The Containerized Jupyter Platform

by Joshua Cook

The Containerized Jupyter Platform

Using Docker Cloud to deploy a secure Jupyter installation to Amazon Web Services

Joshua Cook

This book is for sale at http://leanpub.com/thecontainerizedjupyterplatform. This version was published on April 18, 2016. This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2016 Joshua Cook, Santa Monica, California

Table of Contents

1

Docker

Docker is a system for developing applications such that the local development environment, as well as any possible environment into which we would deploy the application, are identical. On its face, this would seem to be an impossible task. As of 2014, there were 285 actively maintained Linux Distributions and multiple major versions of both OS X and Windows. How could we possibly write a system to allow for all possible development, testing, and production environments?

Docker solves this problem via virtualization. We can not guarantee that our remote environments will be running the same OS as we are locally. We often know for a fact that it never will (I develop using Mac OS X and usually deploy to systems running Ubuntu). That said, we can guarantee that both our development and deployment environments will be able to run the Docker daemon.

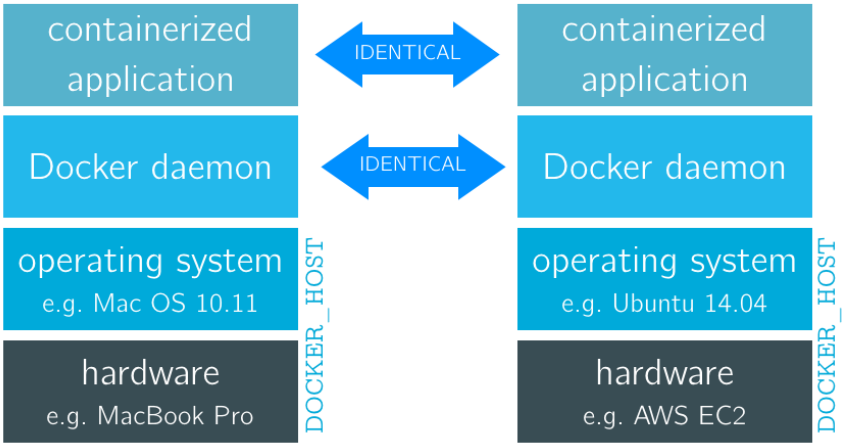


Figure 1.1: Containerized Applications

Using Docker's toolbox we build our applications to run within a Docker container. We can do this locally and test that our applications run on our local Docker daemon. Having confirmed this, it is trivial to deploy our containerized application to a remote machine that is running the same Docker daemon.

the Docker Container Ecosystem

We will begin looking at Docker by focusing on the immediate ecosystem of the container. Later, we will begin to leverage Docker's tools for composing larger systems with the containers we have built, but first things first. In the immediate ecosystem of the Docker container, it is important to keep track of the following concepts:

- the Docker client
- the `DOCKER_HOST` [1]
- the Docker daemon
- the Docker container
- the Docker image
- the Docker registry

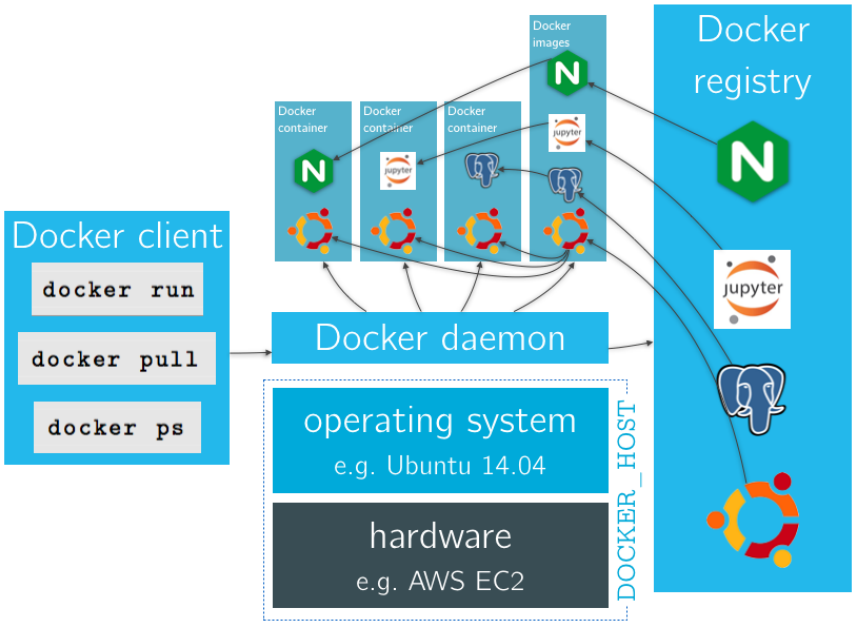


Figure 1.2: the Docker Container Ecosystem

[1] Why is this written in code style? Because it is a value that is often set as an environment variable.

the Docker client The Docker client is an interface used to give instructions to the Docker daemon. This is similar to the client-server architecture of the web, in which a client system uses an interface (typically a web browser, but possibly a RESTful API) to interface with a remote server. In the case of Docker, the Docker client talk to the Docker daemon that performs the work of containers and containerization.

For the vast majority of the work we will do, we will be using the Docker command line interface as our client and the daemon will be running on our local system. Using the client, we tell the daemon to `pull` an image from a registry. We can then tell the daemon to `run` that image. Having done so we might ask the daemon which `ps` or containers are currently running [2]. It is also possible to interface with a Docker daemon via a RESTful API.

the `DOCKER_HOST` The `DOCKER_HOST` is a machine on which we will run the Docker daemon. Locally, the `DOCKER_HOST` will depend upon your computer's operating system. If you are running Mac OS X and Windows, `DOCKER_HOST` will be an instance of `boot2Docker.iso`, an image provided by Docker for this purpose, running as a virtual machine. If you are running Linux, `DOCKER_HOST` will be your machine itself. Remotely, we will set up a machine specifically to serve as `DOCKER_HOST`. The important thing is that while we will always need **A** `DOCKER_HOST` the details of that host are irrelevant. It could be a virtual machine on your mac, a c4.8xlarge EC2 instance on Amazon Web Services, or a bare metal server in your university basement. Regardless, your application will behave the same.

the Docker daemon The Docker daemon is a service running on `DOCKER_HOST`. It does the core work of Docker – building, running, and distributing our Docker containers. We will never interact with the daemon directly but will do so through the Docker client. The power of Docker lies in our ability to work with the daemon. If we can do so locally, we can count on any work we do to behave the same we on a remote machine.

[2] This command is a descendent of the `bash` command `ps`. In `bash`, this command lists running processes, whereas in Docker this command lists running containers. This, fits however, inline with Docker's recommended best practice of one process per container.

Docker images and containers Docker images are "read-only". This is not to say that we can not make changes to an image, but that once we have it is a new kind of image [3]. The Docker engine has several methods for building our own images including the client and via a domain specific language known as the Dockerfile. We can also download images that other people have created.

Docker containers are instances of Docker images. They are stand-alone, containing everything a single instance of an application needs to run (OS, dependencies, source, meta-data) and can be run, started, stopped, moved, and deleted. They are also isolated and secure.

It is helpful to think of Docker images and containers in terms of object-oriented programming. A Docker image is a defined "class" of container that we might create. A Docker container would then be an "instance" of that class. The Docker daemon will manage multiple Docker containers running on `DOCKER_HOST`. When the daemon runs a container from an image, it adds a read-write layer on top of the image in which our application can run.

the Docker registry Docker registries hold images. These are public or private stores from which you upload or download images. For the purposes of this book we will use the public Docker registry at Docker Hub. All of the images we will be using are hosted there.

As we develop in our abilities, we will post our own images to the Docker Hub and ultimately use the Hub to pull our images when we compose larger systems.

[3] I like to think about languages with immutable data structures such as tuples in Python. Once you define a tuple, you can not modify it, though you can define a new tuple that takes the original and modifies it in some way.

Running a Docker image

Run the base ubuntu image and connect to it via shell

```
$ docker run -it ubuntu  /bin/bash
root@eb5f4278d040:/# exit
exit
```

This is sort of an elemental Docker command – run the latest ubuntu image (`run ubuntu`) and connect to it via a bash shell (`-i -t /bin/bash`). When you execute this command, the daemon does the following:

1. Checks for the `ubuntu` image in your local cache of images
2. Downloads the image from Docker Hub, unless the image exists locally
3. Creates a new container using the image
4. Allocates a filesystem and adds a read-write layer to the top of the image
5. Sets up an IP address for the system
6. Executes the process `/bin/bash` within the container
7. Connects us via our current terminal to the running `/bin/bash` process

2

the Blueprint

Overview

In this book, we will use Docker virtualization technology to design, build, and maintain a reusable data science system. Docker's larger ecosystem consists of Containers, Images, Services, Stacks, and Nodes as well as two cloud services we will take advantage of, Docker Hub and Docker Cloud. Using these tools we will build our application, a Jupyter Data Science Platform which will:

- be a computational mathematics platform
- be secured by https
- have available data services Redis and PostgreSQL
- have ORM libraries to access these data services
- have persistent libraries and data stores

Figure 2.1: Our Jupyter Stack

While this is a fairly complicated stack, the beauty of Docker is that the difficult details are abstracted away from us and the implementation is surprisingly straightforward.

We have already discussed Docker images and containers. Three of the containers we will use, those for nginx, Redis, and PostgreSQL, will come directly from the public Docker registry at Docker Hub, though we may make a few slight modifications to these images, especially around data persistence. We will use the Docker client and its build facility to build our own custom Jupyter image based upon the publicly available Jupyter images developed by the Jupyter project. We will then use the `docker-compose` tool to design a Stack consisting of these four images.

Deployment A Node is a cloud-based system running Docker to which we can deploy our containers. We will walk through setting up a Node using Amazon Web Services. Each of the four containers in a our stack will be configured on this Node as a running Service. Finally, we will integrate the work that we did with `docker-compose` with Docker Cloud's `Stackfile` system to deploy our Stack to this Node.

3

Docker Engine

Docker Engine is the core technology upon which we will do our work. For our purposes, we can think of Docker Engine as the Docker daemon and the Docker client we use to give the daemon instructions. Docker as a whole consists of both the Engine and the Hub used to store images. If I have not emphasized this enough, the magic happens because we can count on the Docker Engine to work the same way no matter our underlying hardware (or virtual hardware) and operating system. We build it using Docker Engine, we test it using Docker Engine, we deploy it using Docker Engine.

Get your workstation running Docker

Docker is a rapidly changing technology. As such it is best to refer to Docker's latest instructions on installation. The base installation include each of the core techonogies we will be using – `docker`, `docker-machine`, and `docker-compose`

After installing Docker for your system go to the Post-installation Check.

Installation on MacOS or Windows Refer to the appropriate instructions for mac or windows. For the purposes of this book, you will begin any Docker work by running the Docker Quickstart Terminal application, which:

1. opens a terminal window
2. creates a default VM if it doesn't exist and starts the VM after
3. points the terminal environment to this VM
4. if it worked you will see the whale

Figure 3.1: the Docker Whale

Installation on Linux Distributions

All instructions are available at http://docs.Docker.com under the Docker Engine >> Install tab. Unless you choose to run Docker on a virtual machine, your machine itself will be the `DOCKER_HOST`.

