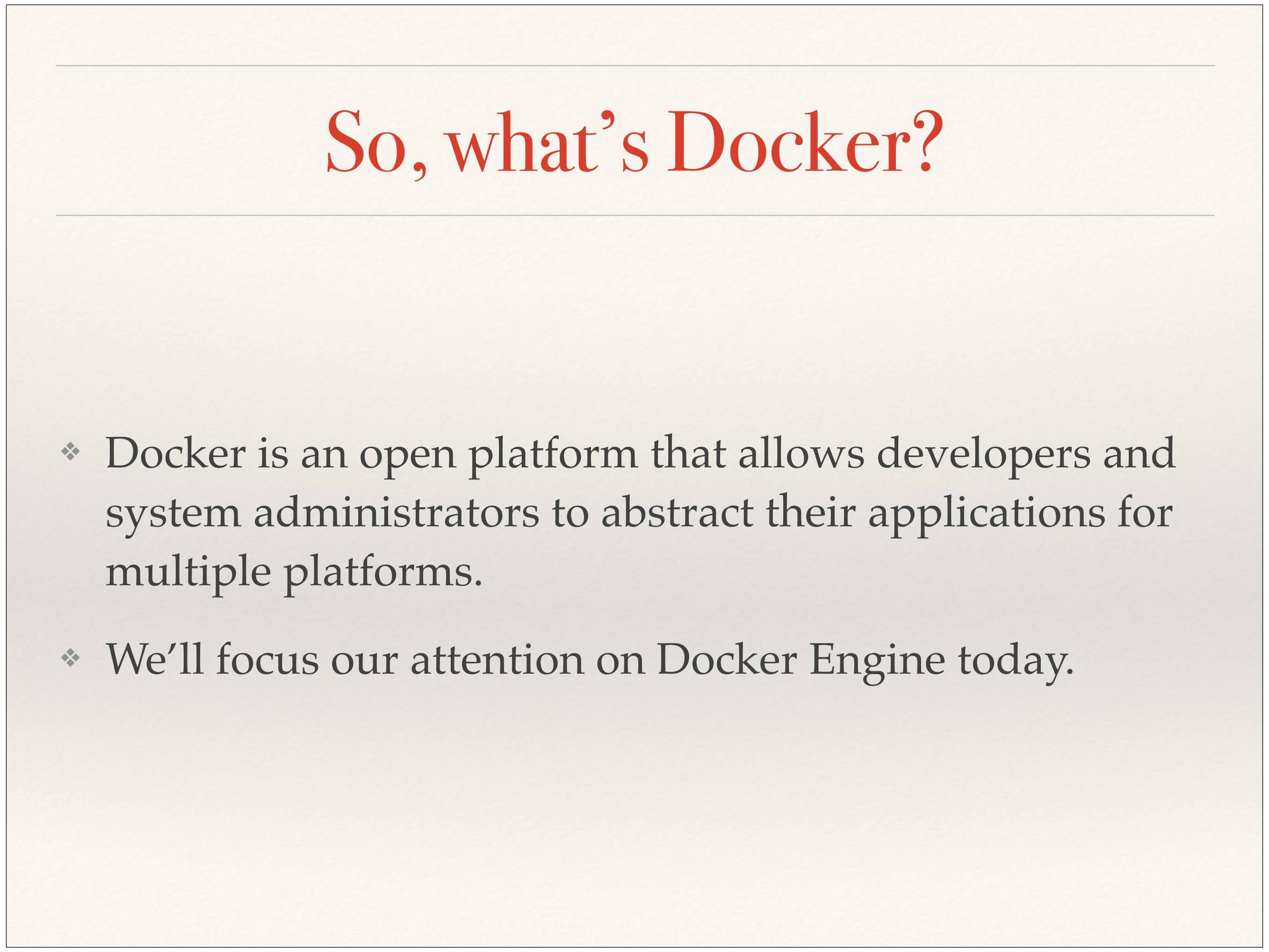
Docker for Fun and Profit

Kel Cecil
@praisechaos



So, what's Docker?

- Docker is an open platform that allows developers and system administrators to abstract their applications for multiple platforms.
- We'll focus our attention on Docker Engine today.

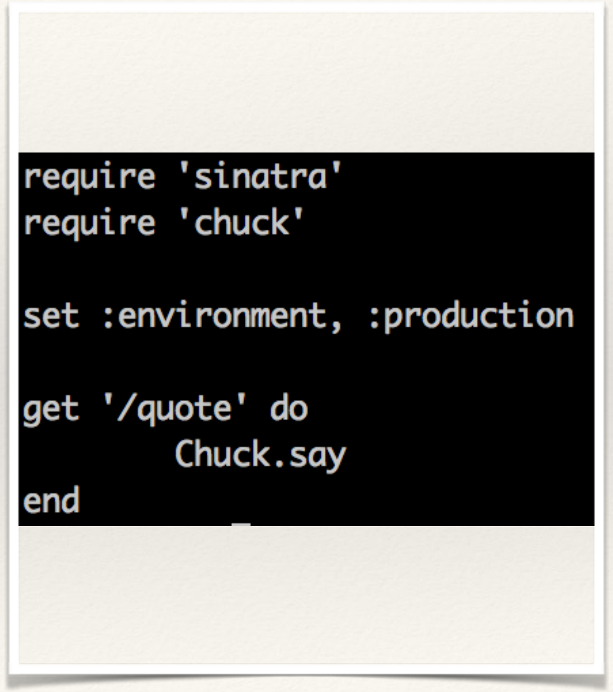
Docker defines itself as open platform that allows developers and system administrators to abstract their applications for multiple platforms.

Easy to have self-contained applications with dependencies from any sources.
Apps with dependencies installed from debians repository can be ran on Fedora boxes with ease.

Docker's platform is growing over time with new additions like Swarm
We'll concentrate today on what most people think of when they think of Docker. Docker Engine.

Disrupt Chuck Norris Quotes!

- We're going to containerize and run our simple app.
- Find the app and Dockerfile at: https://github.com/kelcecil/docker-techfest15

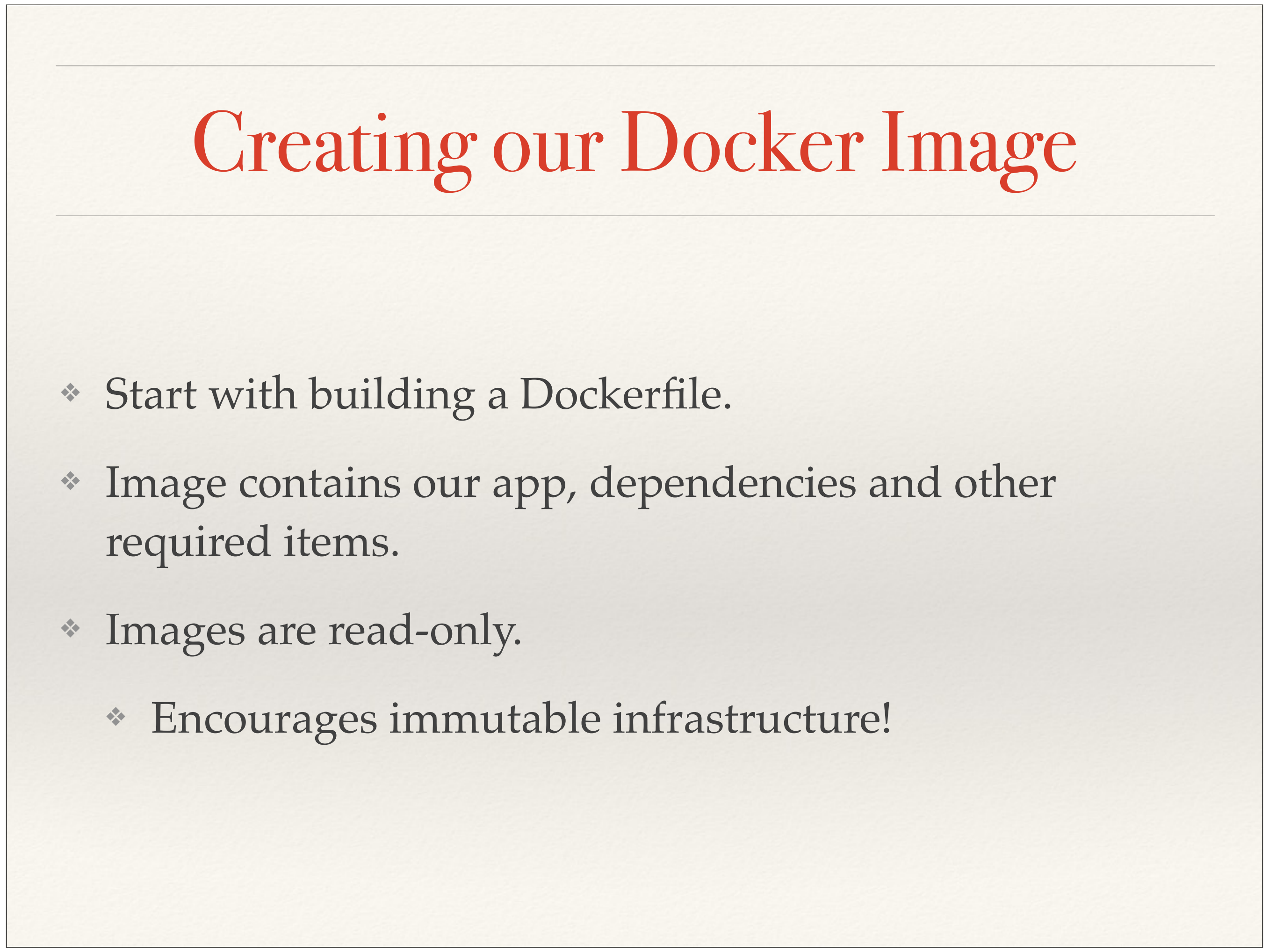


```
require 'sinatra'
require 'chuck'

set :environment, :production

get '/quote' do
        Chuck.say
end
```

One of the best ways to understand containerization is to containerize an app.

We'll be containerizing a simple Sinatra web app with a single GET endpoint that just returns a Chuck Norris quote.

Don't worry about taking notes. You can find these slides and example files in this repo on Github. I'll tweet out the link after the presentation, so just remember my twitter handle.



Creating our Docker Image

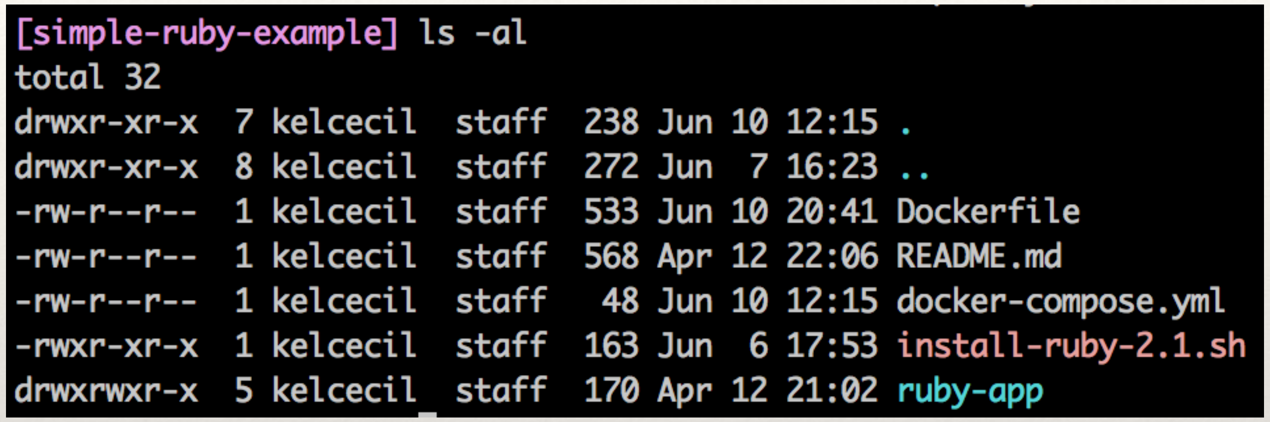
- Start with building a Dockerfile.
- Image contains our app, dependencies and other required items.
- Images are read-only.
  - Encourages immutable infrastructure!

Start containerizing our application by creating a Docker image.

An image is a read-only file system containing our application code, dependencies, and anything else we might need.

Images are what we share through Docker repositories. Their read-only nature encourages us to embrace the idea of immutable architecture.

Docker Build Context



```
[simple-ruby-example] ls -al
total 32
drwxr-xr-x  7 kelcecil  staff  238 Jun 10 12:15 .
drwxr-xr-x  8 kelcecil  staff  272 Jun  7 16:23 ..
-rw-r--r--  1 kelcecil  staff  533 Jun 10 20:41 Dockerfile
-rw-r--r--  1 kelcecil  staff  568 Apr 12 22:06 README.md
-rw-r--r--  1 kelcecil  staff   48 Jun 10 12:15 docker-compose.yml
-rwxr-xr-x  1 kelcecil  staff  163 Jun  6 17:53 install-ruby-2.1.sh
drwxrwxr-x  5 kelcecil  staff  170 Apr 12 21:02 ruby-app
```

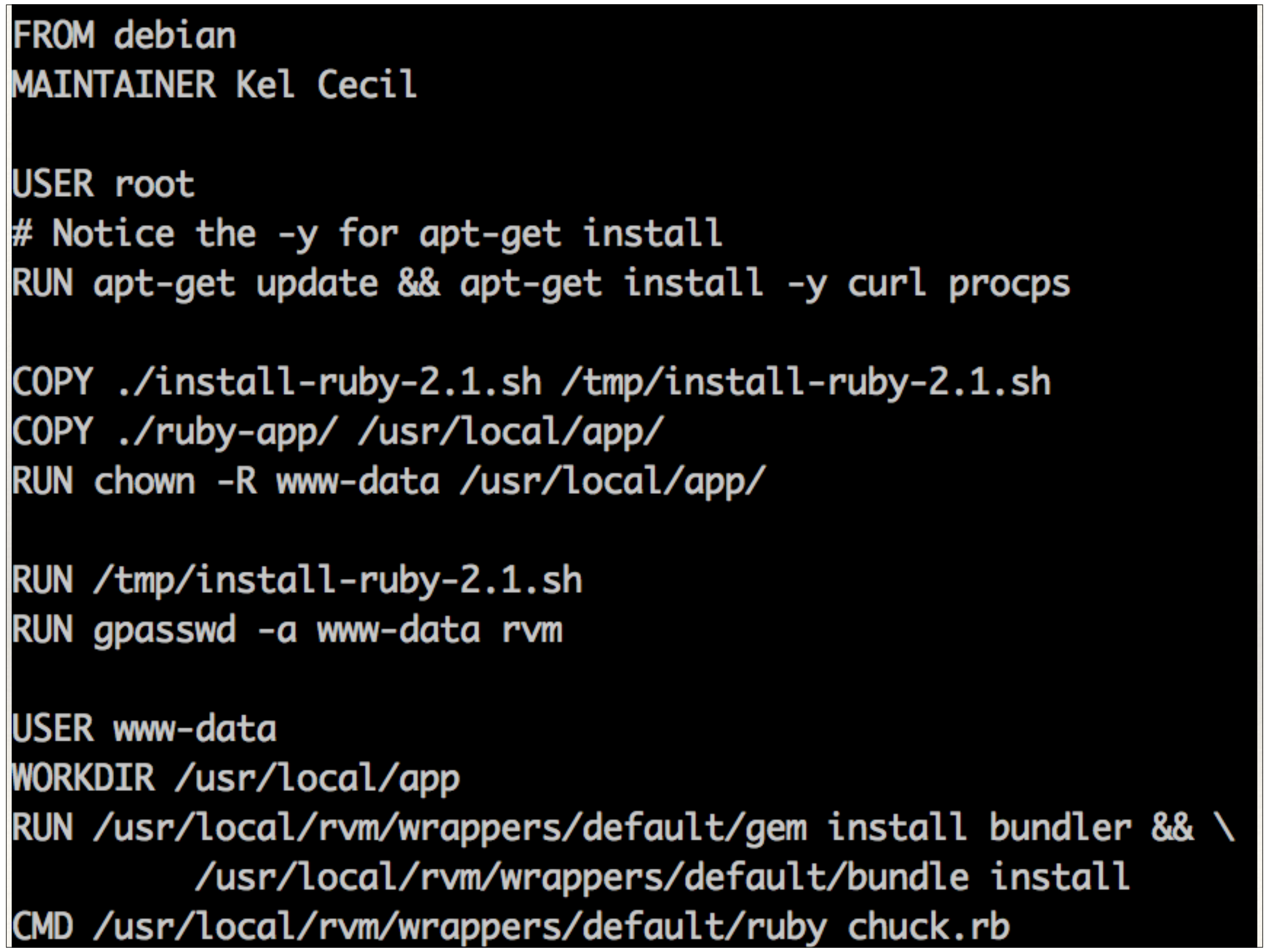
- Everything in context directory is sent when performing build.
- Anything in this context directory can be added to the image.

