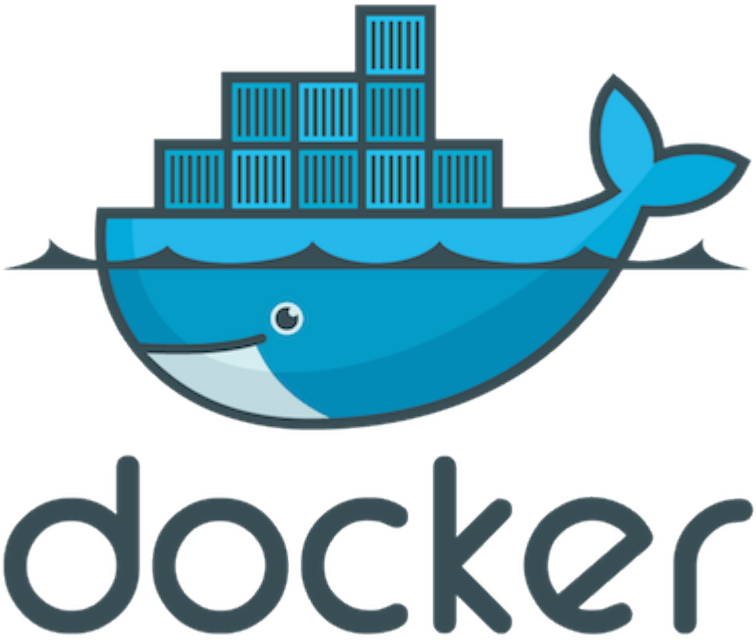
Introduction to Docker

Version: 0.0.3



An Open Platform to Build, Ship, and Run Distributed Applications

Table of Contents

Course Objective

After completing this course, you will be able to deploy Docker and Docker containers using best practices.

Course Overview

You will:

- Learn what Docker is and what it is for.
- Learn the terminology, and define images, containers, etc.
- Learn how to install Docker.
- Use Docker to run containers.
- Use Docker to build containers.
- Learn how to orchestrate Docker containers.
- Interact with the Docker Hub website.
- Acquire tips and good practice.
- Know what's next: the future of Docker, and how to get help.

Course Agenda

Part 1

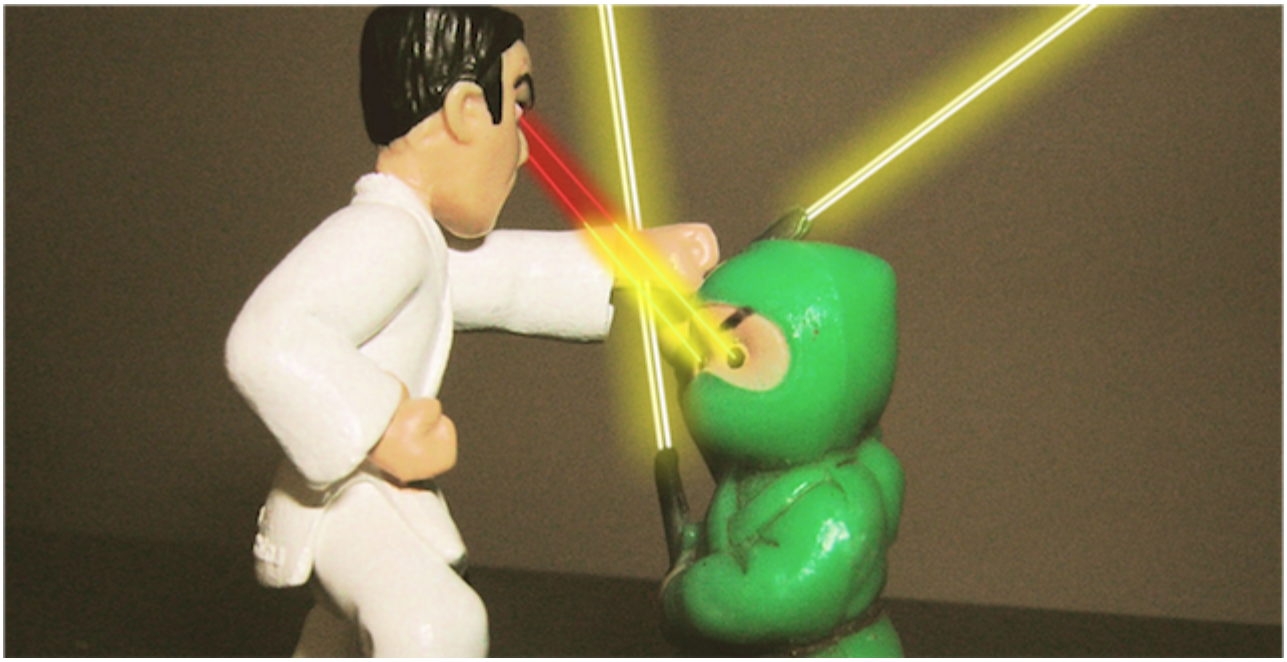
- About Docker
- Instant Gratification
- Install Docker
- Initial Images
- Our First Container
- Building on a Container
- Daemonizing Containers
- More Container Daemonization

Course Agenda

Part 2

- Local Development Workflow with Docker
- Building Images with Dockerfiles
- Working with Images and the Docker Hub website.
- Connecting Containers
- Working with Volumes
- Continuous Integration Workflow with Docker
- The Docker API

About Docker



Overview: About Docker

Objectives

At the end of this lesson, you will be able to:

- About Docker Inc.
- About Docker.
- How to create an account on the Docker Hub.

About Docker Inc.

Focused on Docker and growing the Docker ecosystem:

- Founded in 2009.
- Formerly dotCloud Inc.
- Released Docker in 2013.