Post-installation Check

As a post-installation check we will verify that we can pull and run the `hello-world` image provided by Docker.

Hello Docker!

```
1 $ docker run hello-world
2 Unable to find image 'hello-world:latest' locally
3 latest: Pulling from library/hello-world
4 03f4658f8b78: Pull complete
5 a3ed95caeb02: Pull complete
6 Digest: sha256:8be990ef2aeb16dbcb9271ddfe2610fa6658d13f6dfb8bc72074cc1ca36966a7
7 Status: Downloaded newer image for hello-world:latest)
```

The first time you run this, it will pull the `hello-world` image from Docker Hub. Thus in line 2, the Docker daemon is informing us that it was unable to locate the `hello-world` image in our local cache of images. In line 3, it tells us that it is pulling the image from library/hello-world. Lines 4 and 5 describe the progress of pulling the two layers that comprise the `hello-world` image to our local machine. Line 6 gives us the SHA associated with the version of the image we downloaded. When this is complete, line 7 updates the status informing us that a newer image than we previously had for `hello-world` has been pulled. If successful you should see the following:

Output

```
Hello from Docker.
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to
    your terminal.
```

Quickstart

Assume you have completed installation of Docker (`core`, `machine` and `compose`) for your system.

The Docker command line interface

```
$ docker
Usage: docker [OPTIONS] COMMAND [arg...]
       docker [ --help
```

Running the command `docker` alone returns usage. If you are ever in doubt as to which commands can be run and which arguments they require, try runnning the command with no arguments.

`docker info`

```
$ docker info
Containers: ...
```

We can display system-wide information using this command. Here we use it is a minimal verification of a working Docker installation.

Download a pre-built ubuntu image from the Docker registry

```
$ docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
...
Digest: sha256:f6e8757419147ba099af8cec4db365b3603472bd6182722b16fdae932c0bf3bf
Status: Downloaded newer image for ubuntu:latest
```

When we pulled `hello-world` image, we did so implicitly as part of the `docker run` command. Here we explicitly pull the image. As we did not specify a tag for pulling, the Docker daemon defaults to using the tag `latest`. The Docker daemon then finds the `latest ubuntu` image by name on Docker Hub and downloads the image to a local image cache. The `ubuntu` image consists of four layers, each of which is pulled in parallel.

Run an interactive shell to an ubuntu container

```
$ docker run -i -t ubuntu /bin/bash
root@b7549b176c6d:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
```

If we wish to run an interactive shell to the ubuntu image we downloaded, we can do so via application of the `run` command. The `-i` flag starts an interactive container. The `-t` flag creates a pseudo-TTY [1] that attaches `stdin` and `stdout`. You should receive a shell prompt for a shell attached to the ubuntu image.

It is somewhat abstract what exactly it is we have done. We have launched an instance of the `ubuntu` image. The docker image then created a temporary layer on top of this image that we can interact with. The underlying image and this read/write layer on top of that image are a Docker container.

The Docker engine then connected us to the container via a bash shell. Here we have root access to that running container. The container has the container id `b7549b176c6d`. We ran the `ls` command and displayed, in this case, the directories in the root directory (‘`/`’) of the machine.

When we close the bash, the Docker engine will kill the container. The read/write layer will continue to exist in our cache but it is not currently connected to the underlying image. If we launch a new container by running the same command again, the Docker engine will create a new read/write layer.

[1] command line interpreter or shell

Next Steps

Hello World in a Container

`docker run` is used to run an application as a container. Let's walk through this.

Display Docker containers

```
$ docker ps
CONTAINER ID      IMAGE      COMMAND      CREATED      STATUS      PORTS      NAMES
```

First, take a look at all Docker containers currently *running*. You should see the headers of an empty table signifying that there are no containers currently running.

Display Docker images

```
$ docker images
REPOSITORY          TAG             IMAGE ID            CREATED             SIZE
ubuntu              latest          b549a9959a66        31 hours ago        188 MB
hello-world         latest          690ed74de00f        5 months ago        960 B
```

Next, show all images. If you have been following along previously you should have two. The significance of these two images is that they are stored locally. We can `run` either of them instantly. Let's run the `hello-world` instance.

Run the `hello-world` image

```
$ docker run hello-world
Hello from Docker.
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to
    your terminal.
```

Previously, we ran this command and had to wait while the images was pulled from the registry. Now it runs immediately and shows the same response.

Display currently running containers

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
```

Display currently running containers to confirm that the `hello-world` container shut down again. This shows that there is no container running. This means that the `hello-world` image launched, displayed its message, then shut down again.

Run `echo` as a service

```
$ docker run ubuntu /bin/echo 'Hello World!'
Hello World!
```

Here we `echo` a 'Hello World!' as a service. In the Docker ecosystem, a service is a process that has been "containerized". As this is in the title of this book, as you might imagine, this is critical to what we are attempting to do. Crack your knuckles. This is where we are officially getting started. Say, "Hello World!" as you pass a command to be executed by our container.

Display currently running containers

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
```

Display currently running containers to confirm that the `ubuntu` container shut down again. Again, no machines are running. The ubuntu image launched, used the `echo` command to echo 'Hello World!', then shut down again. The gravity of this may not be apparent. To fully comprehend what is happening, let's time the whole thing.

Timing the ubuntu image bootstrap

Time the execution of the `echo` service

```
$ time docker run ubuntu /bin/echo 'Hello World!'

Hello World!

real  0m0.339s
user  0m0.026s
sys   0m0.023s
```

Use the built-in `time` command to see how long it takes to echo a string as a service.

Time the execution of `echo` natively

```
$ time echo 'Hello World!'

Hello World!

real  0m0.006s
user  0m0.000s
sys   0m0.001s
```

Now, we use the built-in `time` command to see how long it takes to do the same natively, that is, directly on our machine.

That is a pretty significant jump. Running the command in the Docker instance made the task take 60 times longer. But consider this from another perspective. Instead consider the fact that booting up an entire ubuntu instance only added .330s to our processing time.

Attempting to isolate the bootstrap time

Time a hard `sleep` natively

```
$ time sleep 2

real  0m2.015s
user  0m0.001s
sys   0m0.003s
```

Let's try this experiment once more with a process for which we can guarantee a certain amount of time.

Time a hard `sleep` as a service

```
$ time Docker run ubuntu /bin/sleep 2

real  0m2.418s
user  0m0.027s
sys   0m0.031s
```

Note that we see approximately the same increase. This signifies that "containerizing" our service adds about .4 seconds to our runtime, significant when `echo`ing "Hello World!" negligible when doing much else.

A Daemonized Hello World

We will want to daemonize our Docker containers, that is set them up to run indefinitely. We do this by running our containers in **detached** mode via the `-d` flag.

Run the ubuntu image in detached mode

```
$ docker run -d ubuntu /bin/sh -c "while␣true;␣do␣echo␣hello␣world;␣sleep␣1;␣done"
```

Here we run `ubuntu` in detached mode, that is, run the image as a container and leave it running. As we gave it a job to do, while it is running it will `echo` 'hello world' every second ad infinitum.

Show running containers

```
$ docker ps
CONTAINER ID    IMAGE     COMMAND                  ...    PORTS     NAMES
720ba4ef7858    ubuntu    "/bin/sh␣-c␣'while␣tr"   ...               reverent_borg
```

Show running containers to confirm that the container **is** running. Because we passed the `run` command the `-d` flag for detached mode, it should still be running.But how do we confirm that it is doing the job it has been tasked with?

Show logs for a detached container

```
$ docker logs reverent_borg
hello world
hello world
hello world
hello world
...
```

We view logs for our container currently running in detached mode. Not the most exciting log, but sufficient to confirm that our container is doing its job.

Stop a detached container

```
$ docker stop reverent_borg
```

Let's give it some rest, using the `stop` command to shut it down.

4

Jupyter

Jupyter is a web-based interactive environment combining live code, markdown- and latex-rich text, images, plots and more in a single document. Jupyter is the successor to the IPython notebook and was renamed as the platform supported more and more software kernels. As it began with support for Julia, Python, and R, it was renamed as JuPyteR, though now the platform supports Scala, Haskell, Ruby, amongst many others. A list of all kernels supported can be found here.

Jupyter provides opinionated stacks for use in a variety of contexts: the All Spark stack, the minimal stack. Let's take a look at the demo stack. The demo stack can be tried immediately by visiting http://tmpnb.org and is a fully-functioning jupyter system with multiple kernels - Bash, Haskell, Julia, Python 2 and 3, R, Ruby, and Scala - preinstalled.

Jupyter Demo Stack

As a way to dive deeper into both Docker and Jupyter lets bring the Jupyter Demo Stack online in our local environment. If you are on a linux box running the Docker daemon natively, you will be able to access any exposed (we'll get to that in a minute) Docker containers by visiting http://localhost in a browser at the appropriate port. If you have been running the Docker daemon on a virtual machine, you will need to get the IP of the virtual host and access your machine there instead. If your Docker virtual machine is named `default` you can find this IP by using the `Docker-machine` command.

Get the ip of your `DOCKER_HOST`

```
$ docker-machine ip default
192.168.99.100
```

This signifies that my Docker virtual machine is available at 192.168.99.100. Your machine may be available at a different IP address.

Pull the Jupyter Demo from Docker Hub

```
$ docker pull jupyter/demo
Using default tag: latest
latest: Pulling from jupyter/demo
...
Digest: sha256:d3dd87e52ca1edbfc8b65ad68bfa91f15eb0660d218c64fd5cdb039c1fa10818
Status: Downloaded newer image for jupyter/demo:latest
```

To launch an image we will first pull it from Docker Hub. We could have done this implicitly via the `run` command but running it explicitly gives greater insight into what we are doing.

Run the Jupyter Demo

```
$ docker run -p 80:8888 jupyter/demo
[I 01:11:44.619 NotebookApp] Writing notebook server cookie secret to
    /home/jovyan/.local/share/jupyter/runtime/notebook_cookie_secret
[W 01:11:44.990 NotebookApp] WARNING: The notebook server is listening on all
IP addresses and not using encryption. This is not recommended.
[W 01:11:44.990 NotebookApp] WARNING: The notebook server is listening on all
IP addresses and not using authentication. This is highly insecure and not
    recommended.
[I 01:11:45.059 NotebookApp] Serving notebooks from local directory:
    /home/jovyan/work
[I 01:11:45.060 NotebookApp] 0 active kernels
[I 01:11:45.060 NotebookApp] The IPython Notebook is running at: http://[all ip
    addresses on your system]:8888/
[I 01:11:45.060 NotebookApp] Use Control-C to stop this server and shut down all
    kernels (twice to skip confirmation).
```

We run the `jupyter/demo` image exposing the port 8888 on the `jupyter/demo` Docker container to port 80 on the Docker host. Port 80 is the port most commonly used by HTTP and in this case means that we can access our box by visiting either `localhost` or the the IP identified when you ran `Docker-machine` without appending a port e.g. 192.168.99.100. The vast majority of the output we see here is output generated by the Jupyter application and written to the standard logger. As we are running the container in **foreground** mode, the output is written to our terminal.

the Jupyter File System

Visiting the application in our browser using the IP address of our `DOCKER_HOST` we first see the main Jupyter File System.