Let’s take a look at our app directory before we get into the Dockerfile.

Our directory contains a Dockerfile, our Ruby script in a folder, a bash script to install Ruby, a YAML file (which we’ll discuss later), and a markdown README file.

This directory is what we call the “build context” and is sent to the Docker daemon when we build our image.

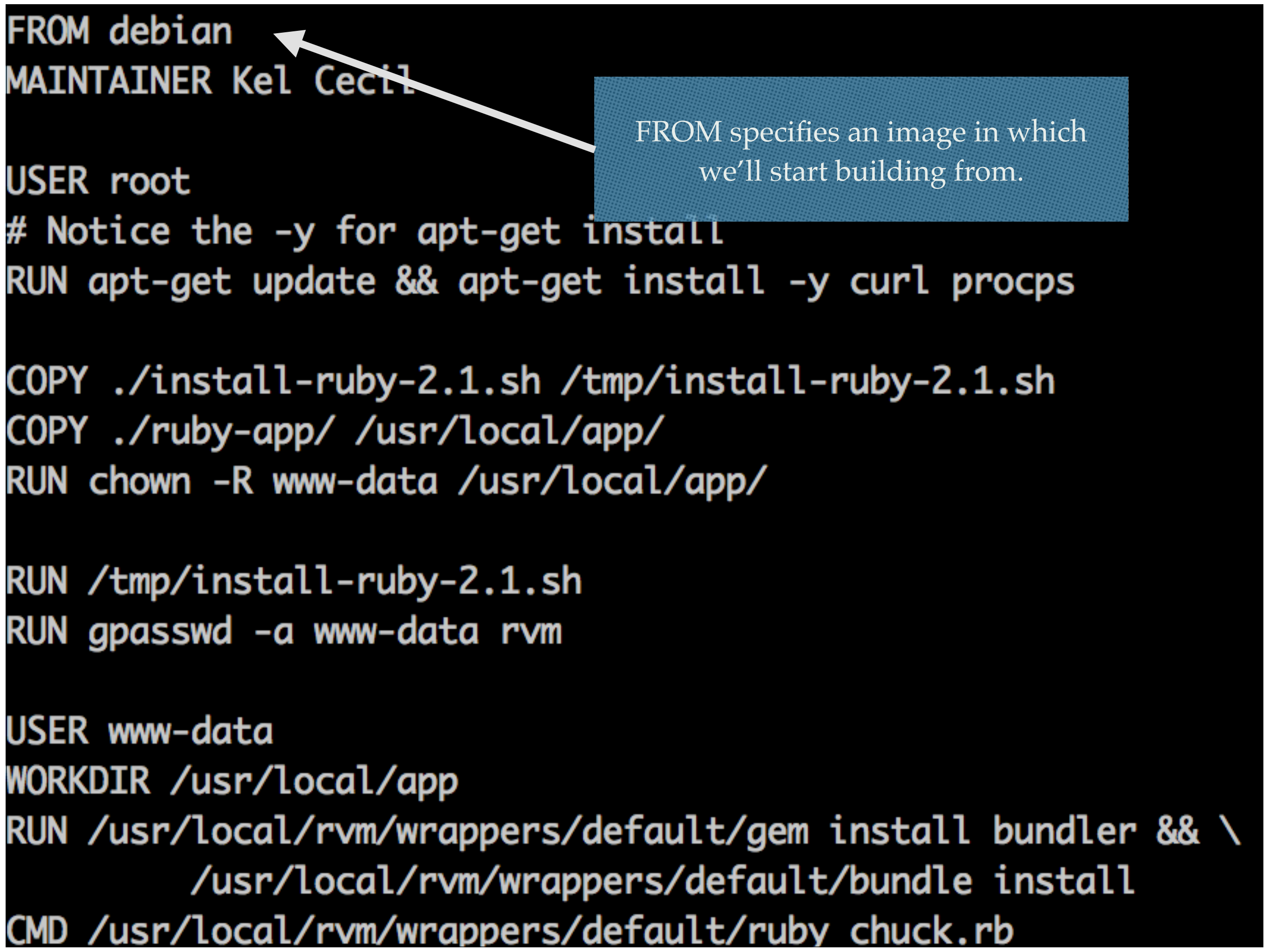
Everything in this directory and it’s descendants are sent to the Docker daemon. Aren’t necessary added to the Docker image. We’ll be specifically adding the files we need in our Dockerfile using commands I’ll show you in a few minutes.



```
FROM debian
MAINTAINER Kel Cecil

USER root
# Notice the -y for apt-get install
RUN apt-get update && apt-get install -y curl procps

COPY ./install-ruby-2.1.sh /tmp/install-ruby-2.1.sh
COPY ./ruby-app/ /usr/local/app/
RUN chown -R www-data /usr/local/app/

RUN /tmp/install-ruby-2.1.sh
RUN gpasswd -a www-data rvm

USER www-data
WORKDIR /usr/local/app
RUN /usr/local/rvm/wrappers/default/gem install bundler && \
         /usr/local/rvm/wrappers/default/bundle install
CMD /usr/local/rvm/wrappers/default/ruby chuck.rb
```

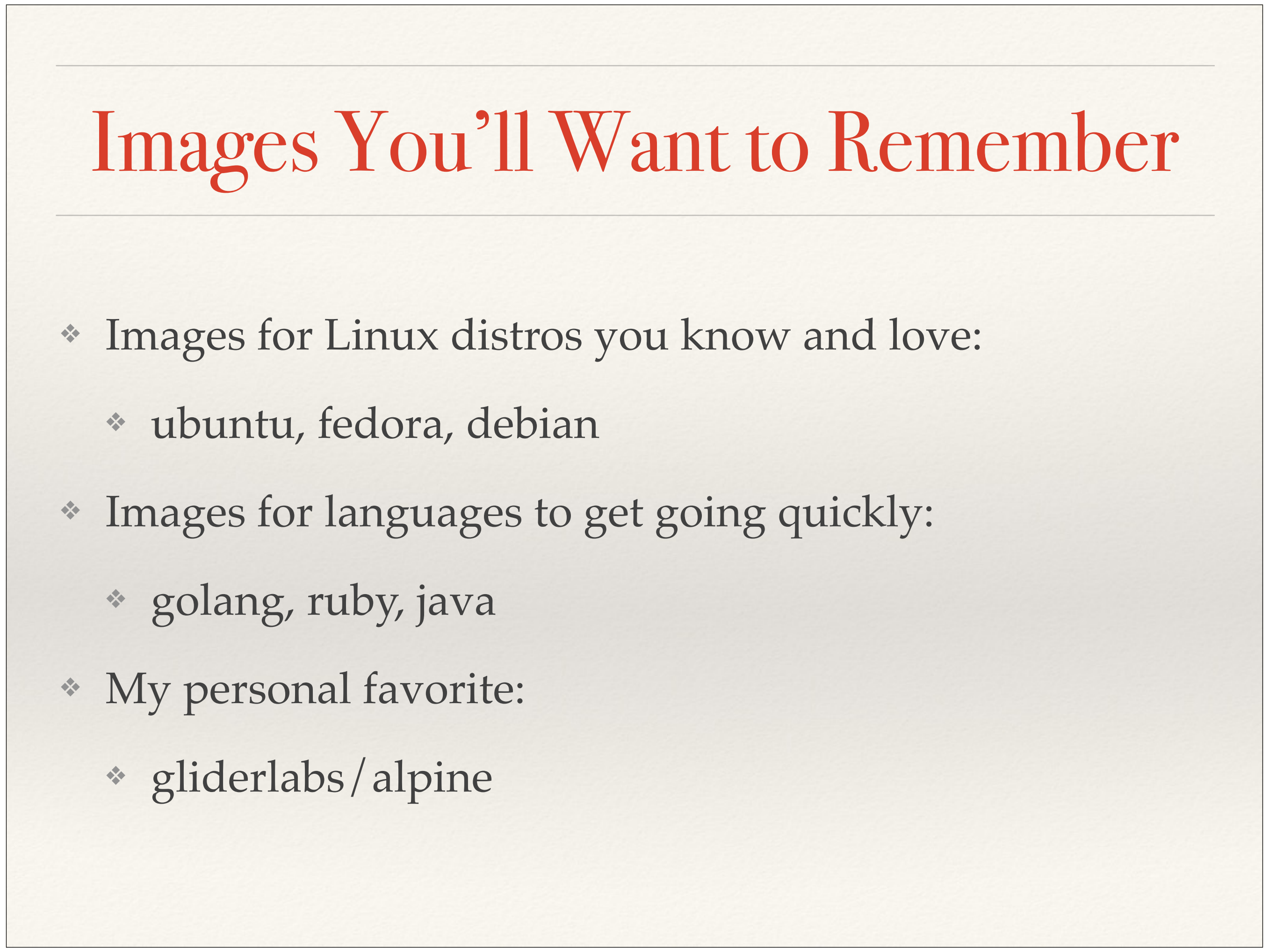
Here's a full completed Dockerfile that creates our image to distribute our container. Let's take a minute to take in how awesome this is.

There's few things going on, so let's check this out step by step.


```
FROM debian
MAINTAINER Kel Cecil

USER root
# Notice the -y for apt-get install
RUN apt-get update && apt-get install -y curl procps

COPY ./install-ruby-2.1.sh /tmp/install-ruby-2.1.sh
COPY ./ruby-app/ /usr/local/app/
RUN chown -R www-data /usr/local/app/

RUN /tmp/install-ruby-2.1.sh
RUN gpasswd -a www-data rvm

USER www-data
WORKDIR /usr/local/app
RUN /usr/local/rvm/wrappers/default/gem install bundler && \
         /usr/local/rvm/wrappers/default/bundle install
CMD /usr/local/rvm/wrappers/default/ruby chuck.rb
```

FROM specifies an image in which we'll start building from.

FROM specifies an image from which we'll build from. You can start with any image name or hash you like. Docker will pull the image from a remote registry if the the image isn't already on your machine.



Images You'll Want to Remember

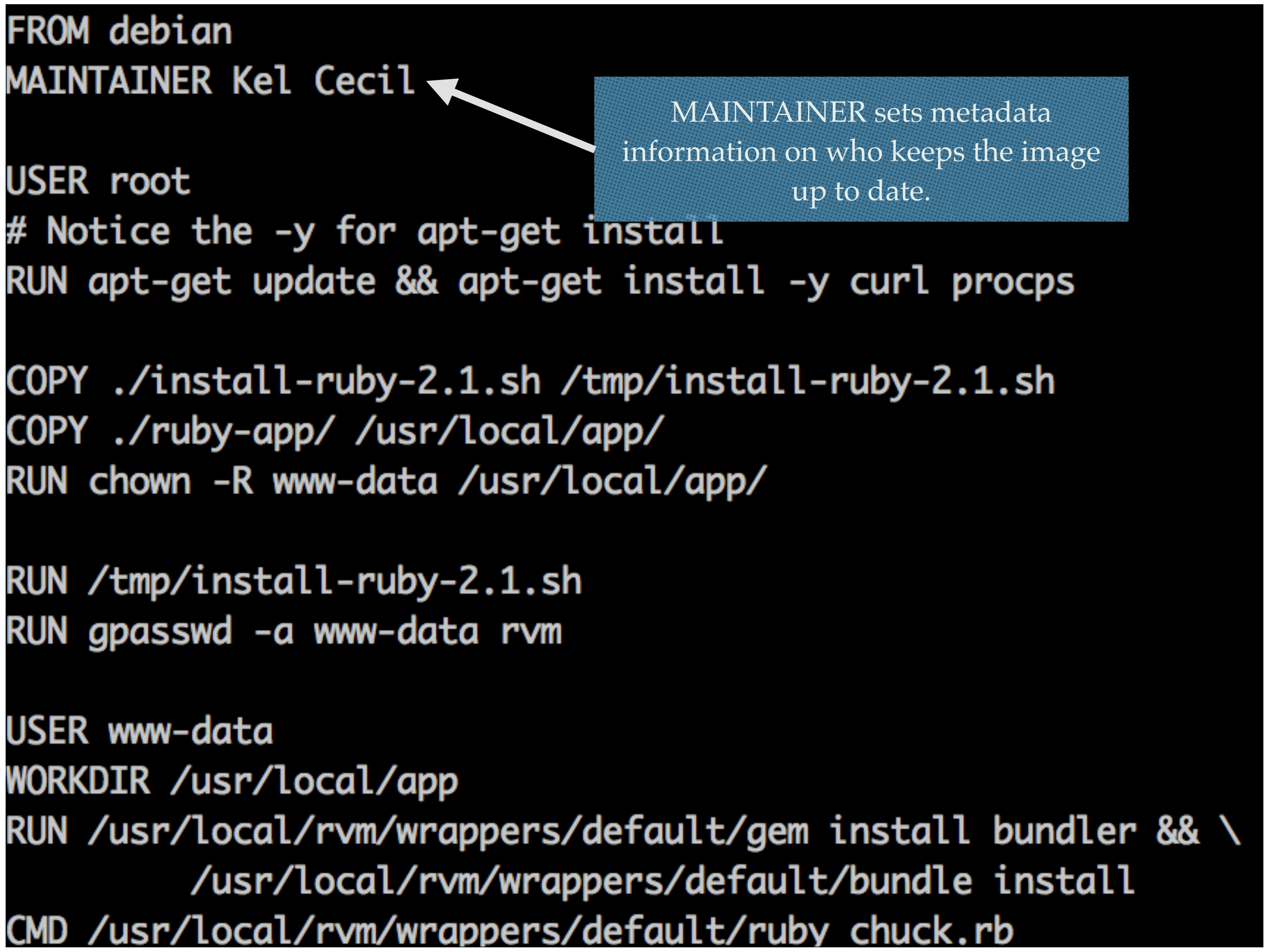
- Images for Linux distros you know and love:
  - ubuntu, fedora, debian
- Images for languages to get going quickly:
  - golang, ruby, java
- My personal favorite:
  - gliderlabs/alpine

Lots of images to use when selecting a base image.