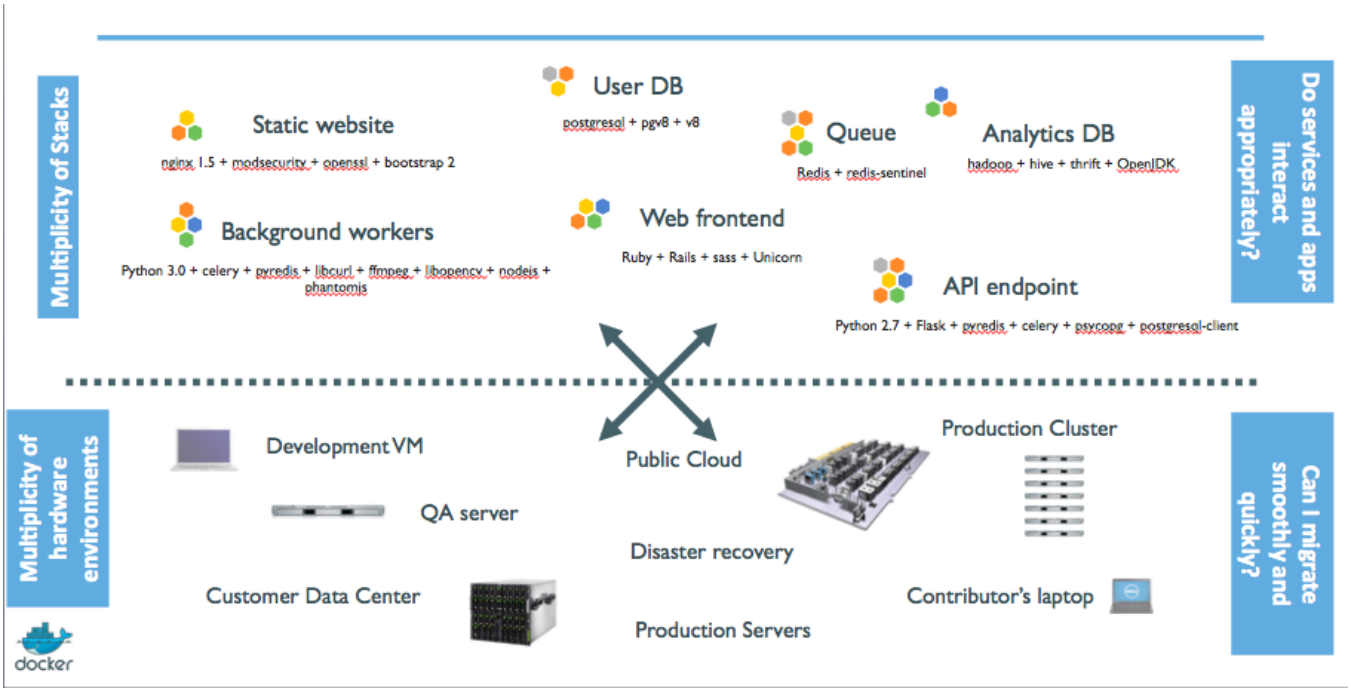
What does Docker Inc. do?

- Docker Engine - open source container management.
- Docker Hub - online home and hub for managing your Docker containers.
- Docker Enterprise Support - commercial support for Docker.
- Docker Services & Training - professional services and training to help you get the best out of Docker.

Introducing Docker

- The Matrix from Hell and the Need for Containers
- Docker & Containers: how they work

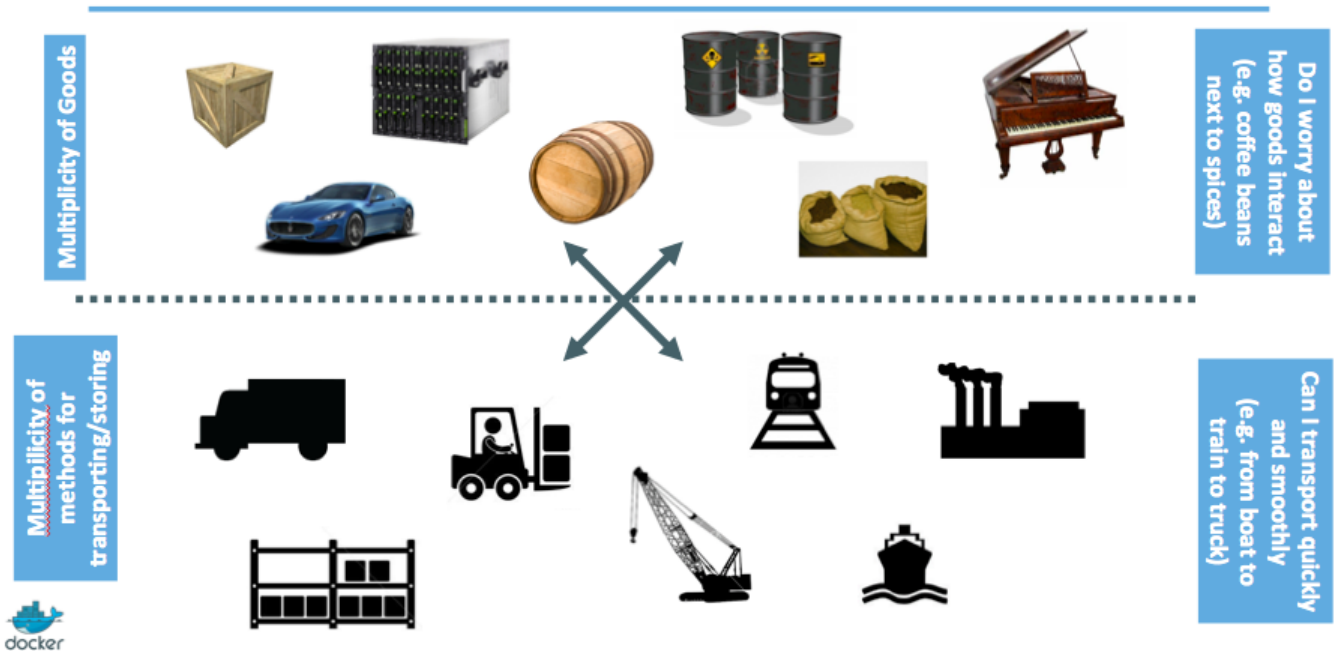
The problem in 2014



The Matrix from Hell

	Development VM	QA Server	Single Prod Server	Onsite Cluster	Public Cloud	Contributor's laptop	Customer Servers
Static website	?	?	?	?	?	?	?
Web frontend	?	?	?	?	?	?	?
Background workers	?	?	?	?	?	?	?
User DB	?	?	?	?	?	?	?
Analytics DB	?	?	?	?	?	?	?
Queue	?	?	?	?	?	?	?

An inspiration and some ancient history!