Figure 4.1: Jupyter File System

We can launch a Jupyter file by clicking any of the files in the File System or launch a new file via the "New" menu in the upper right corner.

A Jupyter File Open the "Welcom to Python" file by clicking the file. After a brief warning about using the file on the Free Hosting on Rackspace – which we can ignore as we are hosting it ourselves – we see the barest of instructions on executing code in a Jupyter file.

Run some Python code!

To run the code below:

1. Click on the cell to select it.
2. Press `SHIFT+ENTER` on your keyboard ...

In[1]:

```
%matplotlib notebook

import pandas as pd
import numpy as np
import matplotlib

from matplotlib import pyplot as plt
import seaborn as sns

ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000',
    periods=1000))
ts = ts.cumsum()

df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index,
                  columns=['A', 'B', 'C', 'D'])
df = df.cumsum()
df.plot(); plt.legend(loc='best')
```

Out[1]:

```
<matplotlib.legend.Legend at 0x7f73cc716b38>
```

Shut down the Jupyter application

Shut down the machine by pressing `ctrl-c`.

Exposing ports

Let's take a brief look at exposing ports. We will also launch our container in **detached** mode via the `-d` flag.

Run the `jupyter/demo` image in detached mode

```
$ docker run -d -p 5000:8888 jupyter/demo
2040f677ad7ffa4666d0d9826e00175a15315ae2b2422314924f6022d6b65622
```

We run the `jupyter/demo` image in detached mode exposing port 8888 to port 5000 on the `DOCKER_HOST`. This will expose the port 8888 on the `jupyter/demo` to port 5000 on the Docker host. We can access the machine by visiting `localhost:5000` or 192.168.99.100:5000 in order. Here we are passed the `CONTAINER_ID` of our container and returned to our terminal.

As we are running the container in **detached** mode. The Jupyter logger's output is not written to our terminal. As before, we can access the output via the `logs` command.

Display containers currently running

```
$ docker ps
CONTAINER ID  IMAGE          COMMAND      PORTS                   NAMES
2040f677ad7f  jupyter/demo   "tini␣..."   0.0.0.0:80->8888/tcp    furious_archimedes
```

Note that in addition to assigning the container a `CONTAINER ID`, the Docker daemon also assigned us a name `furious_archimedes`.

Display logs for a container running in detached mode

```
$ docker logs furious_archimedes
[I 15:38:33.350 NotebookApp] Writing notebook server cookie secret to
    /home/jovyan/.local/share/jupyter/runtime/notebook_cookie_secret
...
```

Examine the port of a running container

```
$ docker port furious_archimedes
8888/tcp -> 0.0.0.0:5000
```

We look at the port mappings for our container with the `port` command. This signifies that port 8888 on our container is mapped via the TCP protocol to port 5000 on **its** localhost (0.0.0.0). Note that this refers to the machine on which the Docker daemon is running. If you are running Docker on a virtual machine then this machine will be at localhost for your **container** but will have an IP on your **system** . We accessed this earlier via `docker-machine ip default` where `default` was the name we had assigned to our virtual machine. From the persepective of the Docker daemon we are accessing the container's 8888 port via localhost:5000, but if we running a virtual machine we are accessing it at 192.168.99.100:5000.

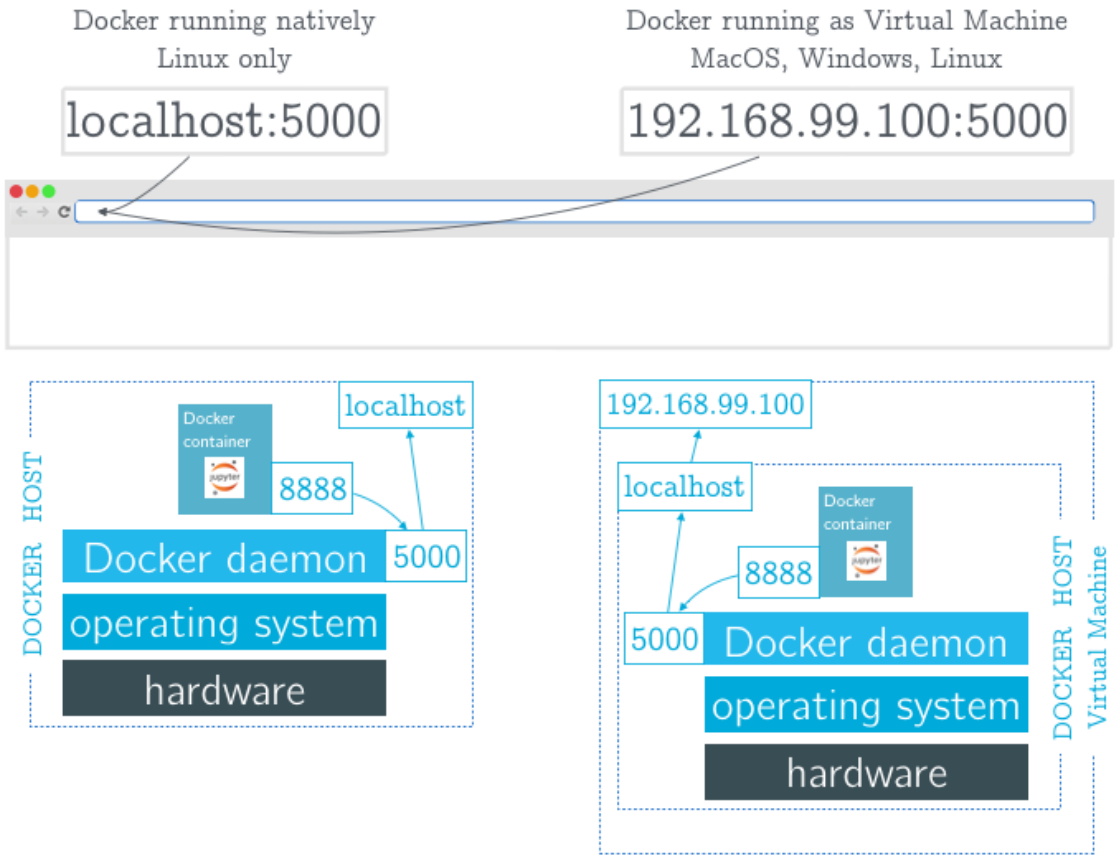


Figure 4.2: Exposing Ports

Data Persistence

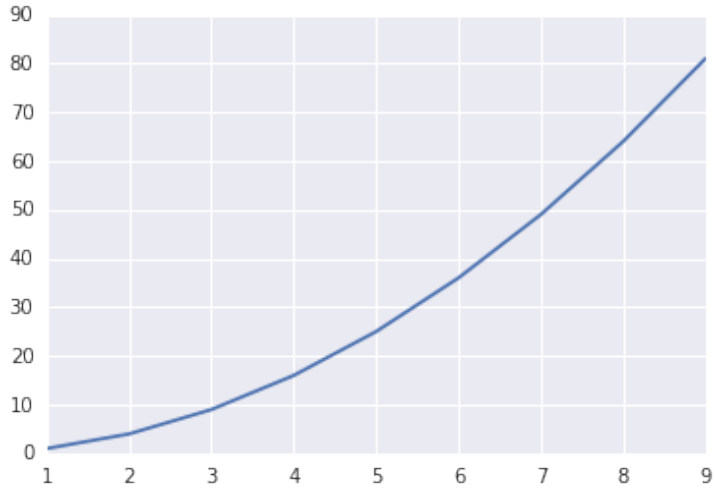
Before we dig further into building our platform, let us for the first time consider the topic of persistence. Visit the Jupyter application in your browser. Once there create a new Python file and run a basic calculation on that file.

In[1]:
```
%matplotlib notebook

import numpy as np
import matplotlib.pyplot as plt
```

In[2]:
```
x = np.arange(1,10,1)
f = lambda x: x**2
y = f(x)
```

In[3]:
```
plt.plot(x,y)
```



Out[3]:
```
<matplotlib.lines.Line2D at 0x7f73cc269978>
```

Rename the file, save your changes, then choose ‘Close and Halt’ from the file menu. Returning to the Jupyter File System, you should see the file you just created.

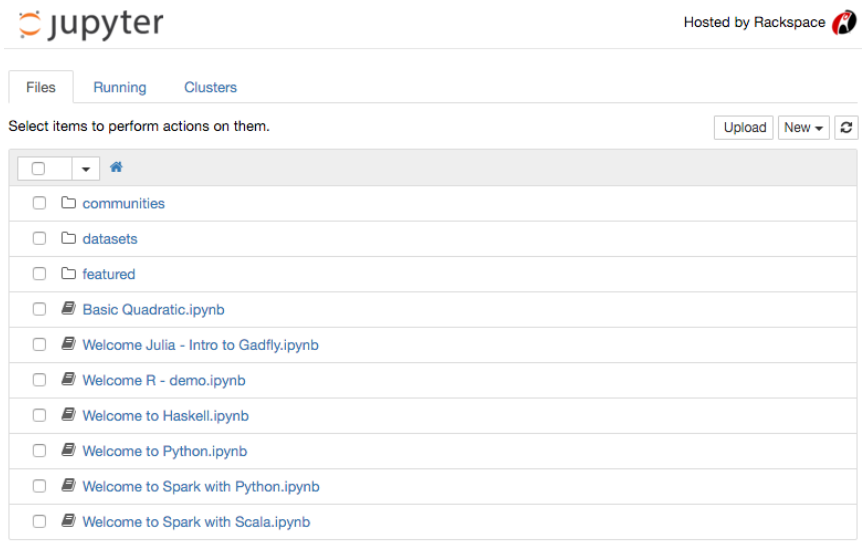


Figure 4.3: Our File in the Jupyter File System

Shut down a running container

```
$ docker stop furious_archimedes
furious_archimedes
$ docker ps
CONTAINER ID        IMAGE       COMMAND       CREATED       STATUS       PORTS       NAMES
```

We shut down your running instance and confirm it is no longer running.

Run the `jupyter/demo` image in detached mode

```
$ docker run -d -p 5000:8888 jupyter/demo
5999d158488d410ac5fbf3a646e4a962d307e968d3cd2f53e60e0a0c7bbe262c
```

Now start an image up again and visit the application in your browser. This file we just created is gone. Data has not persisted from instance to instance. This is a problem.

Attach a Volume This problem can be solved via a run argument. We can attach a volume via a run argument with the flag `-v`. Note that if you are running docker on a virtual machine, as of the writing of this book, you are only able to mount volumes from `/Users` (OS X) or `C:Users` (Windows).

We pass the `-v` flag a single argument that consists of `LOCAL_DIR:CONTAINER_DIR`. The Jupyter Demo stack is serving files from `/home/jovyan/work`. We will serve files in `~/src` to `/home/jovyan/work/src`. Note that you must use the absolute path i.e. `/Users/joshuacook/src`.

Run an image and attach a volume

```
$ docker run -v /Users/joshuacook/src:/home/jovyan/work/src \
    -d -p 5000:8888 jupyter/demo
273ff71c6755670e21accd197461dd4256fbeb129393d137733f36bcb5432a55
```

Run in detached mode and attach a volume. Repeat the above experiment with regard to file persistence. You should notice three things.