Images for distributions you already know: ubuntu, fedora, debian.

Images for languages to get started in an environment quickly: golang, ruby, java

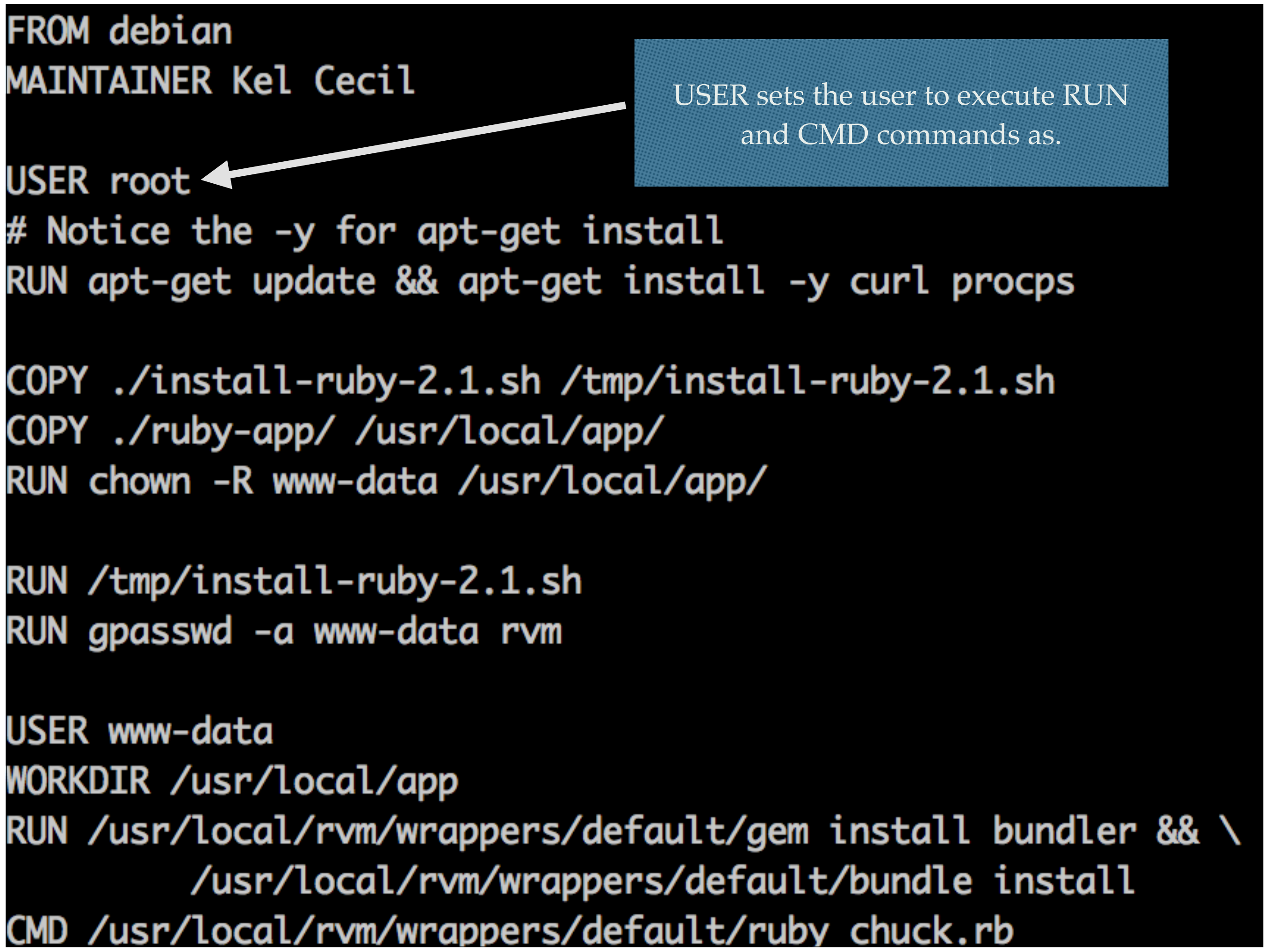
My personal favorite image is Glider Lab's Alpine Linux.
- Optimized to be incredibly small to save disk space and time when initially pulling and updating your images.
- Includes a package system to easily install anything you might need.
- Give it a try!



```
FROM debian
MAINTAINER Kel Cecil

USER root
# Notice the -y for apt-get install
RUN apt-get update && apt-get install -y curl procps

COPY ./install-ruby-2.1.sh /tmp/install-ruby-2.1.sh
COPY ./ruby-app/ /usr/local/app/
RUN chown -R www-data /usr/local/app/

RUN /tmp/install-ruby-2.1.sh
RUN gpasswd -a www-data rvm

USER www-data
WORKDIR /usr/local/app
RUN /usr/local/rvm/wrappers/default/gem install bundler && \
         /usr/local/rvm/wrappers/default/bundle install
CMD /usr/local/rvm/wrappers/default/ruby chuck.rb
```

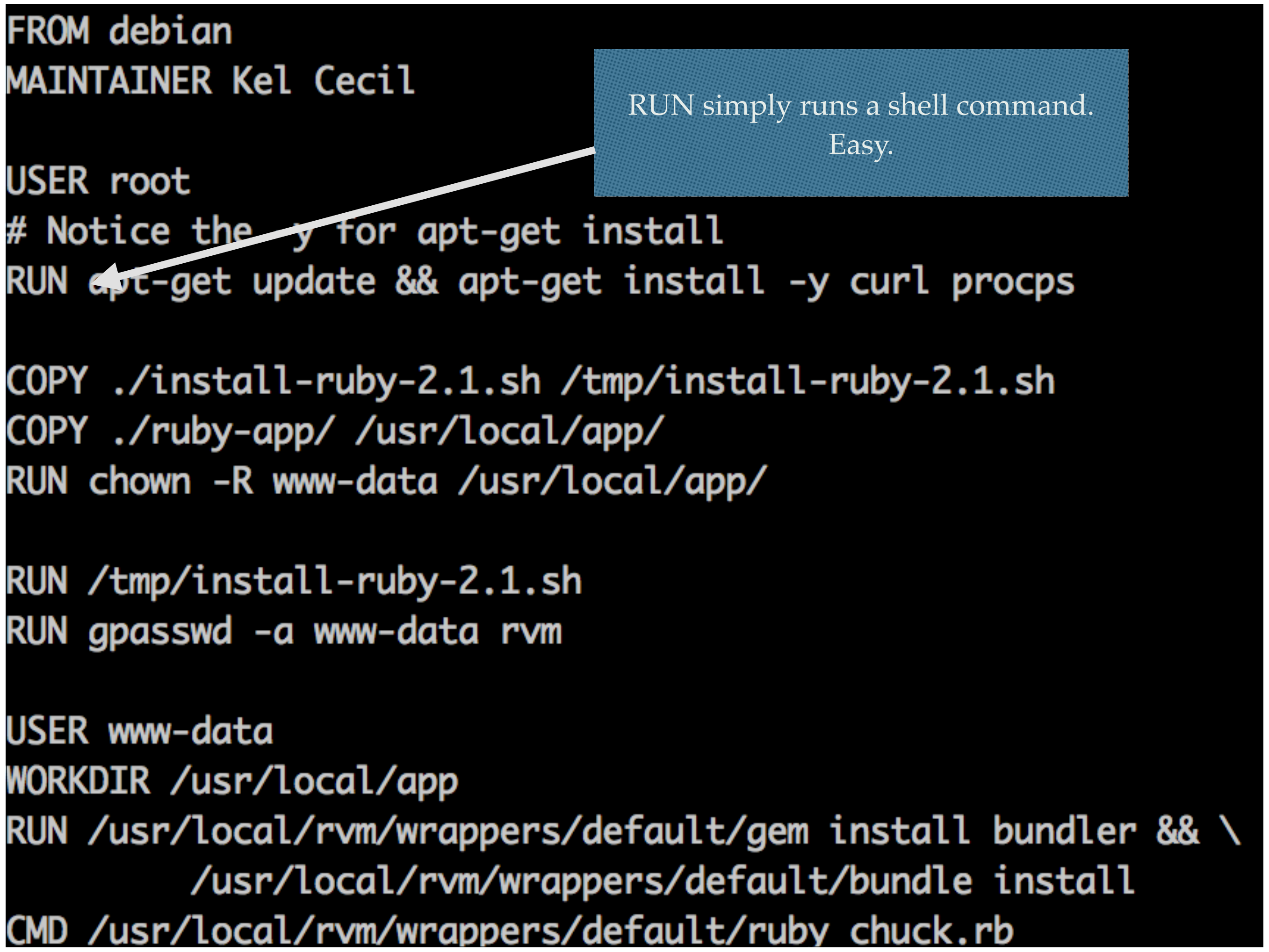
MAINTAINER sets metadata information on who keeps the image up to date.

Maintainer is just a simple line that includes some metadata for the image. You can safely omit this.



```
FROM debian
MAINTAINER Kel Cecil

USER root
# Notice the -y for apt-get install
RUN apt-get update && apt-get install -y curl procps

COPY ./install-ruby-2.1.sh /tmp/install-ruby-2.1.sh
COPY ./ruby-app/ /usr/local/app/
RUN chown -R www-data /usr/local/app/

RUN /tmp/install-ruby-2.1.sh
RUN gpasswd -a www-data rvm

USER www-data
WORKDIR /usr/local/app
RUN /usr/local/rvm/wrappers/default/gem install bundler && \
         /usr/local/rvm/wrappers/default/bundle install
CMD /usr/local/rvm/wrappers/default/ruby chuck.rb
```

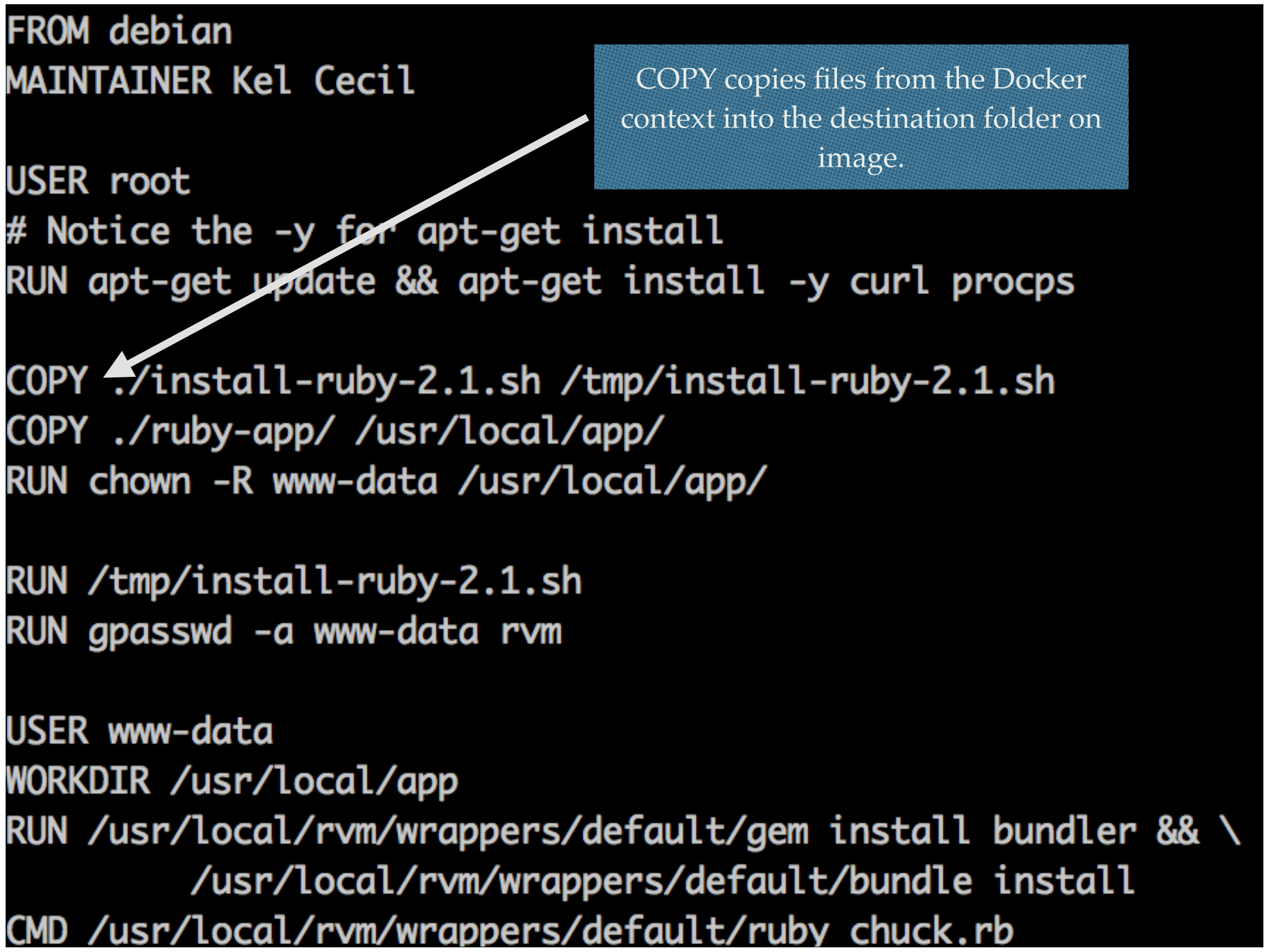
USER sets the user we'll be running RUN and CMD as. Docker best practices suggests to use root until you're finished setting up your system dependencies to avoid potential complications with sudo and switching to another user for running your application.



```
FROM debian
MAINTAINER Kel Cecil

USER root
# Notice the -y for apt-get install
RUN apt-get update && apt-get install -y curl procps

COPY ./install-ruby-2.1.sh /tmp/install-ruby-2.1.sh
COPY ./ruby-app/ /usr/local/app/
RUN chown -R www-data /usr/local/app/

RUN /tmp/install-ruby-2.1.sh
RUN gpasswd -a www-data rvm

USER www-data
WORKDIR /usr/local/app
RUN /usr/local/rvm/wrappers/default/gem install bundler && \
         /usr/local/rvm/wrappers/default/bundle install
CMD /usr/local/rvm/wrappers/default/ruby chuck.rb
```

RUN simply runs a shell command. Easy.

RUN executes a command. You can perform systems tasks or run shell scripts to get your image to where it needs to be to run your application. This step runs when you run docker build.

