak.$(date +%Y%m%d%H%M)
```

This script uses the sh shell (not the /bin/bash shell) because only sh is available on the Docker image

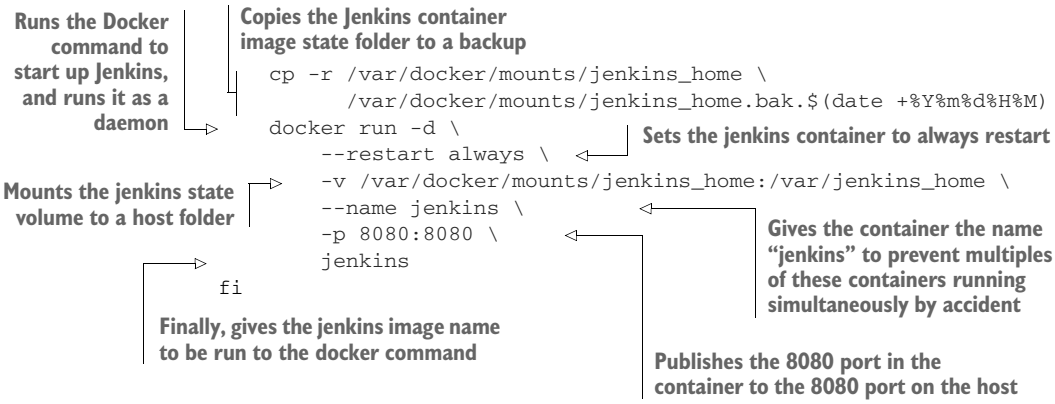
Ensures the script will fail if any of the commands within it fail

Logs all the commands run in the script to standard output

Only fires if "docker pull jenkins" does not output "up to date"

When upgrading, begins by stopping the jenkins container

Once stopped, renames the jenkins container to "jenkins.bak." followed by the time to the minute



The preceding script tries to pull `jenkins` from the Docker Hub with the `docker pull` command. If the output contains the phrase "up to date", the `docker pull | grep …` command returns `true`. But you only want to upgrade when you don't see "up to date" in the output. This is why the `if` statement is negated with a `!` sign after the `if`.

The result is that the code in the `if` block is only fired if you downloaded a new version of the "latest" Jenkins image. Within this block, the running Jenkins container is stopped and renamed. You rename it rather than delete it in case the upgrade doesn't work and you need to reinstate the previous version. Further to this rollback strategy, the mount folder on the host containing Jenkins' state is also backed up.

Finally, the latest-downloaded Jenkins image is started up using the `docker run` command.

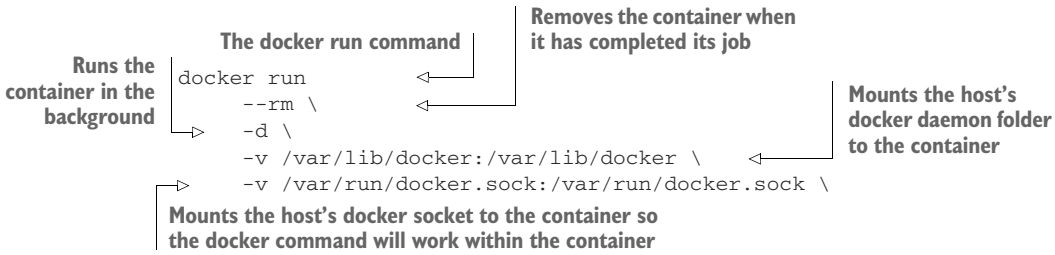
NOTE You may want to change the host mount folder or the name of the running Jenkins container based on personal preference.

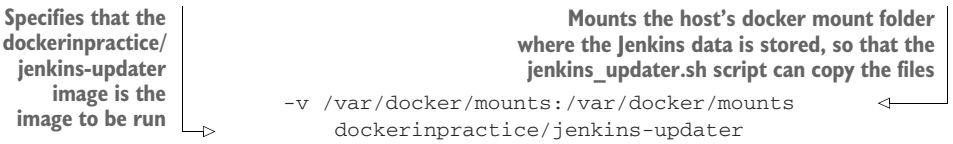
You might be wondering how this Jenkins image is connected to the host's Docker daemon. To achieve this, the image is run using the method seen in technique 66.