1) All files in `~/src` should be immediately available to your Jupyter application.
2) Any files that are written into the `src` directory on the containerized Jupyter application should be written to `~/src` on your machine.
3) Because of both of these files should persist from launch to launch.

5

the Dockerfile

Every Docker image is built as a series of layered abstractions. The first layer might be the virtual machine's operating system - a Debian or Ubuntu Docker image, the next the installation of dependencies necessary for your application to run, and all the way up to the source code of your application. Because each of these is built as a separate abstraction layer, changes to the Docker image are made from the top down when rebuilding the image. If you have only made changes to the code sitting on the top most layer, this is the only layer of the image that will be rebuilt. As a result, builds progress significantly faster. In this author's humble opinion the best way to leverage this system is via the Dockerfile.

Best Practices

A Dockerfile is a way to script the automated building of an image. This script uses a domain specific language to tell the Docker daemon how to sequentially build a Docker image.

Docker holds the following as best practices when creating Dockerfiles:

- Containers should be ephemeral, in that they can be reinstantiated with a minimum of set-up and configuration
- Use a `.dockerignore` file, similar to a `.gitignore`
- Avoid installing unnecessary packages
- Run only one process per container
- Minimize the number of layers
- Sort multi-line arguments

Preparing for the Automated Build Process

As we will be using Docker's Github integration and its automated build capability to ultimately use our image, we will need to configure and use `git` to interface with Github.

```
$ mkdir Docker
$ mkdir Docker/base && cd Docker/base
$ git init
$ touch Dockerfile
$ git add Dockerfile
$ git commit -m 'init'
```

Configure Github

On github, create a repo called `base`.

```
$ git remote add origin git@github.com:#USERNAME#/base.git
$ git push -u origin master
```

Building images using Dockerfiles

Throughout the process we will make heavy use of the `docker build` command. The `docker build` command tells the Docker daemon to construct an image using the specified context and Dockerfile.

Running a Docker build

```
$ docker build -t joshuacook/base  .
sending build context to Docker daemon 53.25 kB
Error response from daemon: The Dockerfile (Dockerfile) cannot be empty
```

This basic command tells the daemon to build an image using the current directory, specified by `.`, as context. The `-t` flag tells the daemon to name that image `joshuacook/base`.

Dockerfile Syntax

```
# Comment
INSTRUCTION arguments
```

Dockerfiles are built using a simple domain specific language. Instructions are case-insensitive but by convention are written in all-caps. Instructions are passed sequentially and Dockerfiles should be thought of as scripts passed to the Docker daemon.

Build `base` image

We could simply use the images provided by Jupyter. Building our images ourselves, however, is fairly straightforward and a worthwhile exercise in working with images and Dockerfiles. Open the Dockerfile in your favorite text editor.

Begin the image with `FROM`, `ARG` and `MAINTAINER`

the `FROM` instruction

```
FROM debian
```

We begin our image with the `FROM` instruction, which sets the **Base Image** upon which we will build our image. A valid Dockerfile requires a `FROM` instruction as its **first** instruction. We will take our cues from the Jupyter project and build use the `debian` image.

Run the build

```
$ docker build -t joshuacook/base .
Sending build context to Docker daemon 53.76 kB
Step 1 : FROM debian
 ---> 47af6ca8a14a
Successfully built 47af6ca8a14a
```

Here we have built an image with a single layer – our underlying operating system.

Show cached images

```
$ docker images
REPOSITORY          TAG         IMAGE ID          CREATED           SIZE
debian              latest      47af6ca8a14a      5 days ago        125.1 MB
joshuacook/base     latest      47af6ca8a14a      5 minutes ago     125.1 MB
...
```

Take a look at the images we have in cache. Note that the `debian` image and the our `base` image have the exact same `IMAGE ID`. This is because they **are the same image** .

Commit changes

```
$ git add Dockerfile
$ git commit -m 'Added FROM instruction'
```

It is a best practice to commit our changes before we move on.

the `MAINTAINER` instruction

```
MAINTAINER Joshua Cook <me@joshuacook.me>
```

While the `MAINTAINER` instruction is not required, it is conventional.

the `ARG` instruction

```
ARG DEBIAN_FRONTEND=noninteractive
```

The `ARG` instruction is used to define a variable that is available at runtime. Here we use it to define the behavior of our interaction with our Debian virtual machine. We could also have defined an `ARG` with no value and then passed the value to the build as an argument e.g. `DEBIAN_FRONTEND=noninteractive docker build`.

The Jupyter image defined by the Jupyter development team used the `ENV` instruction to achieve the same purpose

the `ENV` instruction

```
ENV DEBIAN_FRONTEND noninteractive
```

After reading this thread on the Docker team's Github page, I chose to use the `ARG` instruction. Either will work but it is important to **only use one of these** .

Run the build

```
$ docker build -t joshuacook/base .
Sending build context to Docker daemon 53.76 kB
Step 1 : FROM debian
 ---> 47af6ca8a14a
Step 2 : MAINTAINER Joshua Cook <me@joshuacook.me>
 ---> Using cache
 ---> b82accbca338
Step 3 : ARG DEBIAN_FRONTEND=noninteractive
 ---> Using cache
 ---> d48a7b1deb9e
Successfully built d48a7b1deb9e
```

Now, we're getting somewhere. We have added three lines to our Dockerfile. When we ran the build, it either 1) used a pre-existing layer as in the case at Step 1 or 2) created a new layer containing the results of having run that step on the previous layer.

Commit changes

```
$ git add Dockerfile
$ git commit -m 'Added MAINTAINER and ARG instruction'
```

Provision `base` image

Ultimately, our Jupyter application will run on a virtualized Debian machine. As with any application, it will have many dependencies, or system-level applications, that the Debian system must be able to execute in order to function properly. Were we building this system manually, we would use the command line package manager appropriate to our Operating System `apt-get` for Ubuntu or Debian, `yum` for CentOS, or even `brew` for Mac OS X in a process known as provisioning.

the `RUN` instruction

```
RUN <command>
```

The Docker daemon has no native way of provisioning, but rather has a mechanism for leveraging the base system's package manager via the `RUN` instruction, which has the effect of running `<command>` in a shell to our base image i.e. `/bin/sh -c`. Similar to scripting in a shell language we can add a `\` to the end of a line to continue that command on the next line. In other words, the following two statements are equivalent

Equivalent `RUN` instructions

```
RUN /bin/bash -c 'source $HOME/.bashrc ;\
  echo $HOME'
RUN /bin/bash -c 'source $HOME/.bashrc ; echo $HOME'
```

Our First Provision Statement

```
RUN apt-get update && \
    apt-get install -yq --no-install-recommends \
    build-essential \
    bzip2 \
    ca-certificates \
    emacs \
    git \
    jed \
    libsm6 \
    libxrender1 \
    locales \
    pandoc \
    python-dev \
    sudo \
    texlive-latex-base \
    texlive-latex-extra \
    texlive-fonts-extra \
    texlive-fonts-recommended \
    texlive-generic-recommended \
    unzip \
    vim \
    wget \
    && apt-get clean && \
    rm -rf /var/lib/apt/lists/*
```

At this point we actually have some meat to our image. Let's go ahead and build it. Save your changes and run the build command from within the `base` directory.

Run the build

```
docker build -t base .
Sending build context to Docker daemon 50.18 kB
...
```

The build will take some time. For now we will hold off on examining the output. Running a Docker build is idempotent, meaning that no matter how many times we run it, we receive the same output. We will wait for this run to complete and then run the build once more to receive a compact output.

Idempotently run the build

```
$ docker build -t joshuacook/base .
Sending build context to Docker daemon 54.27 kB
Step 1 : FROM debian
 ---> 47af6ca8a14a
Step 2 : MAINTAINER Joshua Cook <me@joshuacook.me>
 ---> Using cache
 ---> b82accbca338
Step 3 : ARG DEBIAN_FRONTEND=noninteractive
 ---> Using cache
 ---> d48a7b1deb9e
Step 4 : RUN apt-get update &&    ...
 ---> Using cache
 ---> 0e96a5b5d7d1
Successfully built 0e96a5b5d7d1
```

When it completes, we run the build once more in order to see a collapsed view of the image we just built. The build output contains four steps representing instruction from our Dockerfile. The way that Docker works, it creates or uses a cached image for each step. At Step 1, we tell the Docker daemon which image we will be using as our base image `FROM debian`. After each step, we have a new layer and thus a new image and `IMAGE ID`

```
$ docker images
REPOSITORY        TAG       IMAGE ID        CREATED             SIZE
debian            latest    47af6ca8a14a    8 days ago          125.1 MB
joshuacook/base   latest    0e96a5b5d7d1    About a minute ago  1.636 GB
...
```

Note that the final layer's `ID` matches the `IMAGE ID` of our `base` image.

Commit changes

```
$ git add Dockerfile
$ git commit -m 'Provisioning debian machine'
```

Automatic Build via Docker Hub

To deploy our image as a container on a remote machine, we will be passing the description of that image via Docker Hub. This will be identical to the way we have pulled the official images for `ubuntu` or `debian`, but rather we will be pulling from our own repository of images. Here, we will configure Docker Hub to watch a Github repo for changes in order to trigger an automatic build.

Set up Docker Hub Log into to Docker Hub in order to configure an automated build. Select "Create Automated Build". Given an option between Github and BitBucket, choose Github. After authenticating your Github account, you should be shown a list of available repositories on your Github account. Choose the previously created repo, `base`.

Docker will automatically name the build after the associated repo. Your image will have the name `#DOCKER_USER#/base`. Give the image a short description such as "Provisioned debian image". Click "Create". Upon this, the automated build should be created and you will be taken to the page on Docker Hub for your image.

Push to Github and Initiate an automated build

```
$ git add Dockerfile
$ git commit -m 'initial build of Docker Image'
$ git push
```

In the local repo, stage, commit, and push the changes to your Dockerfile to Github.

Check Build Details Upon completing this push to Github, return to your image's page at Docker Hub. Click the link Build Details. You should see that a build has been initiated. In less than a half hour, the build should complete. If the build was successful, our image will now be available for cloud deployment.

Anaconda

Anaconda is a freemium distribution of the Python and R programming languages for large-scale data processing, predictive analytics, and scientific computing. We will use Anaconda to drive our Jupyter platform. The `conda` command is the primary interface for managing Anaconda installations. Miniconda is a small "bootstrap" version that includes only `conda` and `conda-build`, and installs Python. Here we install Miniconda. We have taken this command directly from the Continuum Analytics Docker image for miniconda3.

Install `miniconda3`

```
RUN echo 'export PATH=/opt/conda/bin:$PATH' > /etc/profile.d/conda.sh && \
    wget --quiet
        https://repo.continuum.io/miniconda/Miniconda3-4.0.5-Linux-x86_64.sh && \
    /bin/bash /Miniconda3-4.0.5-Linux-x86_64.sh -b -p /opt/conda && \
    rm Miniconda3-4.0.5-Linux-x86_64.sh
```

Run a build

```
$ docker build -t joshuacook/base .
...
Step 5 : RUN echo 'export PATH=/opt/conda/bin:$PATH' > /etc/profile.d/conda.sh
...
```

Initiate an automated build

```
$ git add Dockerfile
$ git commit -m 'install miniconda3'
$ git push
```

tini

I think it a bit beyond of the progress we have made to this point to begin a discussion of the `PID 1` "Zombie" problem at this point. The text will address this at a later point. The short of it is, Docker best practices are that we run a single process per container. This includes the typical `systemd` or `sysvinit` that would be run by our operating system to handle processes and signals. This can lead to containers that can't be gracefully stopped or zombie containers that persist when they should have died.