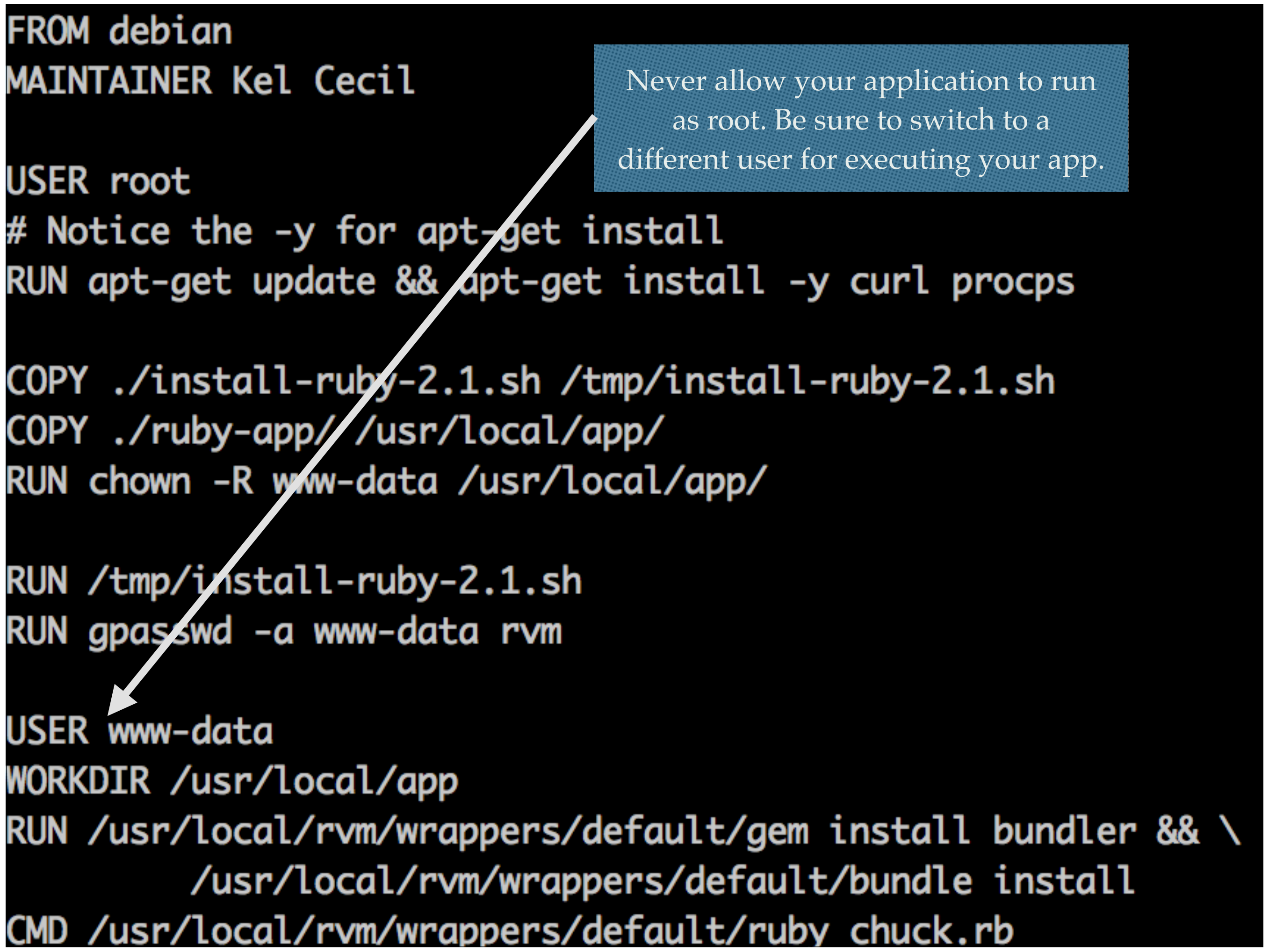


```
FROM debian
MAINTAINER Kel Cecil

USER root
# Notice the -y for apt-get install
RUN apt-get update && apt-get install -y curl procps

COPY ./install-ruby-2.1.sh /tmp/install-ruby-2.1.sh
COPY ./ruby-app/ /usr/local/app/
RUN chown -R www-data /usr/local/app/

RUN /tmp/install-ruby-2.1.sh
RUN gpasswd -a www-data rvm

USER www-data
WORKDIR /usr/local/app
RUN /usr/local/rvm/wrappers/default/gem install bundler && \
         /usr/local/rvm/wrappers/default/bundle install
CMD /usr/local/rvm/wrappers/default/ruby chuck.rb
```

COPY copies files from the Docker context into the destination folder on image.

COPY copies files from the Docker context into the destination folder on the image.

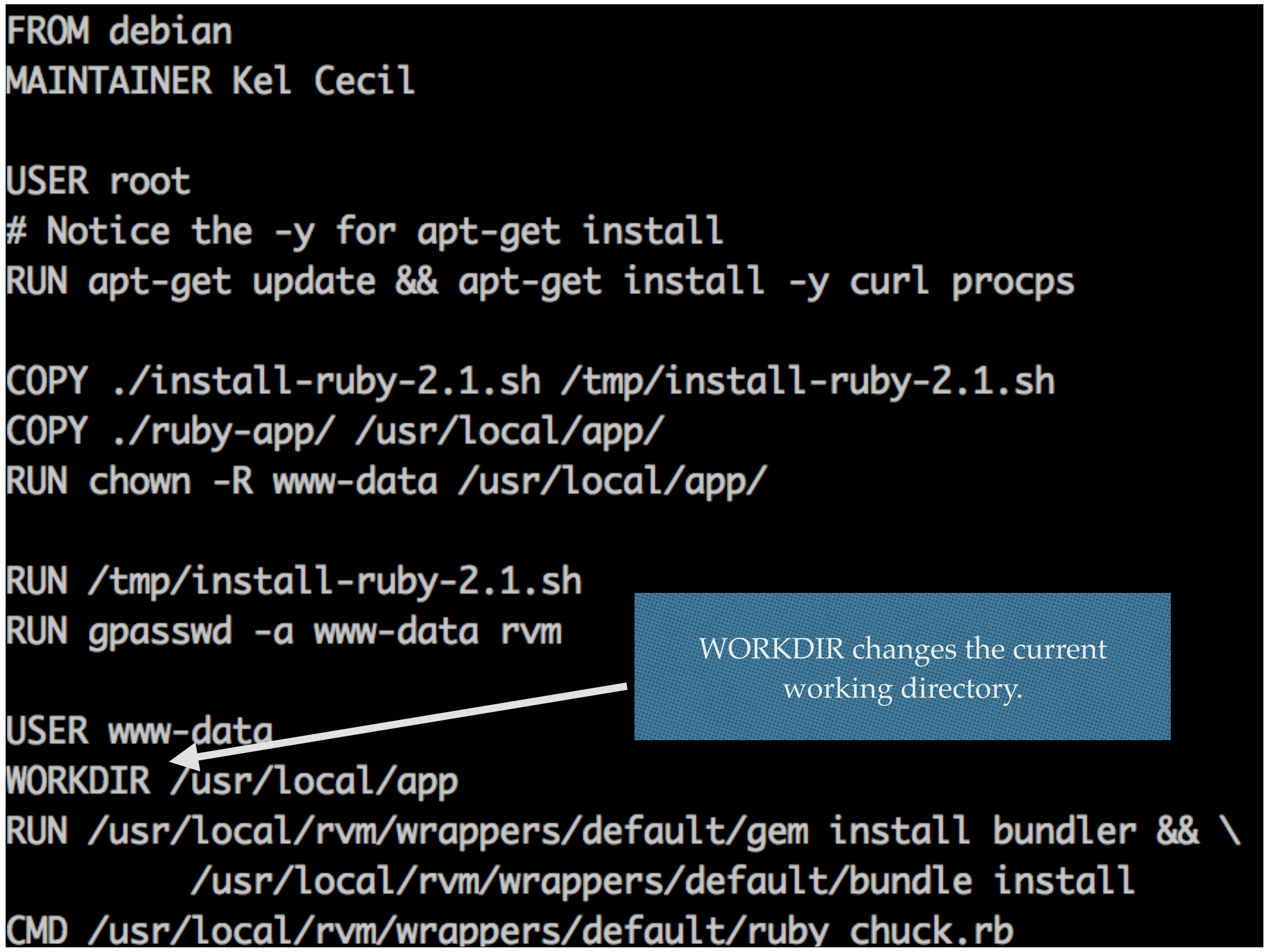
Both COPY and RUN will both be executed when during docker build.



```
FROM debian
MAINTAINER Kel Cecil

USER root
# Notice the -y for apt-get install
RUN apt-get update && apt-get install -y curl procps

COPY ./install-ruby-2.1.sh /tmp/install-ruby-2.1.sh
COPY ./ruby-app/ /usr/local/app/
RUN chown -R www-data /usr/local/app/

RUN /tmp/install-ruby-2.1.sh
RUN gpasswd -a www-data rvm

USER www-data
WORKDIR /usr/local/app
RUN /usr/local/rvm/wrappers/default/gem install bundler && \
         /usr/local/rvm/wrappers/default/bundle install
CMD /usr/local/rvm/wrappers/default/ruby chuck.rb
```

Never allow your application to run as root. Be sure to switch to a different user for executing your app.

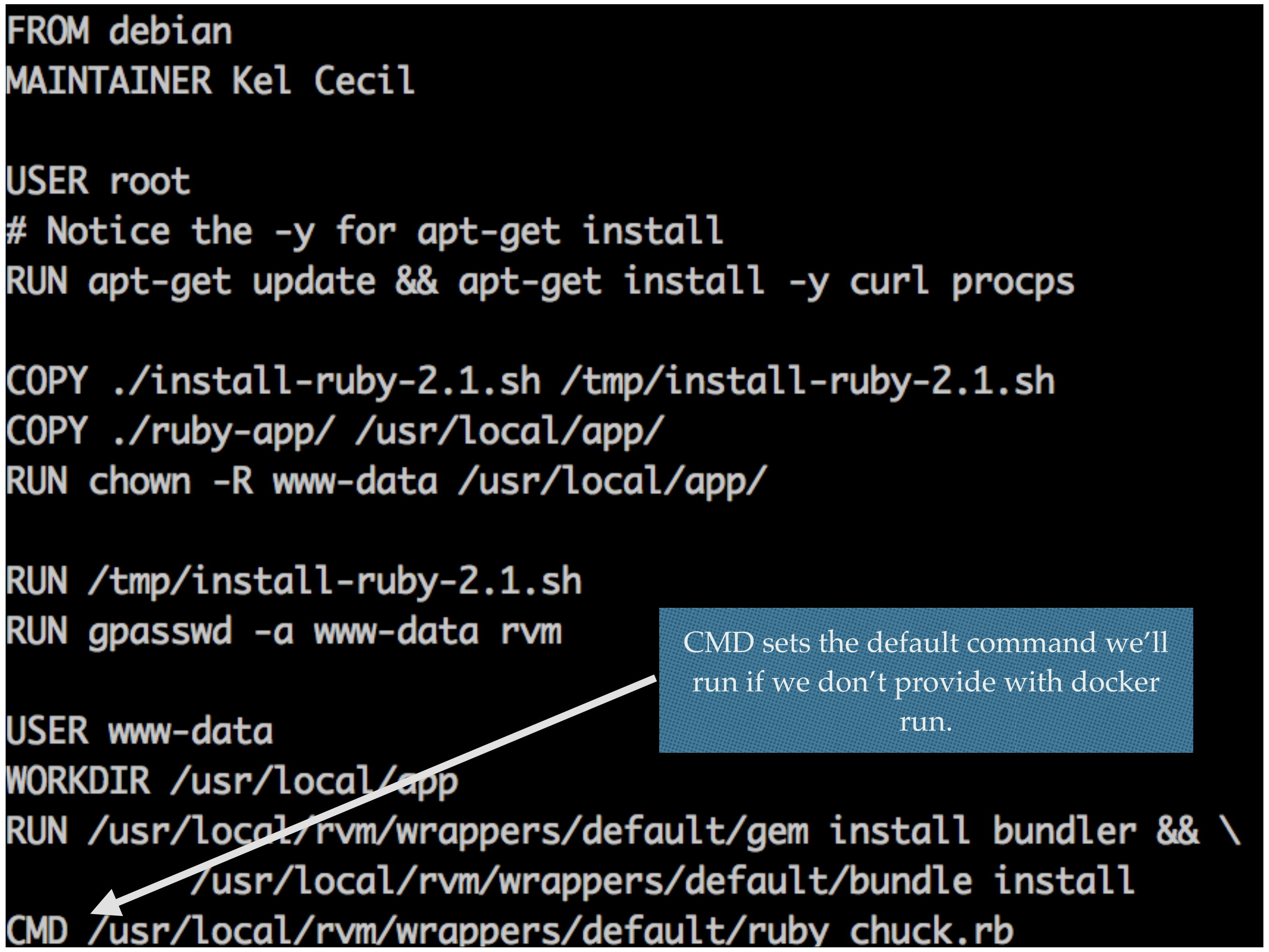
Don’t forget. Don’t run your application as root.



```
FROM debian
MAINTAINER Kel Cecil

USER root
# Notice the -y for apt-get install
RUN apt-get update && apt-get install -y curl procps

COPY ./install-ruby-2.1.sh /tmp/install-ruby-2.1.sh
COPY ./ruby-app/ /usr/local/app/
RUN chown -R www-data /usr/local/app/

RUN /tmp/install-ruby-2.1.sh
RUN gpasswd -a www-data rvm

USER www-data
WORKDIR /usr/local/app
RUN /usr/local/rvm/wrappers/default/gem install bundler && \
         /usr/local/rvm/wrappers/default/bundle install
CMD /usr/local/rvm/wrappers/default/ruby chuck.rb
```

WORKDIR changes the current working directory.