Intermodal shipping containers

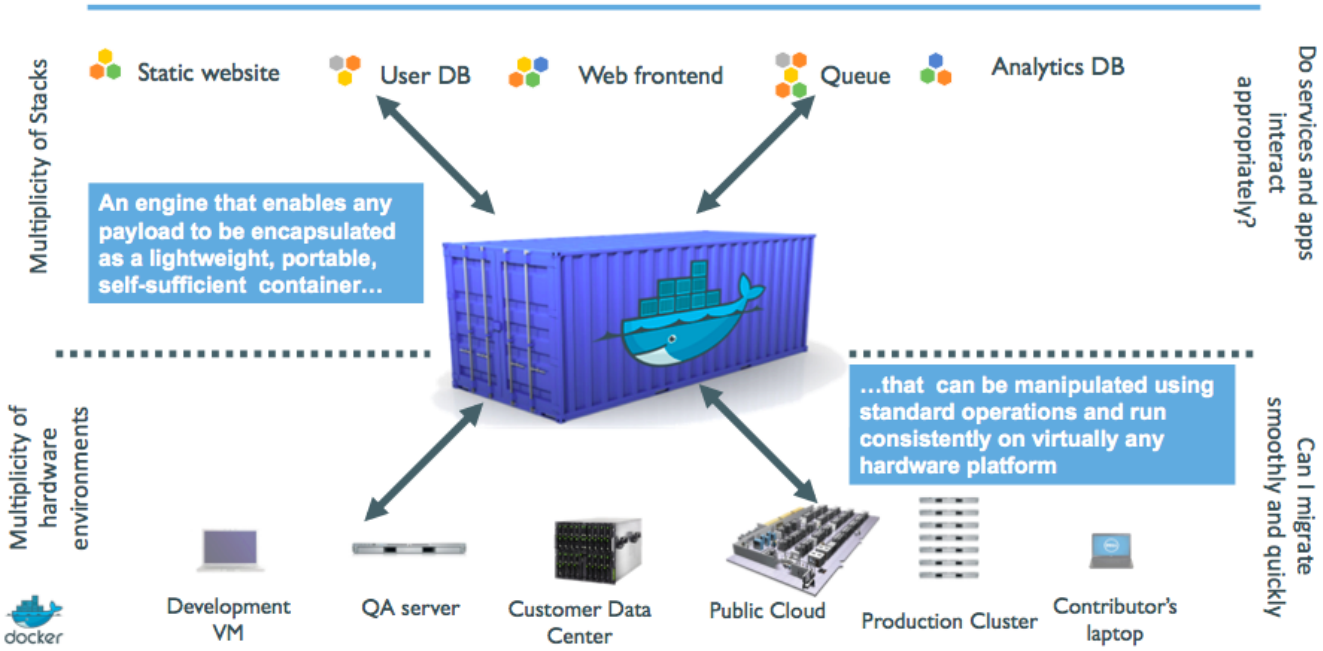
This spawned a Shipping Container Ecosystem!

- 90% of all cargo now shipped in a standard container
- Order of magnitude reduction in cost and time to load and unload ships
- Massive reduction in losses due to theft or damage
- Huge reduction in freight cost as percent of final goods (from >25% to <3%)

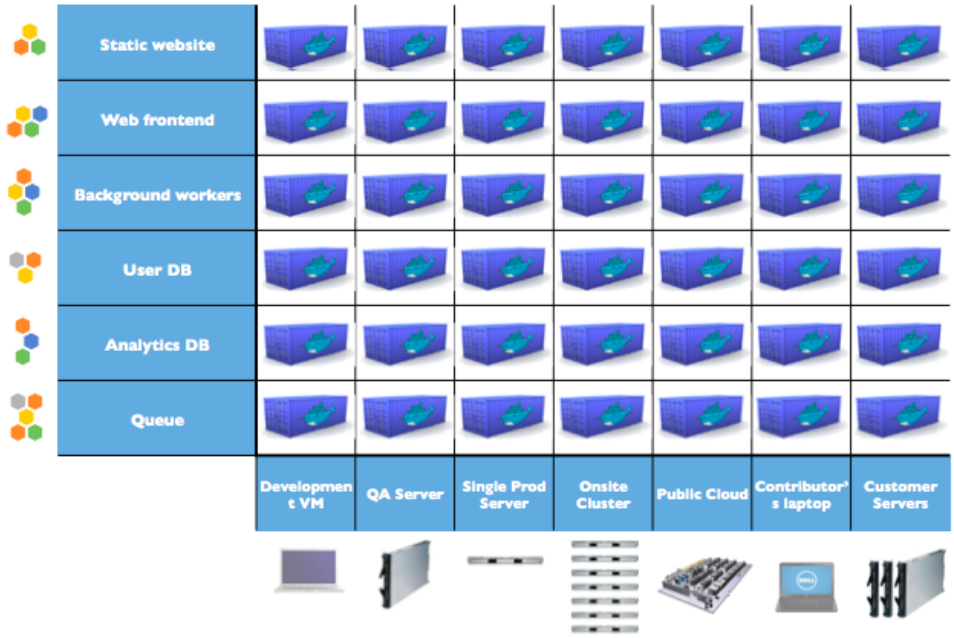
→ massive globalization

- 5000 ships deliver 200M containers per year

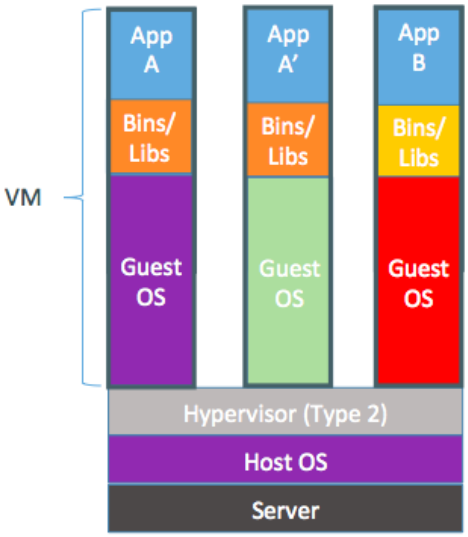
A shipping container system for applications



Eliminate the matrix from Hell

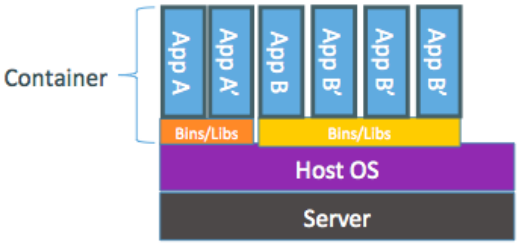


Step 1: Create a lightweight container



Containers are isolated, but share OS kernel and, where appropriate, bins/libraries

...result is significantly faster deployment, much less overhead, easier migration, faster restart

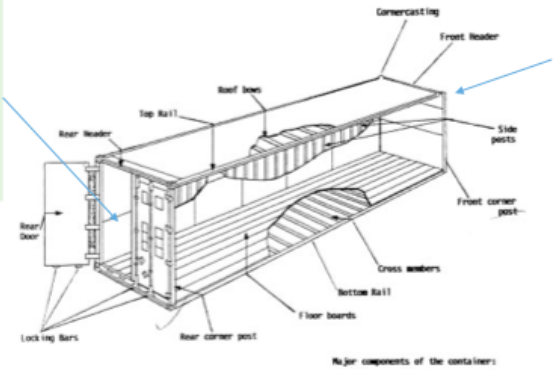


Step 2: Make containers easy to use

- Container technology has been around for a while (c.f. LXC, Solaris Zones, BSD Jails)
- But, their use was largely limited to specialized organizations, with special tools & training. Containers were not portable
- Analogy: Shipping containers are not just steel boxes. They are steel boxes that are a standard size, and have hooks and holes in all the same places
- With Docker, Containers get the following:
    - Ease of use, tooling
    - Re-usable components
    - Ability to run on any Linux server today: physical, virtual, VM, cloud, OpenStack, +++
    - Ability to move between any of the above in a matter of seconds-no modification or delay
    - Ability to share containerized components
    - Self contained environment - no dependency hell
    - Tools for how containers work together: linking, nesting, discovery, orchestration, ++