For now, let us simply handle this situation the way that the Jupyter development team does and use `tini`. `tini` is a lightweight solution to this problem with no additional dependencies. It reaps zombies, spawns a single process (which will run as `PID 1`), and when the first process `tini` spawns exits, `tini` exits as well.

In order to include `tini` in our container, we use a command we haven't seen before, `ADD`. `ADD` minimally takes a single argument, a source. We point `ADD` at the Github account serving the `tini` binary.

We then `RUN chmod +x` on the `tini` binary so that it can be executed.

ADD tini

```
ARG TINI_VERSION=v0.9.0

ADD https://github.com/krallin/tini/releases/download/${TINI_VERSION}/tini /tini
RUN chmod +x /tini
```

Run the build

```
$ docker build -t joshuacook/base .
...
Step 6 : ARG TINI_VERSION=v0.9.0
...
Step 8 : RUN chmod +x /tini
...
```

Initiate an automated build

```
$ git add Dockerfile
$ git commit -m 'install tini'
$ git push
```

Set the locale

```
RUN apt-get update -qq
  && apt-get install -y locales -qq
  && locale-gen en_US.UTF-8 en_us
  && dpkg-reconfigure locales
  && dpkg-reconfigure locales
  && locale-gen C.UTF-8
  && /usr/sbin/update-locale LANG=C.UTF-8

ENV LANG C.UTF-8
ENV LANGUAGE C.UTF-8
ENV LC_ALL C.UTF-8
```

A best practice in configuring our machine. Here we use two instructions we have seen before, `RUN` and `ENV`.

Run the build

```
$ docker build -t joshuacook/base .
Sending build context to Docker daemon    64 kB
...
Step 9 : RUN locale-gen en_US.UTF-8
...
Step 12 : ENV LC_ALL en_US.UTF-8
...
```

Initiate an automated build

```
$ git add Dockerfile
$ git commit -m 'Set locale'
$ git push
```

ENTRYPOINT

Ultimately, we provide an `ENTRYPOINT` for our container. `ENTRYPOINT` specifies the process(es) to be launched when the image is instantiated. Here, we tell Docker to run `tini` as `PID 1`.

Set the `ENTRYPOINT`

```
ENTRYPOINT ["/tini", "--"]
```

Run the build

```
$ docker build -t joshuacook/base .
Sending build context to Docker daemon    64 kB
...
Step 13 : ENTRYPOINT /tini --
 ---> Running in b7876a28ae08
 ---> 408463887d9c
Removing intermediate container b7876a28ae08
Successfully built 408463887d9c
```

Initiate the automated build

```
$ git add Dockerfile
$ git commit -m 'Provide entrypoint.'
$ git push
```

Our final `base` Dockerfile

base/Dockerfile

```
FROM debian

MAINTAINER Joshua Cook <me@joshuacook.me>

ARG DEBIAN_FRONTEND=noninteractive

RUN apt-get update && \
  apt-get install -yq --no-install-recommends \
  build-essential \
  bzip2 \
  ca-certificates \
  emacs \
  git \
  jed \
  libsm6 \
  libxrender1 \
  locales \
  pandoc \
  python-dev \
  sudo \
  texlive-latex-base \
  texlive-latex-extra \
  texlive-fonts-extra \
  texlive-fonts-recommended \
  texlive-generic-recommended \
  unzip \
  vim \
  wget \
  && apt-get clean && \
  rm -rf /var/lib/apt/lists/*

RUN echo 'export PATH=/opt/conda/bin:$PATH' > /etc/profile.d/conda.sh && \
    wget --quiet
        https://repo.continuum.io/miniconda/Miniconda3-4.0.5-Linux-x86_64.sh && \
    /bin/bash /Miniconda3-4.0.5-Linux-x86_64.sh -b -p /opt/conda && \
    rm Miniconda3-4.0.5-Linux-x86_64.sh

ARG TINI_VERSION=v0.9.0
ADD https://github.com/krallin/tini/releases/download/${TINI_VERSION}/tini /tini
RUN chmod +x /tini

RUN apt-get update -qq
  && apt-get install -y locales -qq
  && locale-gen en_US.UTF-8 en_us
  && dpkg-reconfigure locales
  && dpkg-reconfigure locales
  && locale-gen C.UTF-8
  && /usr/sbin/update-locale LANG=C.UTF-8

ENV LANG C.UTF-8
ENV LANGUAGE C.UTF-8
ENV LC_ALL C.UTF-8

ENTRYPOINT ["/tini", "--"]
```

6

A Brief Cleanup Interlude

At this point, you should have quite a bit of information in the cache of your `DOCKER_HOST`. Before we continue, let's take a moment to look at keeping our system clean.

Show cached images

```
$ docker images
REPOSITORY          TAG     IMAGE ID      CREATED            SIZE
joshuacook/base     latest  0e96a5b5d7d1  About an hour ago  1.769 GB
ubuntu              latest  b549a9959a66  6 days ago         188 MB
debian              latest  47af6ca8a14a  8 days ago         125.1 MB
hello-world         latest  690ed74de00f  6 months ago       960 B
```

We have four images in our local cache. We do not need all of these. Let's remove `ubuntu` and `hello-world`.

Remove `ubuntu` and `hello-world` images

```
docker rmi ubuntu hello-world
Untagged: ubuntu:latest
Deleted: ...
Untagged: hello-world:latest
Deleted: ...
```

We also have many containers in our system that are no longer running. We can see **all** containers by running `docker ps` with the `-a` flag.

Show all containers.

```
$ docker ps -a
CONTAINER ID    IMAGE     COMMAND       ...   PORTS    NAMES
c4536a97d700    debian    "/bin/bash"   ...            serene_dubinsky
cacae17cf192    debian    "/bin/bash"   ...            desperate_fermat
278b2587dae0    debian    "/bin/bash"   ...            thirsty_mcclintock
...
```

Working with Containers

If you haven't quite grasped the concept of Docker images and Docker containers, it might be helpful to take a moment and examine these two lists.

Show all containes and images

```
$ docker ps -a
CONTAINER ID    IMAGE     COMMAND       ...   PORTS    NAMES
c4536a97d700    debian    "/bin/bash"   ...            serene_dubinsky
cacae17cf192    debian    "/bin/bash"   ...            desperate_fermat
278b2587dae0    debian    "/bin/bash"   ...            thirsty_mcclintock
...
$ docker images
REPOSITORY         TAG      IMAGE ID      CREATED             SIZE
joshuacook/base    latest   0e96a5b5d7d1  About an hour ago   1.769 GB
debian             latest   47af6ca8a14a  8 days ago          125.1 MB
```

Here we have multiple containers, that are no longer running. We have just two images. Note, that each container is an instance of the one of our images.

Let's take advantage of the Docker daemon's ability to manipulate these stopped containers.

First, let's get a container who's behavior we can be certain of.

Attach a shell a `debian` image

```
docker run -it debian /bin/bash
root@b43fb95fdde7:/# exit
```

As soon as you see the bash prompt type exit to shut the container down again.

Run `docker ps -a` once more to get the name of the container you just shut down.

Show all containers

```
$ docker ps -a
CONTAINER ID    IMAGE     COMMAND        ...   PORTS    NAMES
a5119f25fc6a    debian    "/bin/bash"                   romantic_kowalevski
```

Because this container is still in our cache, we can start it up again. Let's pass the interactive flag `-i` to be given a shell to the container once more.

Start a stopped container

```
$ docker start -i romantic_kowalevski
root@b43fb95fdde7:/#
```

Press the up arrow on your keyboard to show that there is in fact `.bash_history` and this *is* the container you just shut down.

Okay, exit once more.

This time, start the container in detached mode. Then, run `docker ps` to see that it is running.

Start a stopped container in detached mode

```
$ docker start romantic_kowalevski
romantic_kowalevski
$ docker ps
CONTAINER ID    IMAGE     COMMAND        ...   PORTS    NAMES
a5119f25fc6a    debian    "/bin/bash"                   romantic_kowalevski
```

But how do we shut it down again.

Stop a container in detached mode

```
$ docker stop romantic_kowalevski
romantic_kowalevski
```

In some cases we may need to take more drastic measure, or we may need to be specific about which signal we send to our container's running process.

Send the `kill` signal to a container

```
$ docker kill --help

Usage:  docker kill [OPTIONS] CONTAINER [CONTAINER...]

Kill a running container

  --help               Print usage
  -s, --signal=KILL    Signal to send to the container
```

Retire a container

Containers are intended to be disposable. We spin them up and tear them down frequently. But we have seen that the remain in cache when we are done using them. Again taking cues from `bash`, the Docker client uses the `rm` argument to tell the daemon to permanently get rid of a container.

This one is **RISKY**. Make sure you understand what you are doing before you run this command.

Remove ALL containers

```
$ docker rm $(docker ps -a -q)
```

First, let's talk about the `-q` flag. We already know that `-a` means show **all** containers, even those that are no longer running. The `-q` flag means "quiet mode" and tells the docker daemon to return only the `CONTAINER_ID` of each container.

Second, a little `bash`. The `$()` syntax in `bash` is the way to launch a process in a subshell. For example, `$(echo 'foo')` is equivalent to writing `foo`. Here, we are running `docker ps -a -q` inside that subprocess. As the subprocess will return a list of `CONTAINER_ID`s associated with all containers, this is equivalent to passing these ids to the final command.

Finally, we run `docker rm` which will remove any container that is passed to it as argument. We passed a list of containers via the `bash` subprocess.

The final effect: **THIS WILL DELETE ALL CONTAINERS ON YOUR SYSTEM** .

I find this command very useful for cleaning up after a long session of Docker development, but ...

- it is nuclear
- there is no turning back
- do so at your own risk

A Clean System

```
$ docker ps -a
CONTAINER ID        IMAGE       COMMAND       CREATED       STATUS       PORTS       NAMES
$ docker images
REPOSITORY          TAG      IMAGE ID       CREATED             SIZE
joshuacook/base     latest   0e96a5b5d7d1   About an hour ago   1.636 GB
debian              latest   47af6ca8a14a   8 days ago          125.1 MB
```

7

Building a `minimal-notebook`

At this point, we have a basic process for building a Docker image.

1) We write an instruction into our `Dockerfile`.
2) We run `docker build`
3) If the build is successful, we push our changes to Github.
4) These changes will be picked up by Docker Hub and an automatic build will be run.

We also have a `base` image.

Getting Started

We have a method: `INSTRUCTION`, `build`, `push`.

We have a clean system.

Let's get down to the business of building our Jupyter container. For the next step, rather than adding to the `Dockerfile` we have already written, we will leverage the `FROM` instruction to start from where we left off with a fresh `Dockerfile`.