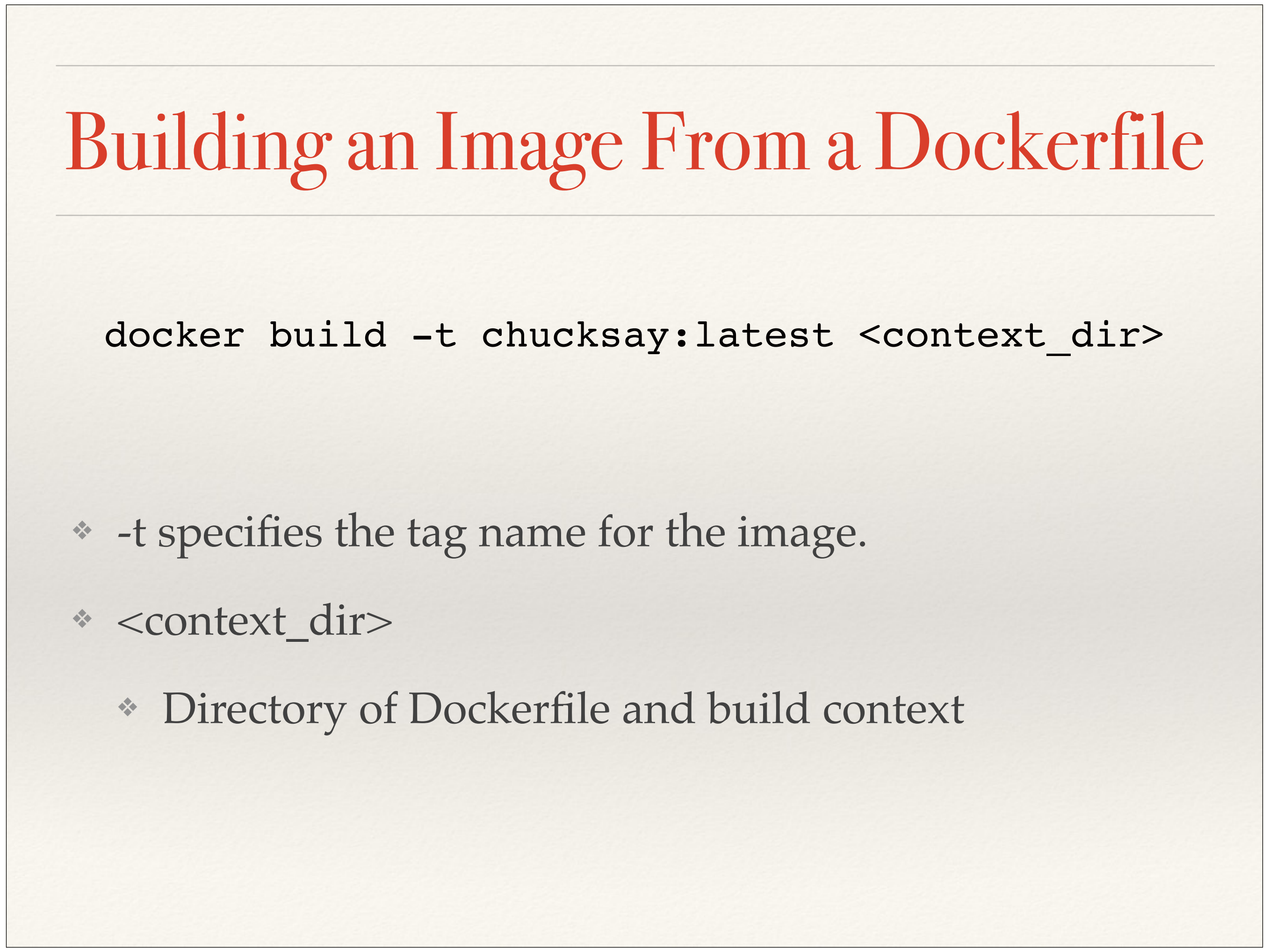
WORKDIR just sets the working directory for running applications. You'll notice I take advantage of this to run Ruby's bundler in the app's directory.



```
FROM debian
MAINTAINER Kel Cecil

USER root
# Notice the -y for apt-get install
RUN apt-get update && apt-get install -y curl procps

COPY ./install-ruby-2.1.sh /tmp/install-ruby-2.1.sh
COPY ./ruby-app/ /usr/local/app/
RUN chown -R www-data /usr/local/app/

RUN /tmp/install-ruby-2.1.sh
RUN gpasswd -a www-data rvm

USER www-data
WORKDIR /usr/local/app
RUN /usr/local/rvm/wrappers/default/gem install bundler && \
         /usr/local/rvm/wrappers/default/bundle install
CMD /usr/local/rvm/wrappers/default/ruby chuck.rb
```

CMD sets the default command we'll run if we don't provide with docker run.

CMD sets the default command we'll run if we don't provide a command to run when invoking docker run. This can easily be override when we run our container, and it can be omitted now if we don't want to provide a default.



Building an Image From a Dockerfile

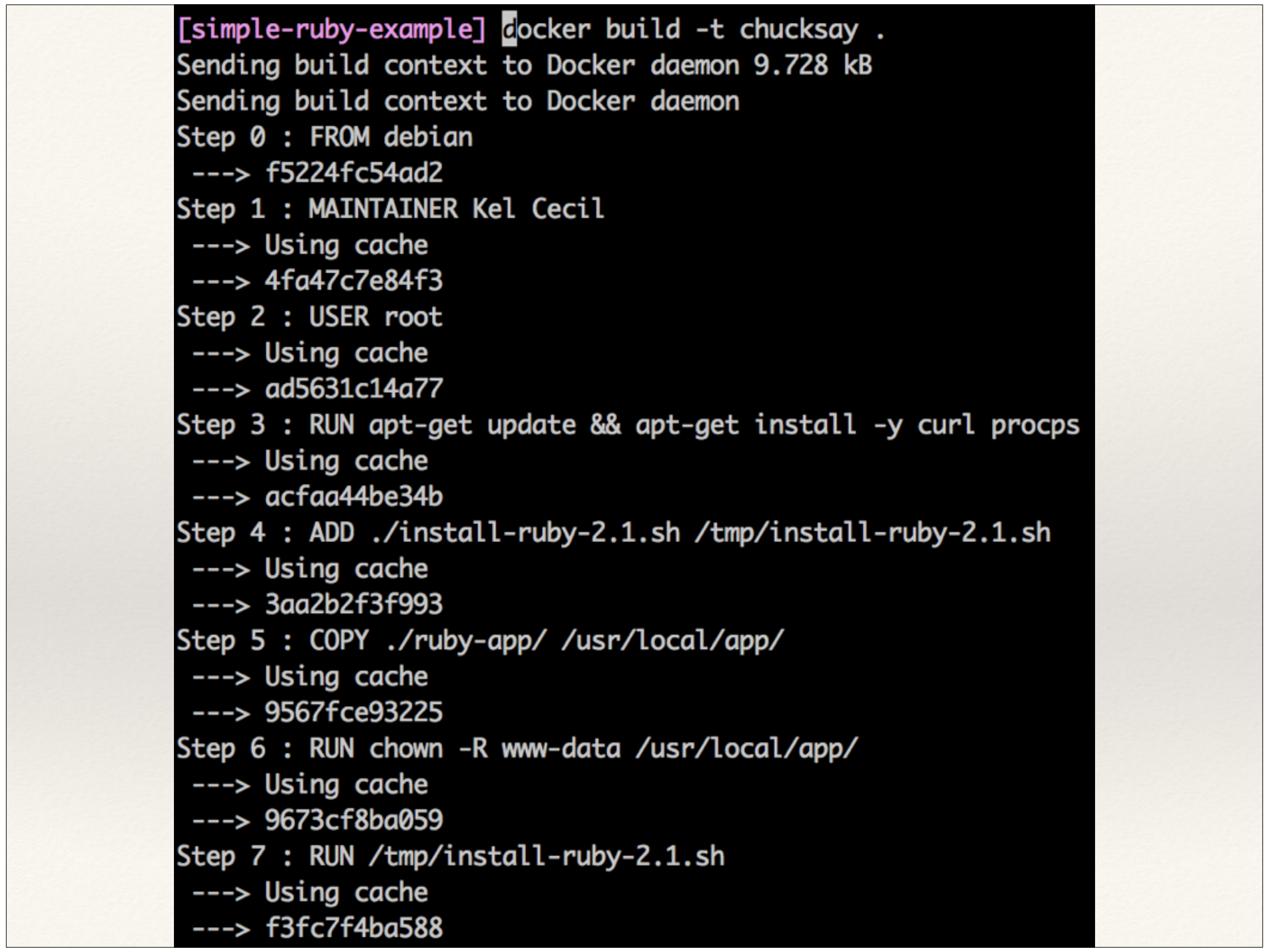
```
docker build -t chucksay:latest <context_dir>
```

- -t specifies the tag name for the image.
- <context_dir>
  - Directory of Dockerfile and build context

We now want to build our image from our Dockerfile.

We'll use the -t parameter to specify a tag name to apply to an image.

Our final parameter is the directory of the Dockerfile and the build context. The directory will be sent to the Docker daemon to build our image.



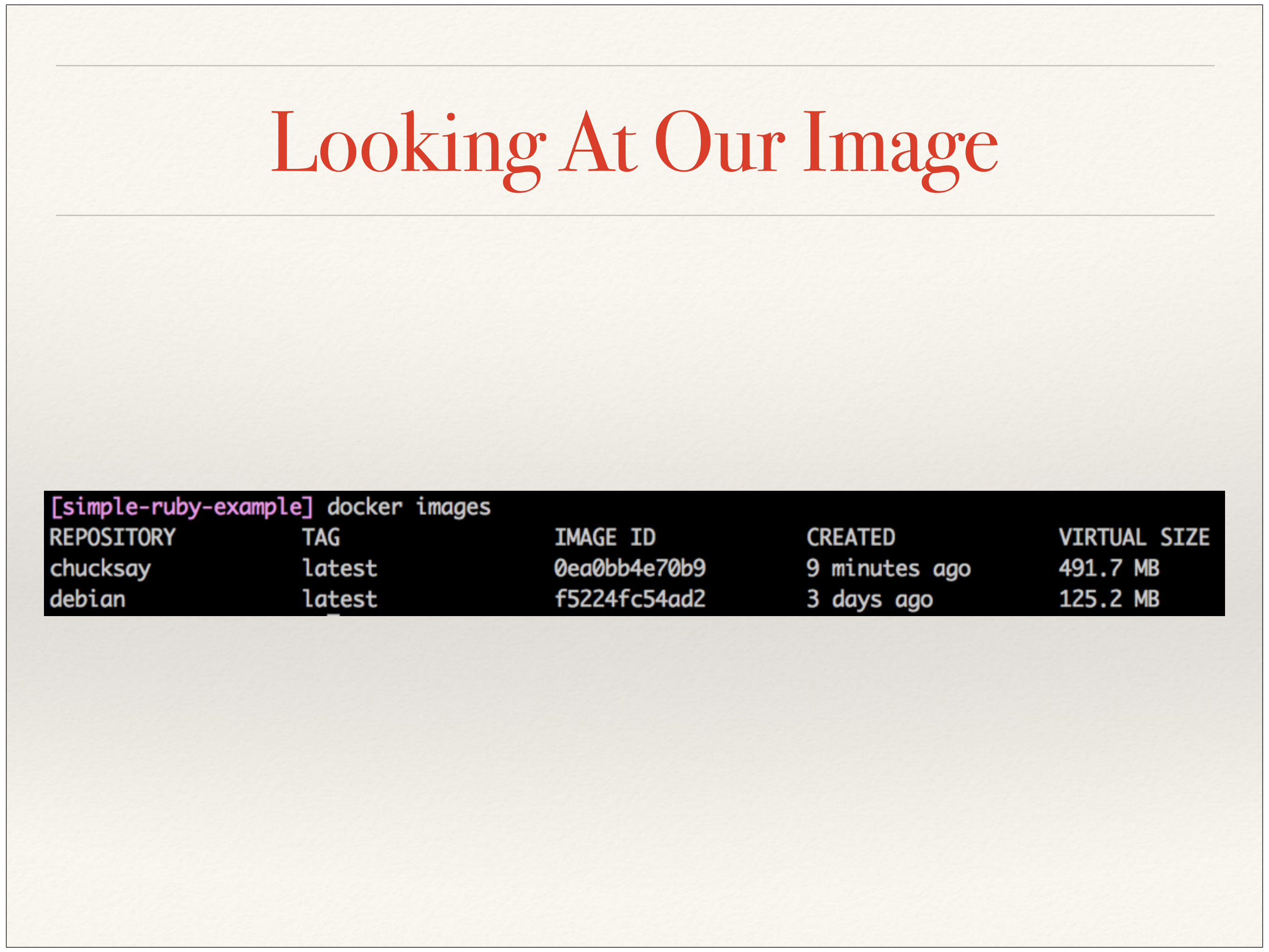
```
[simple-ruby-example] docker build -t chucksay .
Sending build context to Docker daemon 9.728 kB
Sending build context to Docker daemon
Step 0 : FROM debian
 ---> f5224fc54ad2
Step 1 : MAINTAINER Kel Cecil
 ---> Using cache
 ---> 4fa47c7e84f3
Step 2 : USER root
 ---> Using cache
 ---> ad5631c14a77
Step 3 : RUN apt-get update && apt-get install -y curl procps
 ---> Using cache
 ---> acfaa44be34b
Step 4 : ADD ./install-ruby-2.1.sh /tmp/install-ruby-2.1.sh
 ---> Using cache
 ---> 3aa2b2f3f993
Step 5 : COPY ./ruby-app/ /usr/local/app/
 ---> Using cache
 ---> 9567fce93225
Step 6 : RUN chown -R www-data /usr/local/app/
 ---> Using cache
 ---> 9673cf8ba059
Step 7 : RUN /tmp/install-ruby-2.1.sh
 ---> Using cache
 ---> f3fc7f4ba588
```

Here’s output from running Docker Build.

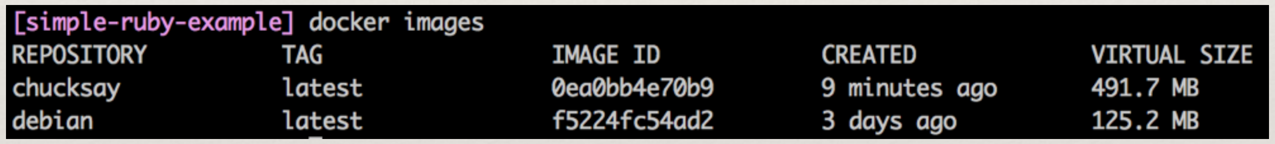
Each line in Dockerfile is executed and an independent image is created. These images are called layers.
These layer are put together when the container is created thanks to Docker’s Union File System.

You’ll notice that every step in my output specifies that it’s using cache.
Docker has heuristics built in to know when to rebuild your layers.
Check out the Dockerfile Best Practices for more information on this works.

Image caching is fantastic for reducing the layers needed to be rebuilt or transferred when updating an image. Translates to faster deploys for you!