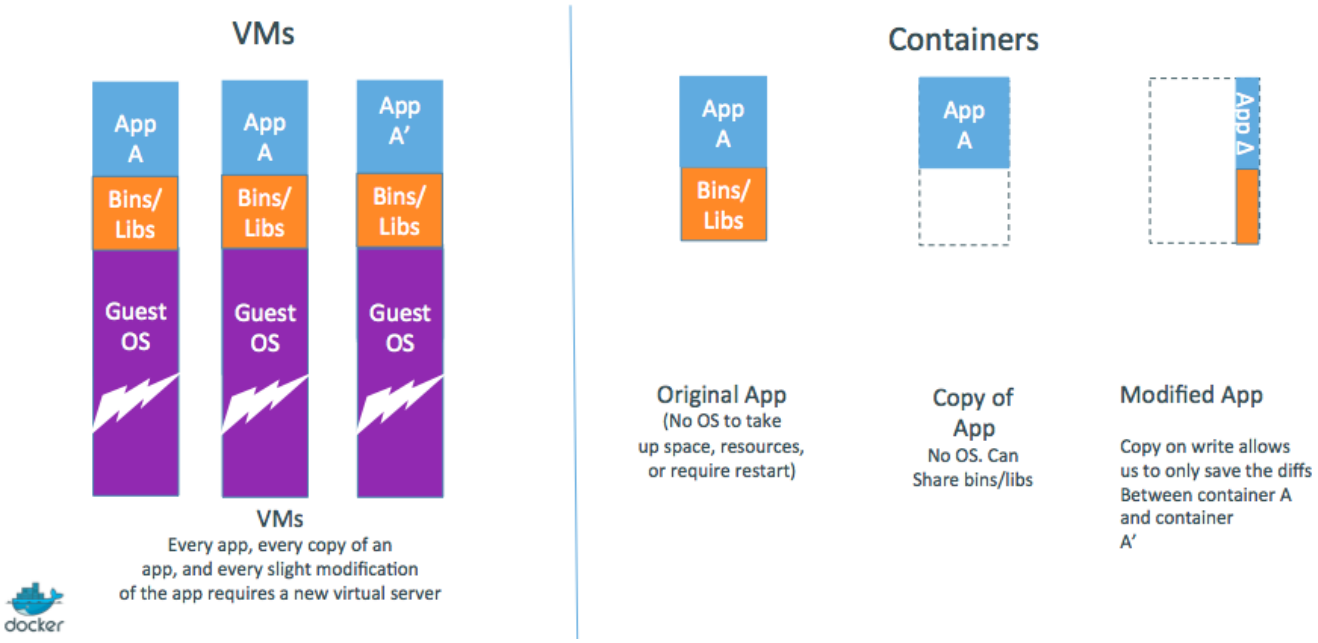
Technical & cultural revolution: separation of concerns

- Dan the Developer
  - Worries about what's "inside" the container
    - His code
    - His Libraries
    - His Package Manager
    - His Apps
    - His Data
  - All Linux servers look the same

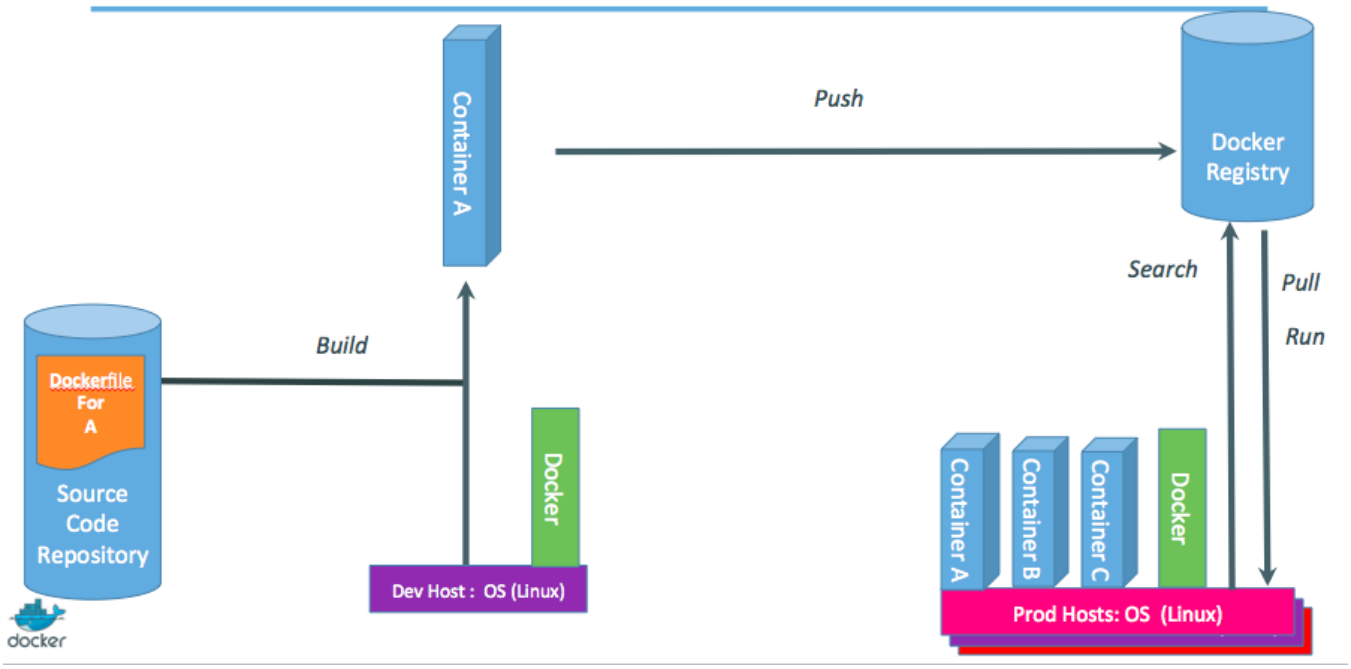


- Oscar the Ops Guy
  - Worries about what's "outside" the container
    - Logging
    - Remote access
    - Monitoring
    - Network config
  - All containers start, stop, copy, attach, migrate, etc. the same way

Step 3: Make containers super lightweight



Step 4: Build a System for creating, managing, deploying code



Instant Gratification

Overview: Instant Gratification

Objectives

At the end of this lesson, you will have:

- Seen Docker in action

Instant Gratification

- Follow the bouncing ball
- We'll all get a chance to do this shortly

Hello World

```
$ sudo docker run busybox echo hello world
hello world
```


That was our first container!

- Busybox (small, lean image)
- Ran a single process and echo'ed `hello world`

Download the ubuntu image

- The Busybox image doesn't do much.
- We will use a beefier Ubuntu image later.
- It will take a few minutes to download, so let's do that now.

```
$ sudo docker pull ubuntu:latest
Pulling repository ubuntu
9f676bd305a4: Download complete
9cd978db300e: Download complete
bac448df371d: Downloading
[============>          ] 10.04 MB/39.93 MB 23s
e7d62a8128cf: Downloading
[======>              ] 8.982 MB/68.32 MB
1m21s
f323cf34fd77: Download complete
321f7f4200f4: Download complete
```

(Let the download run in the background while we move on!)