Configure `git`

```
$ mkdir minimal-notebook && cd minimal-notebook
$ git init
$ touch Dockerfile
$ git add Dockerfile
$ git commit -m 'init'
```

Configure Github

On github, create a repo called `minimal-notebook`.

```
$ git remote add origin git@github.com:#USERNAME#/minimal-notebook.git
$ git push -u origin master
```

Create an Automated Build on Docker Hub

Select 'Create Automated Build'. Given an option between Github and BitBucket, choose Github. Choose the previously created repo, `minimal-notebook`.

Build the image on the `base` image we just made We will start where we left off and add ourselves as maintainer.

```
FROM joshuacook/base

MAINTAINER Joshua Cook <me@joshuacook.me>
```

Build

```
$ docker build -t joshuacook/minimal-notebook .
...
Step 1 : FROM joshuacook/base
 ...
Successfully built 0373a95a1335
```

Push

```
$ git add Dockerfile
$ git commit -m 'add FROM and MAINTAINER statements'
$ git push
```

This should be old hat for us by now. Of note, the `IMAGE ID` matches the `IMAGE ID` for our `base` image. After the push, the automated build should start on Docker Hub.

Configure Environment The environment for our Jupyter installation requires some configuration. Here we will set a few environment variables and add the user, `jovyan`, under which we will provision our installation and ultimately run our machine.

```
# Configure environment
ENV CONDA_DIR /opt/conda
ENV PATH $CONDA_DIR/bin:$PATH
ENV SHELL /bin/bash
ENV NB_USER jovyan
ENV NB_UID 1000

# Create jovyan user with UID=1000 and in the 'users' group
RUN useradd -m -s /bin/bash -N -u $NB_UID $NB_USER &&
    mkdir -p /opt/conda &&
    chown jovyan /opt/conda
```

Build

```
$ docker build -t joshuacook/minimal-notebook .
...
Step 3 : ENV CONDA_DIR /opt/conda
...
Step 8 : RUN useradd ...
...
Successfully built b61d465e26ee
```

Push

```
$ git add Dockerfile
$ git commit -m 'configure environment and add jovyan user'
$ git push
```

Install Jupyter

```
# Install Jupyter notebook
RUN conda install --quiet --yes
    'notebook=4.1*'
    terminado
    && conda clean -tipsy
```

Build

```
$ docker build -t joshuacook/minimal-notebook .
...
Step 9 : RUN conda install --quiet --yes     'notebook=4.1*'     terminado     &&
    conda clean -tipsy
...
Successfully built eb91faf9a5d5
```

Push

```
$ git add Dockerfile
$ git commit -m 'install Jupyter notebook'
$ git push
```

Operate as `jovyan`

To this point, we have not paid any attention to the issue of which user we are using to build our image. By default, Docker uses `root` to do its installation. Having just created the `jovyan` user that will be the primary user for our jupyter system, we will now for the first time, use a different user. This is by design. We now provision our image with the core libraries required by Jupyter. These libraries must be owned by the main Jupyter user. This is done with the instruction `USER`.

```
USER jovyan
```

Build

```
$ docker build -t joshuacook/minimal-notebook .
...
Step 10 : USER jovyan
...
Successfully built 539930ba07da
```

Push

```
$ git add Dockerfile
$ git commit -m 'switch to user jovyan'
$ git push
```

Setup the Jovyan home directory The source and configuration files we will be using to run our Jupyter installation will all be stored in the home directory of the `jovyan` user. Let us take the time to properly set up this directory.

```
# Setup Jovyan home directory
RUN mkdir /home/$NB_USER/work &&
    mkdir /home/$NB_USER/.jupyter &&
    mkdir /home/$NB_USER/.local &&
    echo "cacert=/etc/ssl/certs/ca-certificates.crt" > /home/$NB_USER/.curlrc
```

Take note of `$NB_USER`. Recall that we defined this environment variable earlier in the Dockerfile. The `work` directory is where we will store all of our Jupyter files. In chapter 4, when we discussed data persistence, we attached a local volume to this directory in order to save files from the local system to the Juptyer instance. `.jupyter` will be used to store our Jupyter configuration such as additional kernels we might wish to install. `.local` will store user specific application configurations.

Build

```
$ docker build -t joshuacook/minimal-notebook .
...
Step 11 : RUN mkdir /home/$NB_USER/work ...
...
```

Push

```
$ git add Dockerfile
$ git commit -m 'setup jovyan home directory'
$ git push
```

Configure container startup

This set of instructions requires `root`-level permsissions so we switch back to `USER root`. We hardcode the port over which we will expose the Jupyter platform using the `EXPOSE` instruction. We set a `WORKDIR` to use for all subsequent commands. Finally, we add an argument using the `CMD` instruction to be passed to our `ENTRYPOINT` (which was defined in our `base` image). Recall that we defined our `ENTRYPOINT` as `["/tini", "--"]`. With this instruction, `tini` will start and then run `start-notebook.sh` as a subprocess.

```
USER root

# Configure container startup as root
EXPOSE 8888
WORKDIR /home/$NB_USER/work
ENTRYPOINT ["tini", "--"]
CMD ["start-notebook.sh"]
```

Build

```
$ docker build -t joshuacook/minimal-notebook .
...
Step 12 : USER root
...
Step 16 : CMD start-notebook.sh
...
```

Push

```
$ git add Dockerfile
$ git commit -m 'startup configuration'
$ git push
```

Add local files

You may have noticed that we are passing a shell script as a process to be run by our container after `tini` as started. We will need to define this script and make sure that it is included with our image. The Jupyter Project has written a special script to handle the startup of our Jupyter system for several edge cases. We will use this script as provided. Additionally, our Jupyter installation will require a configuration file, `jupyter_notebook_config.py`, to run. We will also use this as provided.

These files should be added to the `minimal-notebook` directory and included with the Github repo for the automated build.

start-notebook.sh

```
#!/bin/bash

# Handle special flags if we're root
if [ $UID == 0 ] ; then
    # Change UID of NB_USER to NB_UID if it does not match
    if [ "$NB_UID" != $(id -u $NB_USER) ] ; then
        usermod -u $NB_UID $NB_USER
        chown -R $NB_UID $CONDA_DIR
    fi

    # Enable sudo if requested
    if [ ! -z "$GRANT_SUDO" ]; then
        echo "$NB_USER ALL=(ALL) NOPASSWD:ALL" > /etc/sudoers.d/notebook
    fi

    # Start the notebook server
    exec su $NB_USER -c "env PATH=$PATH jupyter notebook $*"
else
    # Otherwise just exec the notebook
    exec jupyter notebook $*
fi
```

jupyter_notebook_config.py

```
# Copyright (c) Jupyter Development Team.
from jupyter_core.paths import jupyter_data_dir
import subprocess
import os
import errno
import stat

PEM_FILE = os.path.join(jupyter_data_dir(), 'notebook.pem')

c = get_config()
c.NotebookApp.ip = '*'
c.NotebookApp.port = 8888
c.NotebookApp.open_browser = False

# Set a certificate if USE_HTTPS is set to any value
if 'USE_HTTPS' in os.environ:
    if not os.path.isfile(PEM_FILE):
        # Ensure PEM_FILE directory exists
        dir_name = os.path.dirname(PEM_FILE)
        try:
            os.makedirs(dir_name)
        except OSError as exc: # Python >2.5
            if exc.errno == errno.EEXIST and os.path.isdir(dir_name):
                pass
            else: raise
        # Generate a certificate if one doesn't exist on disk
        subprocess.check_call(['openssl', 'req', '-new',
            '-newkey', 'rsa:2048', '-days', '365', '-nodes', '-x509',
            '-subj', '/C=XX/ST=XX/L=XX/O=generated/CN=generated',
            '-keyout', PEM_FILE, '-out', PEM_FILE])
        # Restrict access to PEM_FILE
        os.chmod(PEM_FILE, stat.S_IRUSR
```

Add local files and set permissions

```
# Add local files as late as possible to avoid cache busting
# Start notebook server
COPY start-notebook.sh /usr/local/bin/
COPY jupyter_notebook_config.py /home/$NB_USER/.jupyter/
RUN chown -R $NB_USER:users /home/$NB_USER/.jupyter
RUN chmod +x /start-notebook.sh
```

Here we use the `COPY` instruction to copy these files from our local directory to the image and use to `RUN` instructions to set permissions for our files.

Build

```
docker build -t joshuacook/minimal-notebook .
...
Step 17 : COPY start-notebook.sh /usr/local/bin/
...
Step 20 : RUN chmod +x /start-notebook.sh
...
Removing intermediate container 81fee7bb70bf
Successfully built ecd1a2503008
```

Push

```
$ git add Dockerfile start-notebook.sh jupyter_notebook_config.py
$ git commit -m 'add startup script and config file and set permissions'
$ git push
```

Switch to jovyan user for final running of platform

```
# Switch back to jovyan to avoid accidental container runs as root
USER jovyan
```

Build

```
docker build -t joshuacook/minimal-notebook .
...
Step 21 : USER jovyan
...
Successfully built 660c0697f1e1
```

Push

```
$ git add Dockerfile start-notebook.sh jupyter_notebook_config.py
$ git commit -m 'switch to jovyan'
$ git push
```

Final `minimal-notebook` Dockerfile

Dockerfile

```
FROM joshuacook/base

MAINTAINER Joshua Cook <me@joshuacook.me>

# Configure environment
ENV CONDA_DIR /opt/conda
ENV PATH $CONDA_DIR/bin:$PATH
ENV SHELL /bin/bash
ENV NB_USER jovyan
ENV NB_UID 1000

# Create jovyan user with UID=1000 and in the 'users' group
RUN useradd -m -s /bin/bash -N -u $NB_UID $NB_USER &&
    mkdir -p /opt/conda &&
    chown jovyan /opt/conda

# Install Jupyter notebook
RUN conda install --quiet --yes
    'notebook=4.1*'
    terminado
    && conda clean -tipsy

USER jovyan

# Setup Jovyan home directory
RUN mkdir /home/$NB_USER/work &&
    mkdir /home/$NB_USER/.jupyter &&
    mkdir /home/$NB_USER/.local &&
    echo "cacert=/etc/ssl/certs/ca-certificates.crt" > /home/$NB_USER/.curlrc

USER root

# Configure container startup as root
EXPOSE 8888
WORKDIR /home/$NB_USER/work
ENTRYPOINT ["/tini", "--"]
CMD ["/start-notebook.sh"]

# Add local files as late as possible to avoid cache busting
# Start notebook server
COPY start-notebook.sh /start-notebook.sh
COPY jupyter_notebook_config.py /home/$NB_USER/.jupyter/
RUN chown -R $NB_USER:users /home/$NB_USER/.jupyter
RUN chmod +x /start-notebook.sh

# Switch back to jovyan to avoid accidental container runs as root
USER jovyan
```

Run the `minimal-notebook` in detached mode on port 5000

```
$ docker run -d -p 5000:8888 joshuacook/minimal-notebook
4b6882515f1bcf233101ca942f366e51e71f537922adb049ea488720e7310a52
```

8

Building a `datascience` image

Being experts at this point, let's sprint through this one wherein we provision our machine with the core computational science libraries and add Python 2 and R kernels.