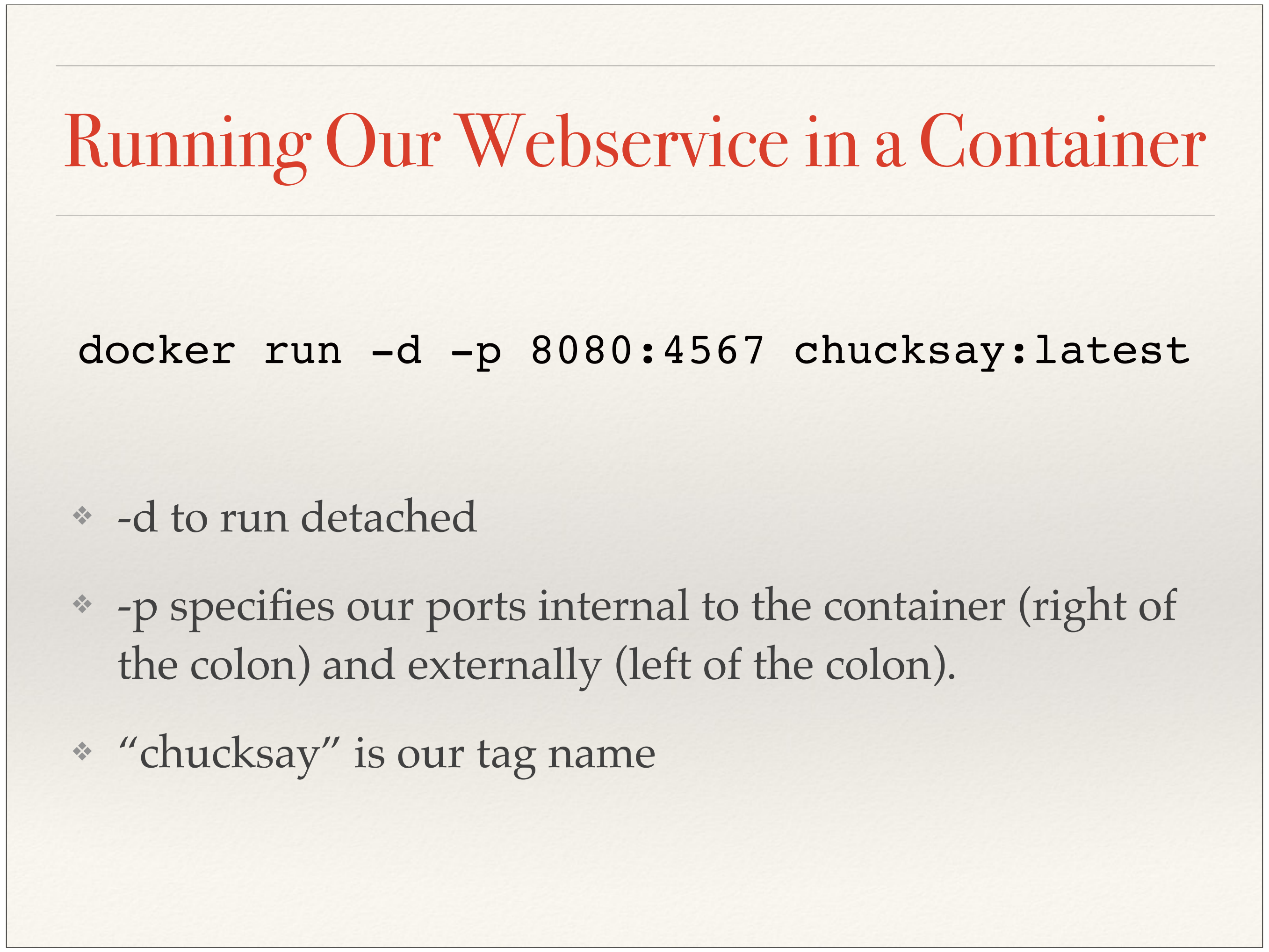
Looking At Our Image



```
[simple-ruby-example] docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
chucksay            latest              0ea0bb4e70b9        9 minutes ago       491.7 MB
debian              latest              f5224fc54ad2        3 days ago          125.2 MB
```

Check out the image we just built with docker images.

We can see both our chucksay image as well as the debian image that we built from.
We can also see our generated image ID.



Running Our Webservice in a Container

```
docker run -d -p 8080:4567 chucksay:latest
```

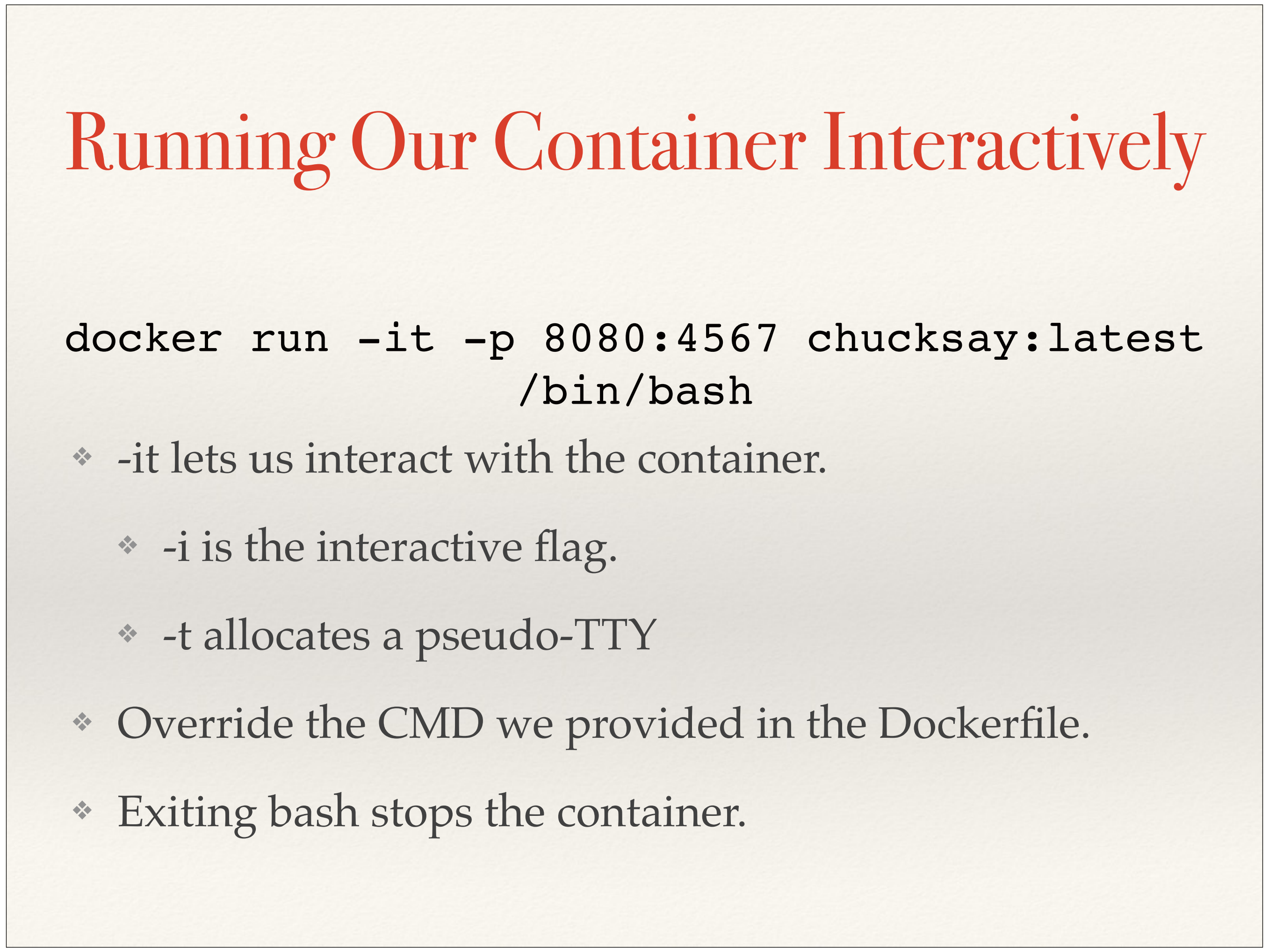
- -d to run detached
- -p specifies our ports internal to the container (right of the colon) and externally (left of the colon).
- "chucksay" is our tag name

Let's try running our webapp. We'll use docker run to create and run our container.

-d runs the container in a detached state that's great for running web services.

-p specifies our ports that we'll be using to communicate with our app internal to the container and externally. Why do this?
- Might want to serve two apps that are both configured to serve over HTTP port 8080.
- We obviously can't do that.
- We don't want to change our app configuration for each container to serve over a different port.
- We can simply tell Docker the port the application will talk to inside the container and map it to a port that will be exposed on the server.
- This enables much easier multi-tenancy on servers without per-container application configuration changes.

We end with specifying our image and tag name.



Running Our Container Interactively

```
docker run -it -p 8080:4567 chucksay:latest
                  /bin/bash
```

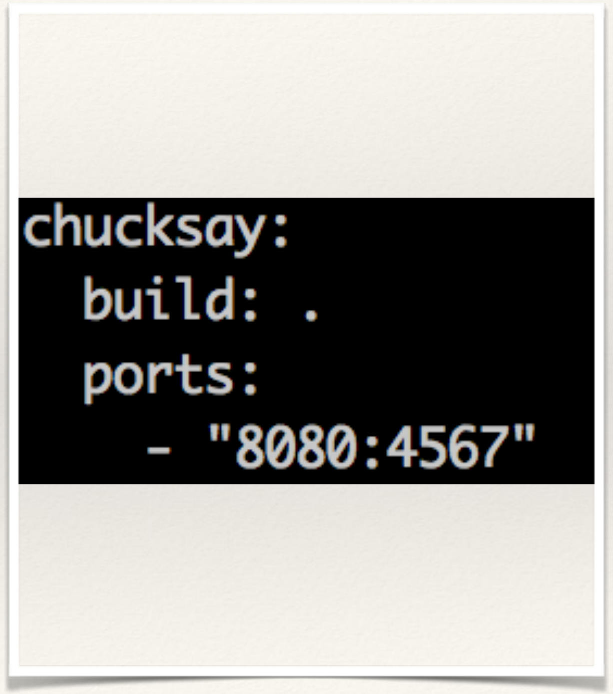
- -it lets us interact with the container.
  - -i is the interactive flag.
  - -t allocates a pseudo-TTY
- Override the CMD we provided in the Dockerfile.
- Exiting bash stops the container.

We can run our container interactively to play around if we'd like.
For example, let's start bash and play around in our image.

We can use -it. This is actually two parameters that both enable interactivity through -i and create a pseudo-TTY interface with -t.
We again specify the image we want to run.
We override the CMD we provided in the Dockerfile to run /bin/bash. Omitting as we saw earlier will let us use the default CMD.

Docker Compose (fig)

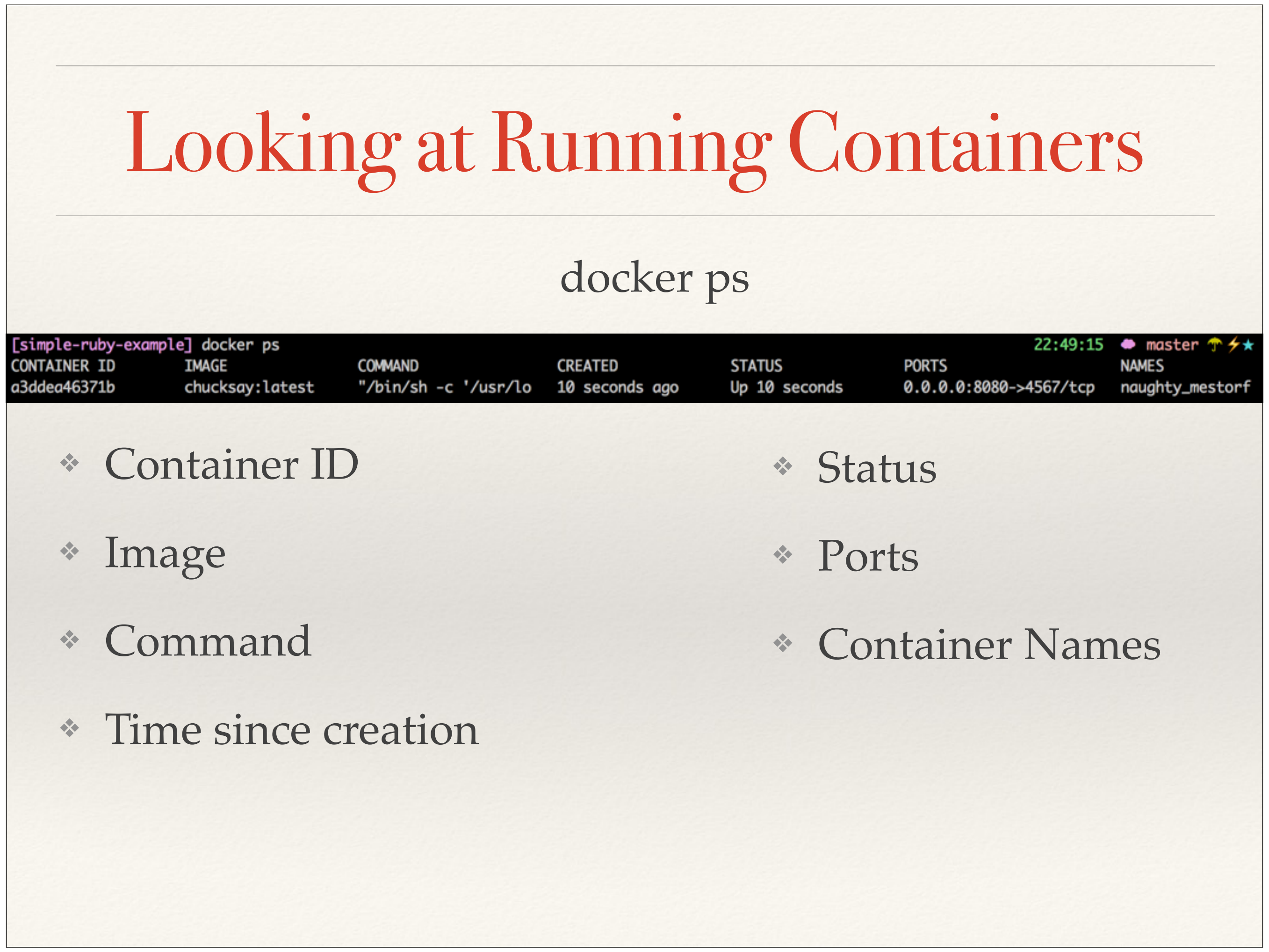
- Keep container parameters in a JSON or YAML file.
- Build or run your application quite easily:
  - `docker-compose up -d`



```
chucksay:
  build: .
  ports:
    - "8080:4567"
```

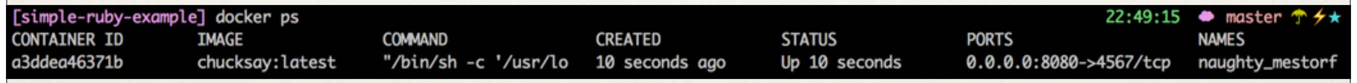
Typing out the docker run command can be more complex as continue to learn about advanced Docker features.
Don’t always want to type out that docker run command.
It can be a pain to share those instructions with others.

Docker Compose can help you share parameters for containers to easily build or run your containers.
A simple docker-compose YAML file is included in the example repository to try out!