Install Docker

Lesson 3: Installing Docker

Objectives

At the end of this lesson, you will be able to:

- Install Docker.
- Run Docker without `sudo`.

Note: if you were provided with a training VM for a hands-on tutorial, do not run the commands in this lesson! That VM already has Docker installed, and Docker has already been setup to run without `sudo`.

Installing Docker

Docker is easy to install.

It runs on:

- A variety of Linux distributions.
- OS X via a virtual machine.
- Microsoft Windows via a virtual machine.

Installing Docker on Linux

It can be installed via:

- Distribution supplied packages on virtually all distros.
  (Includes at least: Arch Linux, CentOS, Debian, Fedora, Gentoo, openSUSE, RHEL, Ubuntu.)
- Packages supplied by Docker.
- Installation script from Docker.
- Binary download from Docker (it's a single file).

Installing Docker on your Linux distribution

On Red Hat and derivatives.

```
$ sudo yum install docker
```

On Debian and derivatives.

```
$ sudo apt-get install docker.io
```


Installation script from Docker

You can use the `curl` command to install on several platforms.

```
$ curl -s https://get.docker.io/ubuntu/ | sudo sh
```

This currently works on:

- Ubuntu;
- Debian;
- Fedora;
- Gentoo.

Installing on OS X and Microsoft Windows

Docker doesn't run natively on OS X and Microsoft Windows.

To install Docker on these platforms we run a small virtual machine using a tool called Boot2Docker.

Docker architecture

- Docker is a client-server application.

The Docker daemon

- The Docker server.
- Receives and processes incoming Docker API requests.

The Docker client

- Command line tool - the `docker` binary.
- Talks to the Docker daemon via the Docker API.

Docker Hub Registry

- Public image registry.
- The Docker daemon talks to it via the registry API.

Test Docker is working

Using the `docker` client:

```
$ sudo docker version
Client version: 1.1.1
Client API version: 1.13
Go version (client): go1.2.1
Git commit (client): bd609d2
Server version: 1.1.1
Server API version: 1.13
Go version (server): go1.2.1
Git commit (server): bd609d2
```


Su-su-sudo

The `docker` group

Warning!

The `docker` user is `root` equivalent.

It provides `root` level access to the host.

You should restrict access to it like you would protect `root`.

Add the Docker group

```
$ sudo groupadd docker
```

Add ourselves to the group

```
$ sudo gpasswd -a $USER docker
```

Restart the Docker daemon

```
$ sudo service docker restart
```

Log out

```
$ exit
```


Hello World again without sudo

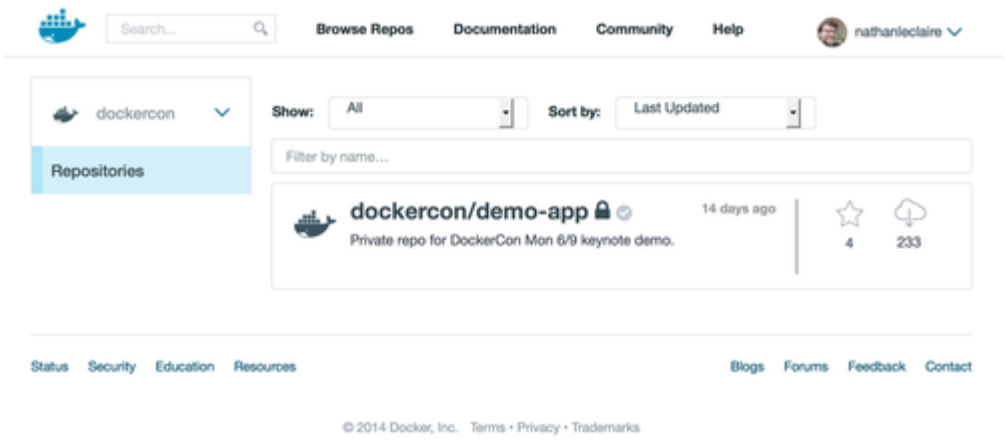
```
$ docker run ubuntu echo hello world
```


Section summary

We've learned how to:

- Install Docker.
- Run Docker without `sudo`.

Introducing Docker Hub



Lesson 4: Introducing Docker Hub

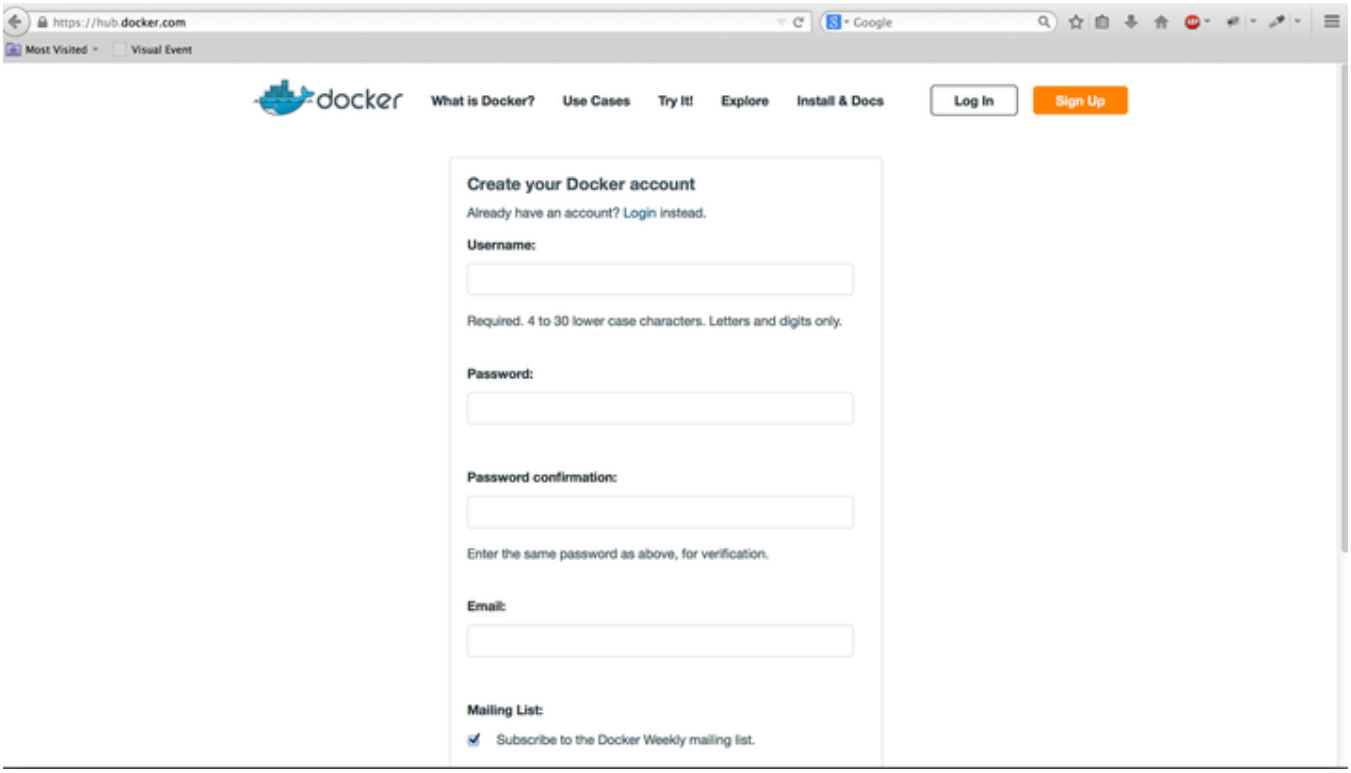
Objectives

At the end of this lesson, you will be able to:

- Register for an account on Docker Hub.
- Login to your account from the command line.