Continue Build from `minimal-notebook`

```
FROM joshuacook/minimal-notebook

MAINTAINER Joshua Cook <me@joshuacook.me>
```

Build and Push

```
$ docker build -t joshuacook/datascience .
$ git add Dockerfile
$ git commit -m 'continue build from minimal-notebook'
$ git push
```

Provision image with `pip`, `pip3`, `gcc`, and `gfortran`

```
USER root

# Provision Image
RUN apt-get update && \
    apt-get install -y --no-install-recommends \
    python2.7 \
    python-dev \
    python-pip \
    python3 \
    python3-dev \
    python3-pip \
    fonts-dejavu \
    gfortran \
    gcc && apt-get clean
```

Build and Push

```
$ docker build -t joshuacook/datascience .
$ git add Dockerfile
$ git commit -m 'pip, pip3, gcc, gfortran'
$ git push
```

Provision for `matplotlib anim`

```
# libav-tools for matplotlib anim
RUN apt-get update && \
    apt-get install -y --no-install-recommends libav-tools && \
    apt-get clean
```

Build and Push

```
$ docker build -t joshuacook/datascience .
$ git add Dockerfile
$ git commit -m 'libav-tools'
$ git push
```

Provision scipy and numpy for Python 3

```
# Provision scipy/numpy
RUN conda install --yes \
    'ipywidgets=4.1*' \
    'pandas=0.17*' \
    'matplotlib=1.5*' \
    'scipy=0.17*' \
    'seaborn=0.7*' \
    'scikit-learn=0.17*' \
    'scikit-image=0.11*' \
    'sympy=0.7*' \
    'cython=0.23*' \
    'patsy=0.4*' \
    'statsmodels=0.6*' \
    'cloudpickle=0.1*' \
    'dill=0.2*' \
    'numba=0.23*' \
    'bokeh=0.11*' \
    'h5py=2.5*' \
    && conda clean -tipsy
```

Build and Push

```
$ docker build -t joshuacook/datascience .
$ git add Dockerfile
$ git commit -m 'scipy/numpy in python 3'
$ git push
```

Provision scipy and numpy for Python 2

```
# Install Python 2 packages
RUN conda create --yes -p $CONDA_DIR/envs/python2 python=2.7 \
    'ipython=4.1*' \
    'ipywidgets=4.1*' \
    'pandas=0.17*' \
    'matplotlib=1.5*' \
    'scipy=0.17*' \
    'seaborn=0.7*' \
    'scikit-learn=0.17*' \
    'scikit-image=0.11*' \
    'sympy=0.7*' \
    'cython=0.23*' \
    'patsy=0.4*' \
    'statsmodels=0.6*' \
    'cloudpickle=0.1*' \
    'dill=0.2*' \
    'numba=0.23*' \
    'bokeh=0.11*' \
    'h5py=2.5*' \
    'pyzmq' \
    && conda clean -tipsy
```

Build and Push

```
$ docker build -t joshuacook/datascience .
$ git add Dockerfile
$ git commit -m 'scipy/numpy in python 2'
$ git push
```

Provision `R`

```
# R packages including IRKernel which gets installed globally.
RUN conda config --add channels r
RUN conda install --yes \
    'rpy2=2.7*' \
    'r-base=3.2*' \
    'r-irkernel=0.5*' \
    'r-plyr=1.8*' \
    'r-devtools=1.9*' \
    'r-dplyr=0.4*' \
    'r-ggplot2=1.0*' \
    'r-tidyr=0.3*' \
    'r-shiny=0.12*' \
    'r-rmarkdown=0.8*' \
    'r-forecast=5.8*' \
    'r-stringr=0.6*' \
    'r-rsqlite=1.0*' \
    'r-reshape2=1.4*' \
    'r-nycflights13=0.1*' \
    'r-caret=6.0*' \
    'r-rcurl=1.95*' \
    'r-randomforest=4.6*' && conda clean -tipsy
```

Build and Push

```
$ docker build -t joshuacook/datascience .
$ git add Dockerfile
$ git commit -m 'provision R'
$ git push
```

Provision libraries for interfacing with PostgresQL and Redis

```
# Install database handling libraries
RUN pip install sqlalchemy redis
RUN pip3 install sqlalchemy redis
```

Build and Push

```
$ docker build -t joshuacook/datascience .
$ git add Dockerfile
$ git commit -m 'PostgresQL and Redis libs'
$ git push
```

Configure Python 2 kernel

```
# Install Python 2 kernel spec globally to avoid permission problems when NB_UID
# switching at runtime.
RUN $CONDA_DIR/envs/python2/bin/python \
    $CONDA_DIR/envs/python2/bin/ipython \
    kernelspec install-self
```

Build and Push

```
$ docker build -t joshuacook/datascience .
$ git add Dockerfile
$ git commit -m ''
$ git push
```

Prepare for running notebook

```
# Switch back to jovyan to avoid accidental container runs as root
USER jovyan
```

Build and Push

```
$ docker build -t joshuacook/datascience .
$ git add Dockerfile
$ git commit -m ''
$ git push
```

Final `datascience` image

Dockerfile

```
FROM joshuacook/minimal-notebook

MAINTAINER Joshua Cook <me@joshuacook.me>

USER root

# Provision Image
RUN apt-get update && \
    apt-get install -y --no-install-recommends \
    python2.7 \
    python-dev \
    python-pip \
    python3 \
    python3-dev \
    python3-pip \
    fonts-dejavu \
    gfortran \
    gcc && apt-get clean

# libav-tools for matplotlib anim
RUN apt-get update && \
    apt-get install -y --no-install-recommends libav-tools && \
    apt-get clean

# Provision scipy/numpy
RUN conda install --yes \
    'ipywidgets=4.1*' \
    'pandas=0.17*' \
    'matplotlib=1.5*' \
    'scipy=0.17*' \
    'seaborn=0.7*' \
    'scikit-learn=0.17*' \
    'scikit-image=0.11*' \
    'sympy=0.7*' \
    'cython=0.23*' \
    'patsy=0.4*' \
    'statsmodels=0.6*' \
    'cloudpickle=0.1*' \
    'dill=0.2*' \
    'numba=0.23*' \
    'bokeh=0.11*' \
    'h5py=2.5*' \
    && conda clean -tipsy
```

Dockerfile (continued)

```
# Install Python 2 packages
RUN conda create --yes -p $CONDA_DIR/envs/python2 python=2.7 \
    'ipython=4.1*' \
    'ipywidgets=4.1*' \
    'pandas=0.17*' \
    'matplotlib=1.5*' \
    'scipy=0.17*' \
    'seaborn=0.7*' \
    'scikit-learn=0.17*' \
    'scikit-image=0.11*' \
    'sympy=0.7*' \
    'cython=0.23*' \
    'patsy=0.4*' \
    'statsmodels=0.6*' \
    'cloudpickle=0.1*' \
    'dill=0.2*' \
    'numba=0.23*' \
    'bokeh=0.11*' \
    'h5py=2.5*' \
    'pyzmq' \
    && conda clean -tipsy

# R packages including IRKernel which gets installed globally.
RUN conda config --add channels r
RUN conda install --yes \
    'rpy2=2.7*' \
    'r-base=3.2*' \
    'r-irkernel=0.5*' \
    'r-plyr=1.8*' \
    'r-devtools=1.9*' \
    'r-dplyr=0.4*' \
    'r-ggplot2=1.0*' \
    'r-tidyr=0.3*' \
    'r-shiny=0.12*' \
    'r-rmarkdown=0.8*' \
    'r-forecast=5.8*' \
    'r-stringr=0.6*' \
    'r-rsqlite=1.0*' \
    'r-reshape2=1.4*' \
    'r-nycflights13=0.1*' \
    'r-caret=6.0*' \
    'r-rcurl=1.95*' \
    'r-randomforest=4.6*' && conda clean -tipsy
```

Dockerfile (continued)

```
# Install database handling libraries
RUN pip install sqlalchemy redis
RUN pip3 install sqlalchemy redis

# Install Python 2 kernel spec globally to avoid permission problems when NB_UID
# switching at runtime.
RUN $CONDA_DIR/envs/python2/bin/python \
    $CONDA_DIR/envs/python2/bin/ipython \
    kernelspec install-self


# Switch back to jovyan to avoid accidental container runs as root
USER jovyan
```

9

Docker Compose and Docker Cloud

Link to a Cloud Provider

Having built our image, we are now ready to deploy our image via Docker Cloud. After setting up a Docker Cloud account and an Amazon Web Services account, we will begin by linking the two. This can be done via Account Info under the tab Cloud Providers. Simply click the link 'Add Credentials' and complete the linking process.

You will need you AWS access_key and secret_access_key available under IAM >> Users. Select your username then the tab Security Credentials. Your access_key will be displayed. Choose Create Access Key to generate a secret_access_key and take note of these two values.

Deploy a Node

Visit the Node tab and click 'Launch new node cluster'. Define the essential information for the node. Since we have built our stack using Docker, we can always change this

information later if we need more or less cpu power or disk space. For now, a basic 2 GB[2 CPU/2 GB RAM] should be sufficient.

Stackfile

To deploy our computational stack we will not be working with individual services. Rather we will use Docker's Stackfile format to define our stack. This is much more straightforward then the Dockerfile can be entered directly into the browser. Visit the Stacks tab and click 'Create Stack'. Give the stack a name such as datascience and enter the following:

Stackfile

```
all-spark:
  image: 'joshuacook/datascience'
  environment:
    - 'PASSWORD=cats with glasses'
  links:
    - postgres
    - redis
  ports:
    - '80:8888'
  tags:
    - spark
postgres:
  image: 'postgres:latest'
  volumes:
    - #NEED TO HANDLE PERSISTENCE#
  ports:
    - "5432"
redis:
  image: 'redis:latest'
  environment:
    - REDIS_APPENDFSYNC=always
    - REDIS_APPENDONLY=yes
    - REDIS_PASS=1lg5udHlMYyK
  ports:
    - "6379"
  restart: always

  volumes:
    - '/usr/local/data:/var/lib/redis/data'
```

Data Persistence

10

Security

11

Advanced Techniques

Brief Primer on Signals

A process is an instance of a running computer program. A *child* process is a process that is initiated by another *parent* process. We designate these process using unique process identifiers or `PID`s. On every linux system, there are two special processes, `PID 0` and `PID 1`. `PID 0` is beyond the scope of this text, is actually a part of the kernel, and is not a normal user-mode process. `PID 1` is the `init` process responsible for starting and stopping the system and in the operating systems we will be using is actively reserved and treated specially.

The way that processes communicate with each other is via **signals** . When a signal is issued to a process, the process is interrupted and a signal handler is executed. Some common signals include `SIGINT`, `SIGKILL`, and `SIGTERM` – interrupt, kill, and terminate, respectively. Processes tell the kernel how they will handle a passed signal.