Looking at Running Containers

docker ps



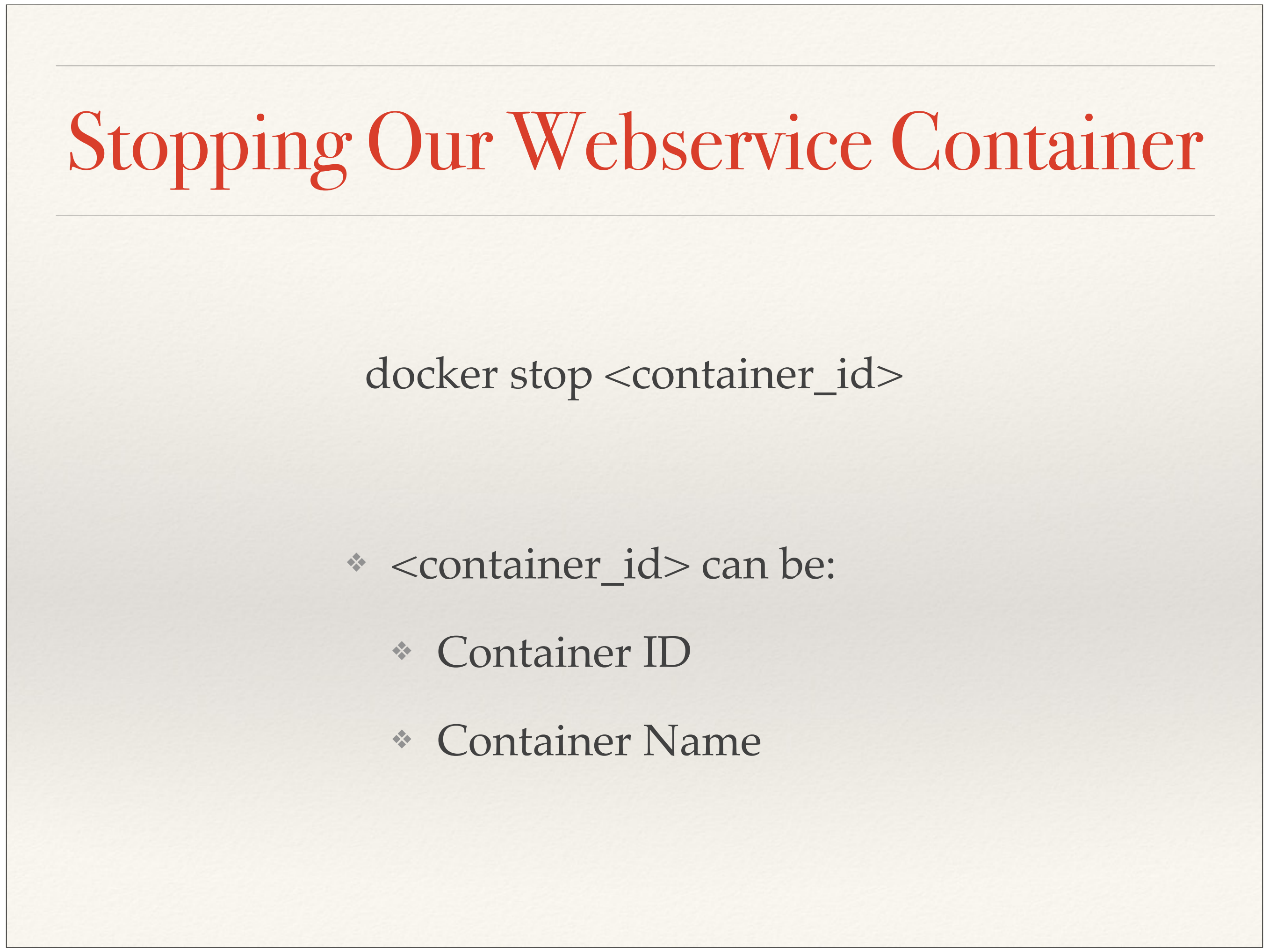
```
[simple-ruby-example] docker ps                                                                          22:49:15  master
CONTAINER ID        IMAGE               COMMAND              CREATED             STATUS              PORTS                    NAMES
a3ddea46371b        chucksay:latest     "/bin/sh -c '/usr/lo  10 seconds ago      Up 10 seconds       0.0.0.0:8080->4567/tcp   naughty_mestorf
```

- Container ID
- Image
- Command
- Time since creation
- Status
- Ports
- Container Names

Apologize for the output being a bit too wide to display well.

We can see containers that are currently running with some helpful formation-
Container ID - A hash identifier uniquely identifying our running container.
Image - The repository and version that is running in the container.
Command - What is running inside the container
Created - Time since creation.
Status - An brief status of how our container is doing.
Ports - A quick reference of what ports are being redirected (think back to our -p argument and rerouting port 8080 to 4567.)
Names - An easy to remember name.

    You can assign this name, or Docker will generate one for you.
    Docker can generate some pretty amusing names.



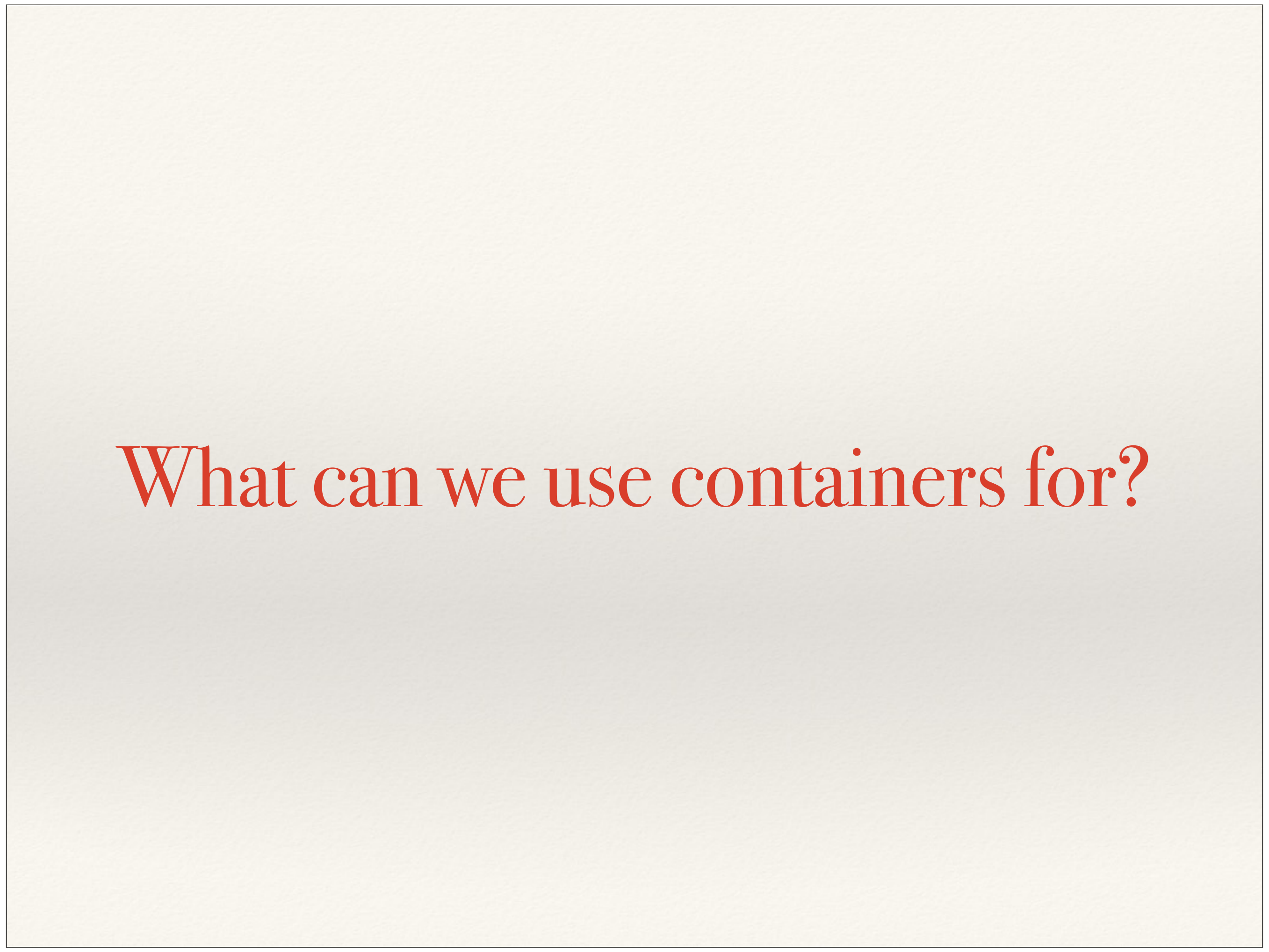
Stopping Our Webservice Container

docker stop <container_id>

- <container_id> can be:
  - Container ID
  - Container Name

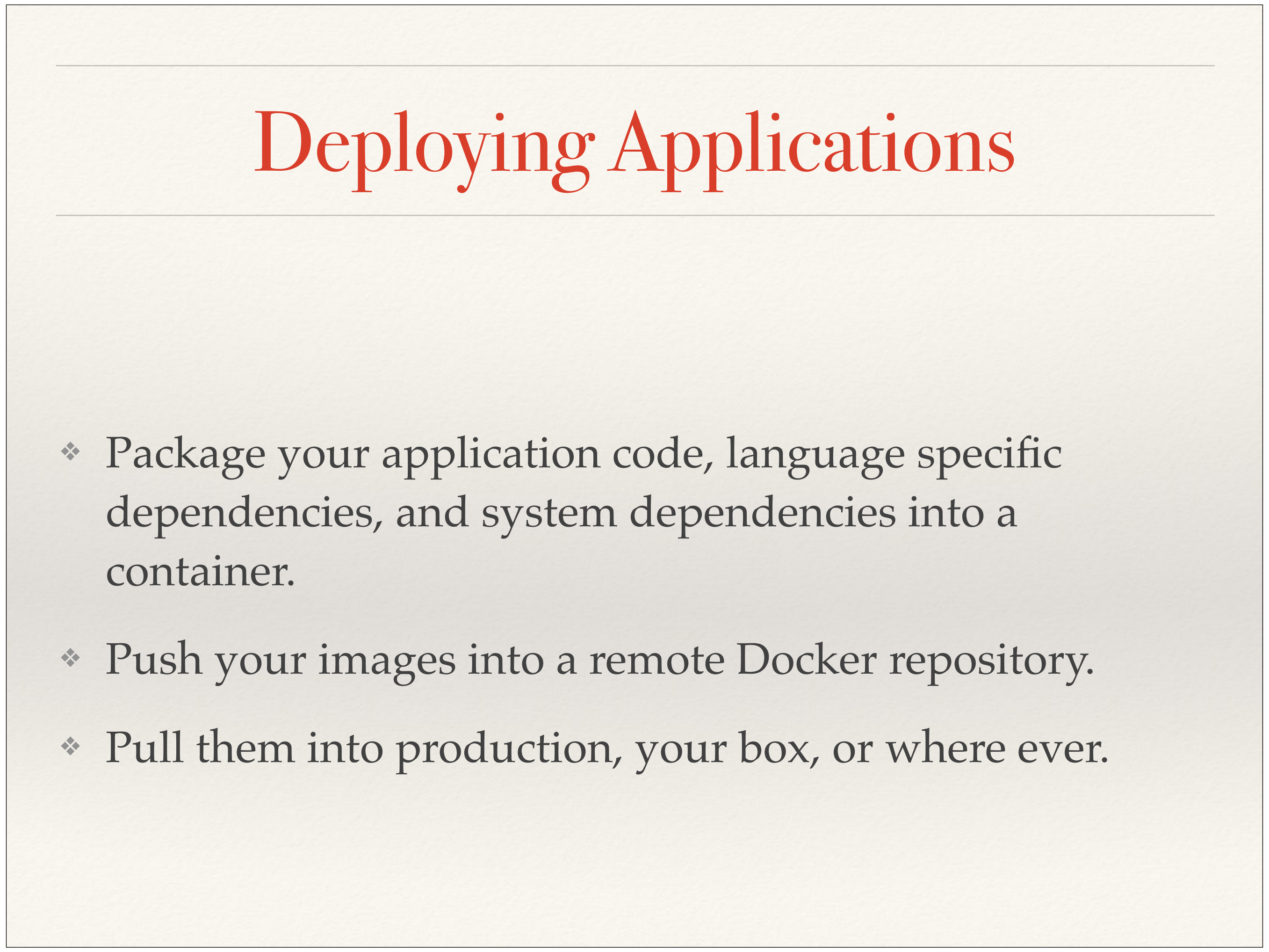
We can docker stop using what we saw from docker ps:
- container ID
- container name

Sends a terminate signal to the command you run
Will send a kill signal to the command if it doesn’t respond within 10 seconds (configurable).



What can we use containers for?

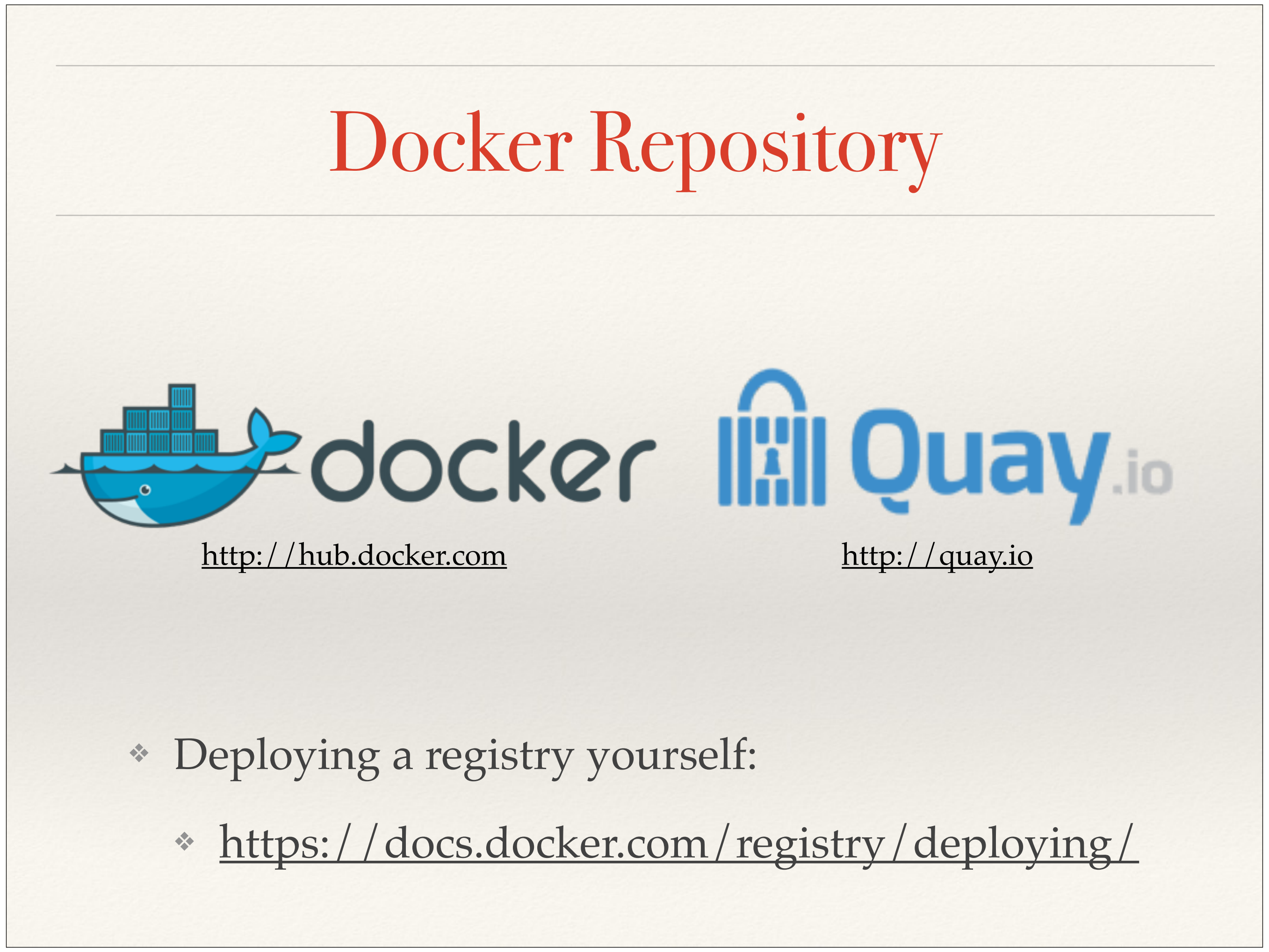
We’ve looked at how to build a simple container.
Let’s be a little creative.
Let’s talk about what we can do with containers.