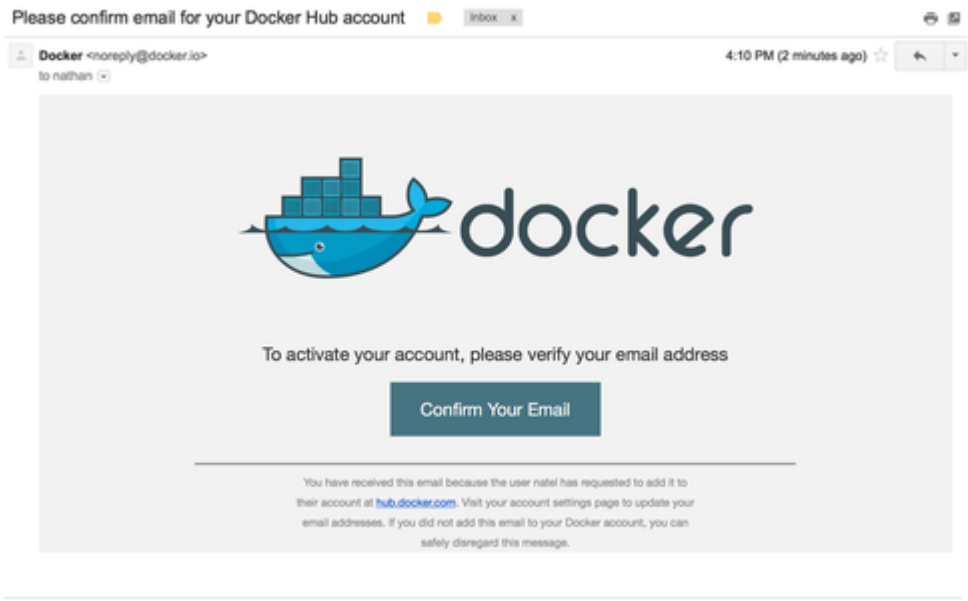
Sign up for a Docker Hub account

- Having a Docker Hub account will allow us to store our images in the registry.
- To sign up, you'll go to hub.docker.com and fill out the form.
- Note: if you have an existing Index/Hub account, this step is not needed.



Activate your account through e-mail.

- Check your e-mail and click the confirmation link.



Login

Let's use our new account to login to the Docker Hub!

```
$ docker login
Username: my_docker_hub_login
Password:
Email: my@email.com
Login Succeeded
```

Our credentials will be stored in `~/.dockercfg`.

The `.dockercfg` configuration file

The `~/.dockercfg` configuration file holds our Docker registry authentication credentials.

```
{
  "https://index.docker.io/v1/": {
    "auth":"amFtdHVyMDE6aTliMUw5ckE=",
    "email":"education@docker.com"
  }
}
```

The `auth` section is Base64 encoding of your user name and password.

It should be owned by your user with permissions of `0600`.

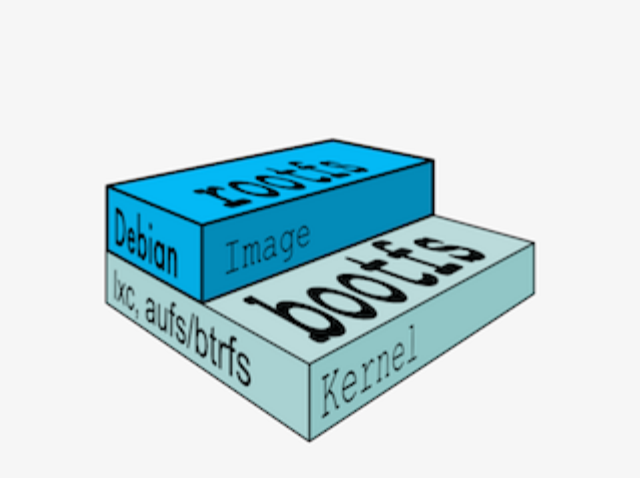
You should protect this file!

Section summary

We've learned how to:

- Register for an account on Docker Hub.
- Login to your account from the command line.

Getting started with Images



Lesson 5: Getting started with Images

Objectives

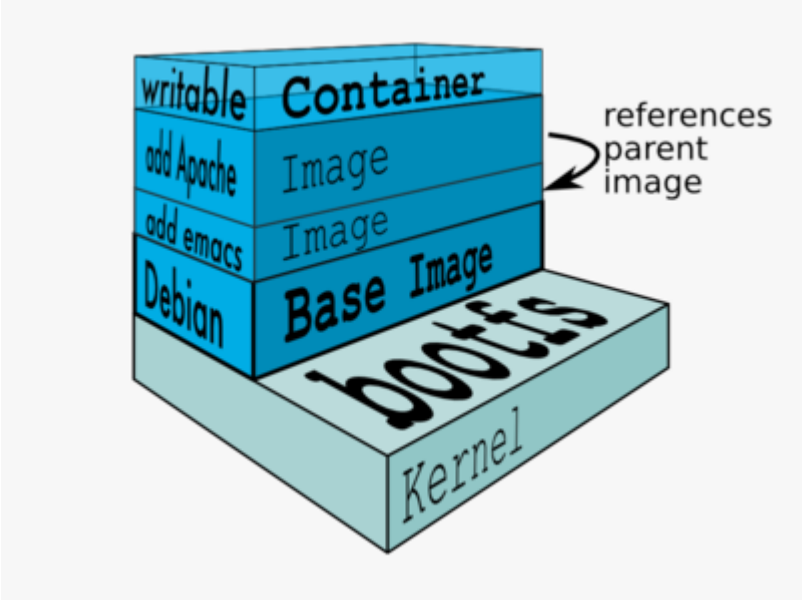
At the end of this lesson, you will be able to:

- Understand images and image tags.
- Search for images.
- Download an image.
- Understand Docker image namespacing.

Images

What are they?

- An image is a collection of files.
- *Base images* (ubuntu, busybox, fedora etc.) are what you build your own custom images on top of.
- Images are *layered*, and each layer represents a diff (what changed) from the previous layer. For instance, you could add Python 3 on top of a base image.