THE JENKINS-UPDATER IMAGE INVOCATION

The following command will perform a Jenkins upgrade, using the image (with shell script inside it) that was created earlier:

Listing 8.17 Docker command to run the Jenkins updater





AUTOMATING THE UPGRADE

The following one-liner makes it easy to run within a crontab. We run this on our home servers.

```
0 * * * * docker run --rm -d -v /var/lib/docker:/var/lib/docker -v
➥ /var/run/docker.sock:/var/run/docker.sock -v
➥ /var/docker/mounts:/var/docker/mounts dockerinpractice/jenkins-updater
```

NOTE The preceding command is all on one line because crontab does not ignore newlines if there is a backslash in front in the way that shell scripts do.

The end result is that a single crontab entry can safely manage the upgrade of your Jenkins instance without you having to worry about it.

The task of automating the cleanup of old backed-up containers and volume mounts is left as an exercise for the reader.

DISCUSSION

This technique exemplifies a few things we come across throughout the book, which can be applied in similar contexts to situations other than Jenkins.

First, it uses the core `docker` image to communicate with the Docker daemon on the host. Other portable scripts might be written to manage Docker daemons in other ways. For example, you might want to write scripts to remove old volumes, or to report on the activity on your daemon.

More specifically, the `if` block pattern could be used to update and restart other images when a new one is available. It's not uncommon for images to be updated for security reasons or to make minor upgrades.

If you're concerned with difficulties in upgrading versions, it's also worth pointing out that you need not take the "latest" image tag (which this technique does). Many images have different tags that track different version numbers. For example, your image `exampleimage` might have an `exampleimage:latest` tag, as well as `exampleimage:v1.1` and `exampleimage:v1` tags. Any of these might be updated at any time, but the `:v1.1` tag is less likely to move to a new version than the `:latest` tag. The `:latest` tag could move to the same version as a new `:v1.2` tag (which might require steps to upgrade) or even a `:v2.1` tag, where the new major version `2` indicates a change more likely to be disruptive to any upgrade process.

This technique also outlines a rollback strategy for Docker upgrades. The separation of container and data (using volume mounts) can create tension about the stability of any upgrade. By retaining the old container and a copy of the old data at the point where the service was working, it's easier to recover from failure.

Database upgrades and Docker

Database upgrades are a particular context in which stability concerns are germane. If you want to upgrade your database to a new version, you have to consider whether the upgrade requires a change to the data structures and storage of the database's data. It's not enough to run the new version's image as a container and expect it to work. It gets a bit more complicated if the database is smart enough to know which version of the data it's seeing and can perform the upgrade itself accordingly. In these cases, you might be more comfortable upgrading.

Many factors feed into your upgrade strategy. Your app might tolerate an optimistic approach (as you see here in the Jenkins example) that assumes everything will be OK, and prepares for failure when (not if) it occurs. On the other hand, you might demand 100% uptime and not tolerate failure of any kind. In such cases, a fully tested upgrade plan and a deeper knowledge of the platform than running `docker pull` is generally desired (with or without the involvement of Docker).

Although Docker doesn't eliminate the upgrade problem, the immutability of versioned images can make it simpler to reason about them. Docker can also help you prepare for failure in two ways: backing up state in host volumes, and making testing predictable state easier. The hit you take in managing and understanding what Docker is doing can give you more control over and certainty about the upgrade process.

Summary

- You can use the Docker Hub workflow to automatically trigger builds on code changes.
- Builds can be sped up significantly by using eatmydata and package caches.
- Builds can also be sped up by using proxy caches for external artifacts such as system packages.
- You can run GUI tests (like Selenium) inside Docker.
- Your CI platform (such as Jenkins) can itself be run from a container.
- 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.