Deploying Applications

- Package your application code, language specific dependencies, and system dependencies into a container.
- Push your images into a remote Docker repository.
- Pull them into production, your box, or where ever.

Deploying applications with docker is the first use case everyone considers when using Docker.
The easiest way to share our Docker images is by pushing to a remote Docker and pulling it to your production box, developer station, or whatever you like.



Docker Repository

http://hub.docker.com

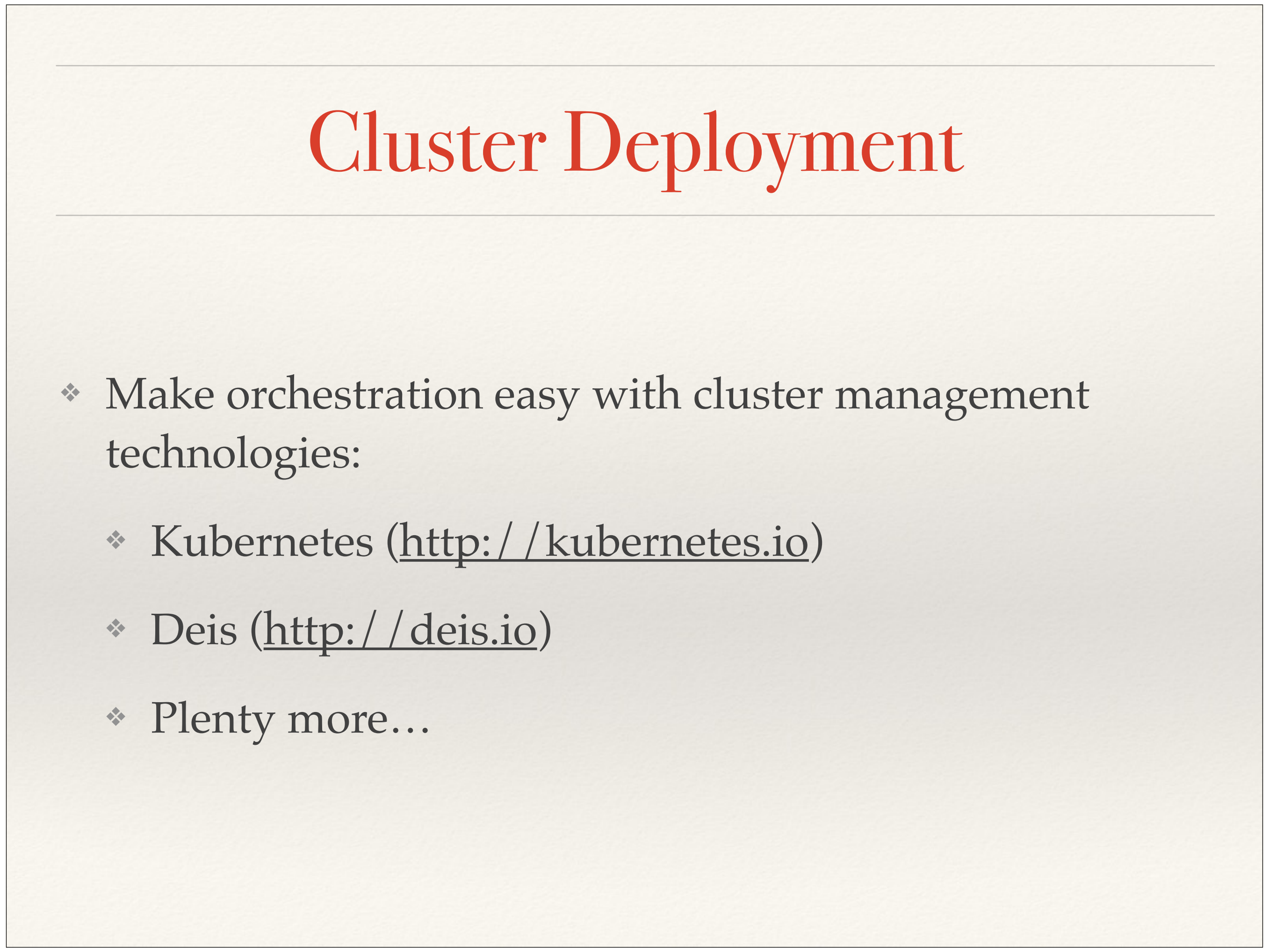
http://quay.io

- Deploying a registry yourself:
  - https://docs.docker.com/registry/deploying/

There are plenty of registry hosting services to try out!
DockerHub is Docker’s official repository hosting.
- Host unlimited public containers
- Host one private container for free to try it out!

Quay.io is CoreOS’s hosting service
- Host unlimited public containers
- Great support for teams!

Both hosts provide cool features like automatic Dockerfile building on a repository push.



Cluster Deployment

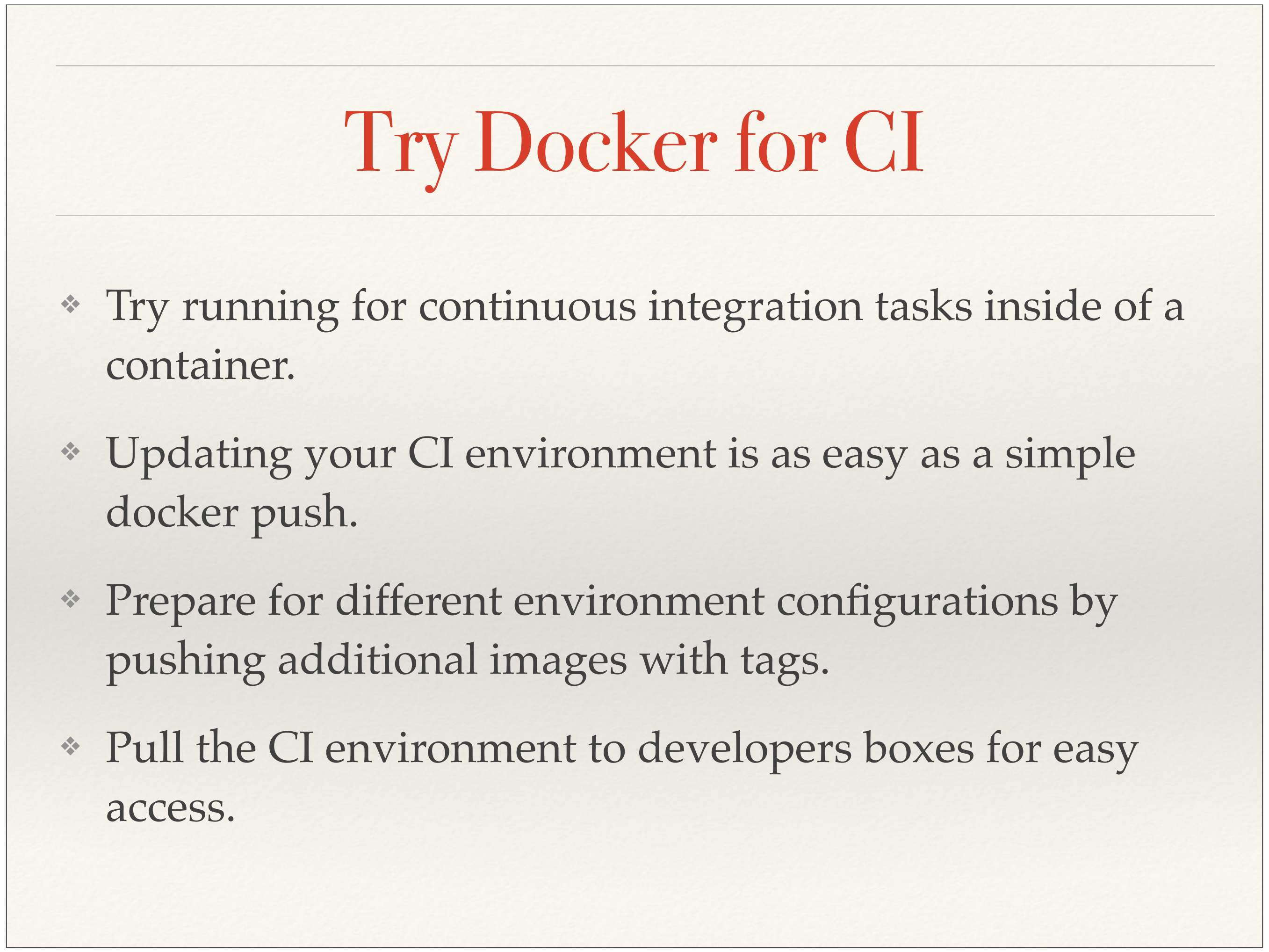
- Make orchestration easy with cluster management technologies:
  - Kubernetes (http://kubernetes.io)
  - Deis (http://deis.io)
  - Plenty more…

You might find that you'd like an easier way to distribute and run your containerized processes as your container grows. There's plenty of cluster orchestration tools for any style of deployment. I'll mention two such tools now.

Kubernetes is an descendant of Google's Borg and intended to provide a process-like abstraction over a cluster of minions. It provides basic scheduling through resources or predicates, easy access to running processes through kubectl, and other cool features.

Deis is a Heroku-like platform that intends to provide simple PaaS on top of other clustering systems. They currently support deployment through Fleet and have plans to support deployment on Kubernetes in the future.

Plenty more to choose from with what seems like a new one every day. We as an industry are still exploring the space, and it's exciting to see what people continue to come up with.



Try Docker for CI

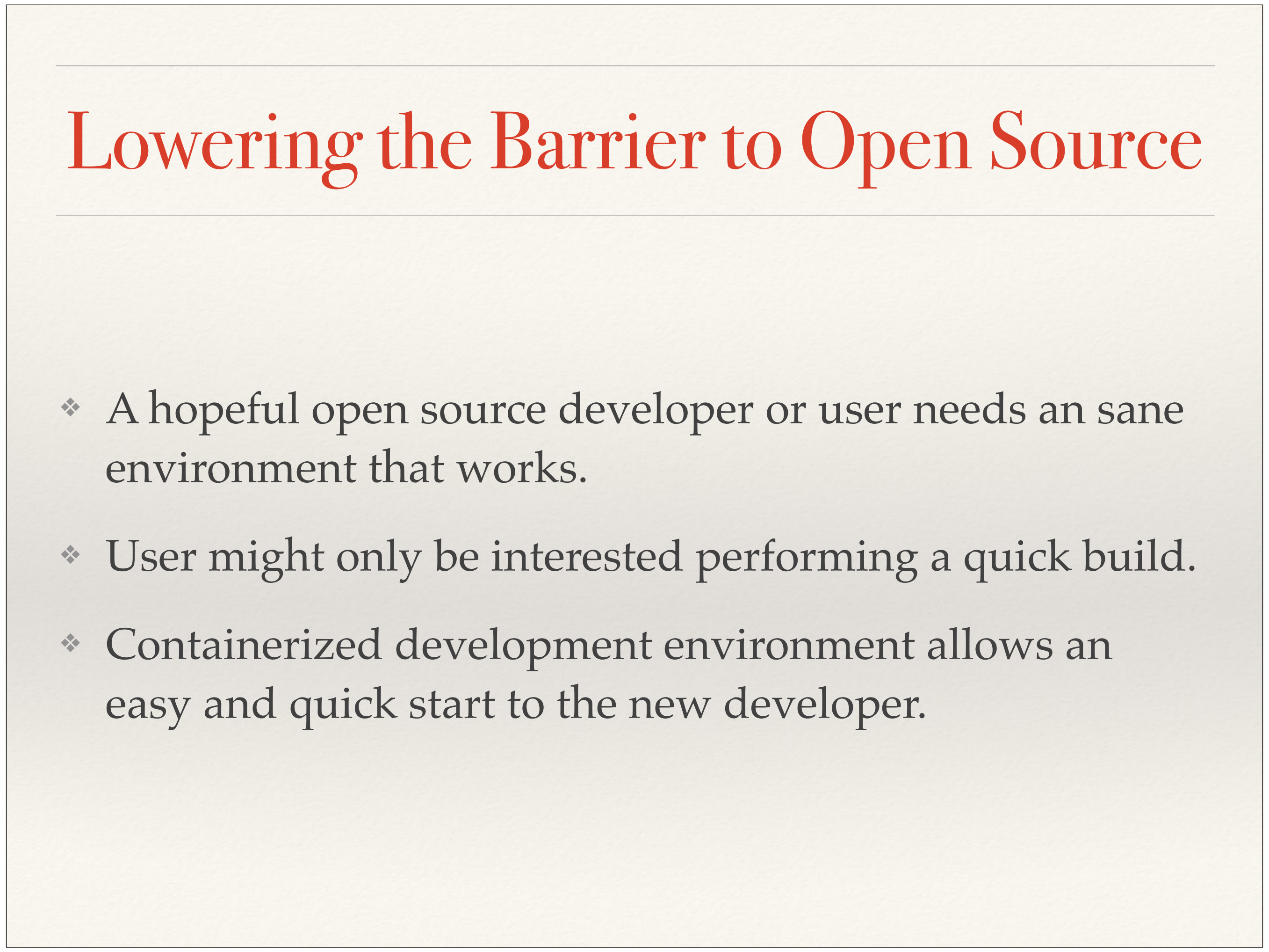
- Try running for continuous integration tasks inside of a container.
- Updating your CI environment is as easy as a simple docker push.
- Prepare for different environment configurations by pushing additional images with tags.
- Pull the CI environment to developers boxes for easy access.

We're hopefully all using continuous integration systems in our infrastructures, but how are you managing your build and testing environment? Install your tools on your local box? How do you update?

Try a Docker image for your build and test environments.
- Easily update your environment by updating your image and push to your repository.
- Try different build configurations without a complex setup.
    - Support multiple versions of languages
    - Try building with newer versions of dependencies
    - Throw it away when you're finished.

Sharing With The World

- A few additional ideas to consider trying for your continued Docker education.
- Working with containers and Docker is best way to understand it.



Lowering the Barrier to Open Source

- A hopeful open source developer or user needs an sane environment that works.
- User might only be interested performing a quick build.
- Containerized development environment allows an easy and quick start to the new developer.

The kube-register project provides a one-liner in their README page to compile a Go binary in a containerized build environment.

Docker project has an official development environment that can be setup with the Dockerfile in the root of their repository.

Consider sending a pull request to containerize an open source application for easy hacking for beginners!

Sharing Dotfiles

- dotfile repositories are plentiful on Github.
- Take someone else’s configuration for a spin!
- Share your own configuration for tech cred.

Thanks for playing along!

- Kel Cecil
- Twitter: @praisechaos
- http://www.kelcecil.com