Redefining signal in Python

```
In [1]: import os, signal, subprocess, sys

In [2]: def handler(signum, frame):
   ...:     print('Shutting down...')
   ...:     print("Yeah, but I don't think I want to quit. You can't make me.")
   ...:

In [3]: pid = os.getpid()

In [4]: pid
Out[4]: 37282

In [5]: signal.signal(signal.SIGTERM, handler)
Out[5]: <Handlers.SIG_DFL: 0>

In [6]: os.kill(pid, signal.SIGTERM)
Shutting down...
Yeah, but I don't think I want to quit. You can't make me.

In [7]: os.kill(pid, signal.SIGKILL)
Killed: 9
```

Here, at `In [2]`, we defined a functioned called `handler`. At `In [5]`, we associated the signal `SIGTERM` with the function `handler`. When we send `SIGTERM` to our process using the `os.kill` command, the function `handler` is executed. Because we did not define an actual method for terminating the process, it did not terminate. We then killed the process using the existing defintion of `SIGKILL`.

Now, we describe a simple webserver that has `SIGTERM` and `SIGINT` defined.

server.py

```
#!/usr/bin/env python
from http.server import BaseHTTPRequestHandler, HTTPServer
import signal, sys

class MyHTTPRequestHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.wfile.write(bytes("Hello world!", "utf8"))

if __name__ == '__main__':
    def MySigtermHandler(signum, frame):
        print("Shutting down server via SIGTERM")
        httpd.socket.close()
        sys.exit(0)

    def MySigintHandler(signum, frame):
        print("Shutting down server via SIGINT")
        httpd.socket.close()
        sys.exit(0)

    signal.signal(signal.SIGTERM, MySigtermHandler)
    signal.signal(signal.SIGINT, MySigintHandler)

    print("starting server...")
    httpd = HTTPServer(('127.0.0.1', 8000), MyHTTPRequestHandler)
    print('running server...')
    httpd.serve_forever()
```

Let's see what all this signal business is about.

Run the simple server

```
$ python3 server.py
starting server...
running server...
```

You can visit `localhost:8000` to confirm that it works.

Send the `SIGINT` signal Press `ctrl-C` on your keyboard.

```
^CShutting down server via SIGINT
```

Perfect. Our special `SIGINT` handler executed. Let's see if we can get the `SIGTERM`

handler to execute.

Run the simple server

```
$ python3 server.py
starting server...
running server...
```

Sending a `SIGTERM` is a bit more challenging. First, let's suspend the server process by sending the `TSTP` signal. Press `ctrl-Z` on your keyboard.

Backround the server process

```
^Z
[1]+  Stopped                 python3 server.py
```

Identify the `PID` of our server

```
$ ps
```

The `bash` command `ps` works similarly to the `docker ps` command, though in this case it displays a list of running processes. Rather than getting a list of all processes, we pipe ( ) the output into a `grep` command which searches the output for the text `server.py` and only displays that line.

We can ignore the second line. This is the command we just ran, a process which definitely contains the string `server.py`! What we are interested in is the first line, which shows us the `PID` of the server we just ran. To test our redefined `SIGTERM`, we want to send `SIGTERM` to this job. We do this by the `kill` command.

Send the `SIGTERM` signal

```
$ kill -SIGTERM 37956
```

Wait, nothing happened! This is because our process is still suspended. Type `fg` to foreground the process we just suspended.

```
$ fg
python3 server.py
Shutting down server via SIGTERM
```

Here the bash shell `echo`ed the process that was unsuspended and then immediately sent `SIGTERM` to the process. This triggered the `MySigtermHandler` and the server shutdown. Perfect.

Signals & Docker

The docker client commands `docker stop` and `docker kill` are the primary way that we send signals to the main process in a Docker container. As mentioned in Chapter 5, Docker best practices hold that each container should run a single process. This is a significant change from your traditional web application running on a server. The significance of this is that our process of interest is `PID 1`, which as previously mentioned, receives special treatment in our system. This is to say, when we send `docker kill`, the signal is handled by `PID 1`. The issue is exactly **HOW** the signal is handled.

An example is in order.

Case 1: The Application is running as a foreground process (`PID1`).

In this case, the application should handle signals directly. Let's take a look. First, let's use a `Dockerfile` to run our simple server.

Simple Server with `CMD` in JSON format

```
FROM python:3.5

COPY ./server.py ./server.py

EXPOSE 8000

CMD [ "python3", "./server.py"]
```

First, build and run the Docker image.

Build & Run

```
$ docker build -t joshuacook/sigterm-case-1 .
Sending build context to Docker daemon 57.86 kB
...
Successfully built ee8e3aa0f4ea
$ docker run -it --rm -p 8000:8000 --me="sigterm-case-1" joshuacook/sigterm-case-1
starting server...
running server...
```

Now, open a new terminal to your `DOCKER_HOST` and issue the `kill` command.

Kill the simple server

```
$ docker kill --signal="SIGTERM" sigterm-case-1
sigterm-case-1
```

Returning to the original terminal should show us that the `MySigtermHandler` was executed.

```
Shutting down server via SIGTERM
```

We have seen that using `docker kill --signal="SIGTERM"` used `MySigtermHandler`.

Case 2: The Application is running as a background process (a child of `PID1`).

In this case, we the difference is *very* subtle. Notice in this `Dockerfile` we are passing the arguments to the `CMD` instruction directly, **NOT in JSON** format.

Simple Server with `CMD` not in JSON format

```
FROM python:3.5

COPY ./server.py ./server.py

RUN chmod +x ./server.py

EXPOSE 8000

CMD python3 server.py
```

Let's see what happens.

First, build and run the Docker image.

Build & Run

```
$ docker build -t joshuacook/sigterm-case-2 .
Sending build context to Docker daemon 57.86 kB
...
Successfully built cf979e0e2de0
$ docker run -it --rm -p 8000:8000 --me="sigterm-case-2" joshuacook/sigterm-case-2
starting server...
running server...
```

Now send the `SIGTERM`.

Kill the simple server

```
$ docker kill --signal="SIGTERM" sigterm-case-2
sigterm-case-2
```

Returning to the previous window, we see that our server is still running.

This is not good.

The `CMD` Instruction

The `CMD` instruction has three forms: *exec* form, *parameters* form, and *shell* form. Here we must consider the identity of `PID 1` in each form.

the exec form `CMD ["executable", "param1", "param2"]`

In case 1, we use the exec form, `CMD [ "python3", "server.py"]`. Here `PID 1` is `python3`. When `docker kill` sends `SIGTERM` to `PID1` is reaches our running process and executes correctly.

the parameters form `CMD ["param1", "param2"]`

ACTIVE: WE DID NOT USE THIS THOUGH WE MAY WITH ENTRYPOINT

the shell form `CMD executable param1 param2`

In case 2, we use the shell form, `CMD python3 server.py`. Here `PID 1` is the default, `/bin/sh -c`. It then spawns a child process to run our server. When `docker kill` sends `SIGTERM` to `PID1` it does not reach our running process. Furthermore, `/bin/sh` ignores any signals while a child process is running. Thus, our server stays alive despite our having sent the `SIGTERM` signal.

It gets worse.

PICTURE

The Silent Failure

Take a look at the second terminal to the `DOCKER_HOST` we opened. In particular, compare the output of the two `docker kill` commands we issued.

```
$ docker kill --signal="SIGTERM" sigterm-case-1
sigterm-case-1
$ docker kill --signal="SIGTERM" sigterm-case-2
sigterm-case-2
```

They are identical. They returned the name of the container to which we sent the signal. There is **ZERO** indication that the first signal successfully terminated the container, but the second failed. In the second case, we would have what is referred to as a zombie process or a zombie container. We are going to need to deal with these zombies.

The Zombie Killer

Using Vagrant to configure your Docker Virtual machine

finer control

on mac

vagrant ssh into machine

don't need to rely on Docker quickstart

Closer to what you will run in production

```
VAGRANTFILE_API_VERSION = "2"
$bootstrap=<<SCRIPT
apt-get update
apt-get -y install wget
wget -qO- https://get.docker.com/ | sh
gpasswd -a vagrant docker
service docker restart
SCRIPT

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
      config.vm.box = "ubuntu/trusty64"
      config.vm.network "private_network", ip: "192.168.33.10"
config.vm.provider "virtualbox" do |vb|
vb.customize ["modifyvm", :id, "--memory", "1024"]
end
config.vm.provision :shell, inline: $bootstrap end
```

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

# Specify Vagrant version and Vagrant API version
Vagrant.require_version ">= 1.6.0"
VAGRANTFILE_API_VERSION = "2"

# Create and configure the VM(s)
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|

  # Assign a friendly name to this host VM
  config.vm.hostname = "docker-host"

  # Skip checking for an updated Vagrant box
  config.vm.box_check_update = false

  # Always use Vagrant's default insecure key
  config.ssh.insert_key = false

  # Spin up a "host box" for use with the Docker provider
  # and then provision it with Docker
  config.vm.box = "slowe/ubuntu-trusty-x64"
  config.vm.provision "docker"

  # Disable synced folders (prevents an NFS error on "vagrant up")
  config.vm.synced_folder ".", "/vagrant", disabled: true
end
```

12

Command Reference

`docker`

paragraph{} Build an image from a Dockerfile. The `-t` flag tells the daemon to name this image `#NAME#` and optionally tag it `#TAG#`.

paragraph{} List images.

paragraph{} List images showing only numeric id.

paragraph{} Display system-wide information.

paragraph{} View logs for container in detached mode.

paragraph{} List running containers.

paragraph{} List all containers.

paragraph{} List all containers showing only numeric ids.

paragraph{} Delete a container.

paragraph{} Delete all containers.

paragraph{} Delete an image.

paragraph{} Delete all images.

paragraph{} 1. Search for an image locally 1. If not found, search for an image with default tag on Docker hub and pull that image 1. Run an instance of that image as a container on `DOCKER_HOST` 1. Shut down again.

'! 1. Search for an image locally 1. If not found, search for an image with default tag on Docker hub and pull that image 1. Run an instance of that image as a container on `DOCKER_HOST` 1. Execute `/bin/echo 'Hello World!'` 1. Shut down again.

paragraph{} 1. Search for an image locally 1. If not found, search for an image with default tag on Docker hub and pull that image 1. Run an instance of that image as a container on `DOCKER_HOST` 1. Run an interactive shell to that container 1. Upon user exit from the shell, shut down again.

paragraph{} 1. Search for an image locally 1. If not found, search for an image with default tag on Docker hub and pull that image 1. Run an instance of that image as a container on `DOCKER_HOST` 1. Expose `#CONTAINER_PORT#` to `#HOST_PORT#` on `DOCKER_HOST` 1. When the underlying process exits, shut the container down once more.

paragraph{} 1. Search for an image locally 1. If not found, search for an image with default tag on Docker hub and pull that image 1. Run an instance of that image as a container in **detached** mode on `DOCKER_HOST` 1. Expose `#CONTAINER_PORT#` to `#HOST_PORT#` on `DOCKER_HOST`

paragraph{} Stop a running container.

paragraph{} Display the running processes of a container.

docker-machine

paragraph{} Restart a machine.

paragraph{} Refresh your environment.

`Dockerfile` domain specific language

paragraph{} Defines a variable that uses can pass at build-time to the daemon. Note that the `DEFAULT_VALUE` is optional, but provides a useful way to pass non-persistent environment variables to a build script.

paragraph{} Sets the base image to be used by all subsequent instructions.

paragraph{} Sets the Author field of the generated images.

Will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the `Dockerfile`.