So what's the difference between Containers and Images?

- Containers represent an encapsulated set of processes based on an image.
- You spawn them with the `docker run` command.
- In our "Instant Gratification" example, you created a shiny new container by executing `docker run`. It was based on the `busybox` image, and we ran the `echo` command.
- Images are like templates or stencils that you can create containers from.

How do you store and manage images?

Images can be stored:

- On your Docker host.
- In a Docker registry.

You can use the Docker client to manage images.

Searching for images

Searches your registry for images:

```
$ docker search training
NAME                                   DESCRIPTION   STARS     OFFICIAL   AUTOMATED
training/jenkins                                     0                    [OK]
training/webapp                                      0                    [OK]
training/ls                                          0                    [OK]
training/namer                                       0                    [OK]
training/postgres                                    0                    [OK]
training/notes                                       0                    [OK]
training/docker-fundamentals-image                   0                    [OK]
training/showoff                                     0
```


Images belong to a namespace

There are three namespaces:

- Root-like

```
ubuntu
```

- User

```
training/docker-fundamentals-image
```

- Self-Hosted

```
registry.example.com:5000/my-private-image
```


Root namespace

The root namespace is for official images. They are put there by Docker Inc., but they are generally authored and maintained by third parties.

Those images include some barebones distro images, for instance:

- ubuntu
- fedora
- centos

Those are ready to be used as bases for your own images.

User namespace

The user namespace holds images for Docker Hub users and organizations.

For example:

```
training/docker-fundamentals-image
```

The Docker Hub user is:

```
training
```

The image name is:

```
docker-fundamentals-image
```


Self-Hosted namespace

This namespace holds images which are not hosted on Docker Hub, but on third party registries.

They contain the hostname (or IP address), and optionally the port, of the registry server.

For example:

```
localhost:5000/wordpress
```

The remote host and port is:

```
localhost:5000
```

The image name is:

```
wordpress
```

Note: self-hosted registries used to be called *private* registries, but this is misleading!

- A self-hosted registry can be public or private.
- A registry in the User namespace on Docker Hub can be public or private.

Downloading images

We already downloaded two root images earlier:

- The busybox image, implicitly, when we did `docker run busybox`.
- The ubuntu image, explicitly, when we did `docker pull ubuntu`.

Download a user image.

```
$ docker pull training/docker-fundamentals-image
Pulling repository training/docker-fundamentals-image
8144a5b2bc0c: Download complete
511136ea3c5a: Download complete
8abc22fbb042: Download complete
58394af37342: Download complete
6ea7713376aa: Download complete
71ef82f6ed3c: Download complete
```


Showing current images

Let's look at what images are on our host now.

```
$ docker images
REPOSITORY                          TAG     IMAGE ID     CREATED     VIRTUAL SIZE
training/docker-fundamentals-image latest  8144a5b2bc0c 5 days ago  835 MB
ubuntu                              13.10   9f676bd305a4 7 weeks ago 178 MB
ubuntu                              saucy   9f676bd305a4 7 weeks ago 178 MB
ubuntu                              raring  eb601b8965b8 7 weeks ago 166.5 MB
ubuntu                              13.04   eb601b8965b8 7 weeks ago 166.5 MB
ubuntu                              12.10   5ac751e8d623 7 weeks ago 161 MB
ubuntu                              quantal 5ac751e8d623 7 weeks ago 161 MB
ubuntu                              10.04   9cc9ea5ea540 7 weeks ago 180.8 MB
ubuntu                              lucid   9cc9ea5ea540 7 weeks ago 180.8 MB
ubuntu                              12.04   9cd978db300e 7 weeks ago 204.4 MB
ubuntu                              latest  9cd978db300e 7 weeks ago 204.4 MB
ubuntu                              precise 9cd978db300e 7 weeks ago 204.4 MB
```


Image and tags

- Images can have tags.
- Tags define image variants.
- When using images it is always best to be specific.

Downloading an image tag

```
$ docker pull debian:jessie
Pulling repository debian
b164861940b8: Download complete
b164861940b8: Pulling image (jessie) from debian
d1881793a057: Download complete
```

- As seen previously, images are made up of layers.
- Docker has downloaded all the necessary layers.

Section summary

We've learned how to:

- Understand images and image tags.
- Search for images.
- Download an image.
- Understand Docker image namespacing.

Building Docker images

Lesson 6: Building Docker Images

Objectives

At the end of this lesson, you will be able to:

- Understand the instructions for a `Dockerfile`.
- Create your own `Dockerfile`s.
- Build an image from a `Dockerfile`.
- Override the `CMD` when you `docker run`.

Introduction to building images

`Dockerfile` and `docker build`

Let's see how to build our own images using:

- A `Dockerfile` which holds Docker image definitions.
  You can think of it as the "build recipe" for a Docker image.
  It contains a series of instructions telling Docker how an image is constructed.
- The `docker build` command which builds an image from a `Dockerfile`.

Our first `Dockerfile`

```
# You can download that file from:
# https://github.com/docker-training/staticweb
FROM ubuntu:14.04
MAINTAINER Docker Education Team <education@docker.com>

RUN apt-get update
RUN apt-get install -y nginx
RUN echo 'Hi, I am in your container' \
    >/usr/share/nginx/html/index.html

CMD [ "nginx", "-g", "daemon off;" ]

EXPOSE 80
```

- `FROM` specifies a source image for our new image. It's mandatory.
- `MAINTAINER` tells us who maintains this image.
- Each `RUN` instruction executes a command to build our image.
- `CMD` defines the default command to run when a container is launched from this image.
- `EXPOSE` lists the network ports to open when a container is launched from this image.

`Dockerfile` usage summary

- `Dockerfile` instructions are executed in order.
- Each instruction creates a new layer in the image.
- Instructions are cached. If no changes are detected then the instruction is skipped and the cached layer used.
- The `FROM` instruction MUST be the first non-comment instruction.
- Lines starting with `#` are treated as comments.
- You can only have one `CMD` and one `ENTRYPOINT` instruction in a `Dockerfile`.

Building our `Dockerfile`

We use the `docker build` command to build images.

```
$ docker build -t web .
```

- The `-t` flag tags an image.
- The `.` indicates the location of the `Dockerfile` being built.

We can also build from other sources.

```
$ docker build -t web https://github.com/docker-training/staticweb.git
```

Here we've specified a GitHub repository to build.

The `FROM` instruction

- Specifies the source image to build this image.
- Must be the first instruction in the `Dockerfile`.
  (Except for comments: it's OK to have comments before `FROM`.)

Can specify a base image:

```
FROM ubuntu
```

An image tagged with a specific version:

```
FROM ubuntu:12.04
```

A user image:

```
FROM training/sinatra
```

Or self-hosted image:

```
FROM localhost:5000/funtoo
```


More about `FROM`

- The `FROM` instruction can be specified more than once to build multiple images.

```
FROM ubuntu:14.04
. . .
FROM fedora:20
. . .
```

  Each FROM instruction marks the beginning of the build of a new image.
  The `-t command-line parameter will only apply to the last image.

- If the build fails, existing tags are left unchanged.

- An optional version tag can be added after the name of the image.
  E.g.: `ubuntu:14.04`.

The `MAINTAINER` instruction

The `MAINTAINER` instruction tells you who wrote the `Dockerfile`.

```
MAINTAINER Docker Education Team <education@docker.com>
```

It's optional but recommended.

The `RUN` instruction

The `RUN` instruction can be specified in two ways.

With shell wrapping, which runs the specified command inside a shell, with `/bin/sh -c`:

```
RUN apt-get update
```

Or using the `exec` method, which avoids shell string expansion, and allows execution in images that don't have `/bin/sh`:

```
RUN [ "apt-get", "update" ]
```


More about the `RUN` instruction

`RUN` will do the following:

- Execute a command.
- Record changes made to the filesystem.
- Work great to install libraries, packages, and various files.

`RUN` will NOT do the following:

- Record state of *processes*.
- Automatically start daemons.

If you want to start something automatically when the container runs, you should use `CMD` and/or `ENTRYPOINT`.

The EXPOSE instruction

The EXPOSE instruction tells Docker what ports are to be published in this image.

```
EXPOSE 8080
```

- All ports are private by default.
- The Dockerfile doesn't control if a port is publicly available.
- When you docker run -p <port> ..., that port becomes public. (Even if it was not declared with EXPOSE.)
- When you docker run -P ... (without port number), all ports declared with EXPOSE become public.

A *public port* is reachable from other containers and from outside the host.

A *private port* is not reachable from outside.

The `ADD` instruction

The `ADD` instruction adds files and content from your host into the image.

```
ADD /src/webapp /opt/webapp
```

This will add the contents of the `/src/webapp/` directory to the `/opt/webapp` directory in the image.

Note: `/src/webapp/` is not relative to the host filesystem, but to the directory containing the `Dockerfile`.

Otherwise, a `Dockerfile` could succeed on host A, but fail on host B.

The `ADD` instruction can also be used to get remote files.

```
ADD http://www.example.com/webapp /opt/
```

This would download the `webapp` file and place it in the `/opt` directory.

More about the `ADD` instruction

- `ADD` is cached. If you recreate the image and no files have changed then a cache is used.
- If the local source is a zip file or a tarball it'll be unpacked to the destination.
- Sources that are URLs and zipped will not be unpacked.
- Any files created by the `ADD` instruction are owned by `root` with permissions of `0755`.

More on `ADD` here.

The `VOLUME` instruction

The `VOLUME` instruction will create a data volume mount point at the specified path.

```
VOLUME [ "/opt/webapp/data" ]
```

- Data volumes bypass the union file system.
  In other words, they are not captured by `docker commit`.
- Data volumes can be shared and reused between containers.
  We'll see how this works in a subsequent lesson.
- It is possible to share a volume with a stopped container.
- Data volumes persist until all containers referencing them are destroyed.

The `WORKDIR` instruction

The `WORKDIR` instruction sets the working directory for subsequent instructions.

It also affects `CMD` and `ENTRYPOINT`, since it sets the working directory used when starting the container.

```
WORKDIR /opt/webapp
```

You can specify `WORKDIR` again to change the working directory for further operations.

The `ENV` instruction

The `ENV` instruction specifies environment variables that should be set in any container launched from the image.

```
ENV WEBAPP_PORT 8080
```

This will result in an environment variable being created in any containers created from this image of

```
WEBAPP_PORT=8080
```

You can also specify environment variables when you use `docker run`.

```
$ docker run -e WEBAPP_PORT=8000 -e WEBAPP_HOST=www.example.com ...
```


The `USER` instruction

The `USER` instruction sets the user name or UID to use when running the image.

It can be used multiple times to change back to root or to another user.

The `CMD` instruction

The `CMD` instruction is a default command run when a container is launched from the image.

```
CMD [ "nginx", "-g", "daemon off;" ]
```

Means we don't need to specify `nginx -g "daemon off;"` when running the container.

Instead of:

```
$ docker run web nginx -g "daemon off;"
```

We can just do:

```
$ docker run web
```


More about the `CMD` instruction

Just like `RUN`, the `CMD` instruction comes in two forms. The first executes in a shell:

```
CMD nginx -g "daemon off;"
```

The second executes directly, without shell processing:

```
CMD [ "nginx", "-g", "daemon off;" ]
```


Overriding the `CMD` instruction

The `CMD` can be overridden when you run a container.

```
$ docker run -t -i web /bin/bash
```

Will run `/bin/bash` instead of `nginx -g "daemon off;"`.

The ENTRYPOINT instruction

The ENTRYPOINT instruction is like the CMD instruction, but arguments given on the command line are *appended* to the entry point.

Note: you have to use the "exec" syntax (["..."]).

```
ENTRYPOINT [ "/bin/ls" ]
```

If we were to run:

```
$ docker run training/ls -l
```

Instead of trying to run -l, the container will run /bin/ls -l.

Overriding the `ENTRYPOINT` instruction

The entry point can be overriden as well.

```
$ docker run --entrypoint /bin/bash -t -i training/ls
```


How CMD and ENTRYPOINT interact

The CMD and ENTRYPOINT instructions work best when used together.

```
ENTRYPOINT [ "nginx" ]
CMD [ "-g", "daemon off;" ]
```

The ENTRYPOINT specifies the command to be run and the CMD specifies its options. On the command line we can then potentially override the options when needed.

```
$ docker run -d web -t
```

This will override the options CMD provided with new flags.

The ONBUILD instruction

The ONBUILD instruction is a trigger. It sets instructions that will be executed when another image is built from the image being build.

This is useful for building images which will be used as a base to build other images.

```
ONBUILD ADD . /app/src
```

- You can't chain ONBUILD instructions with ONBUILD.
- ONBUILD can't be used to trigger FROM and MAINTAINER instructions.

Summary

We've learned how to:

- Understand the instructions for a `Dockerfile`.
- Create your own `Dockerfile`s.
- Build an image from a `Dockerfile`.
- Override the `CMD` when you `docker run`.

Working with Images

Lesson 7: Working with Images

Objectives

At the end of this lesson, you will be able to:

- Pull and push images to the Docker Hub.
- Explore the Docker Hub.
- Understand and create *Automated Builds*.

Working with images

In the last section we created a new image for our web application.

This image would be useful to the whole team but how do we share it?

Using the Docker Hub!

Pulling images

Earlier in this training we saw how to pull images down from the Docker Hub.

```
$ docker pull ubuntu:14.04
```

This will connect to the Docker Hub and download the `ubuntu:14.04` image to allow us to build containers from it.

We can also do the reverse and push an image to the Docker Hub so that others can use it.

Before pushing a Docker image ...

We push images using the `docker push` command.

Images are uploaded via HTTP and authenticated.

You can only push images to the *user namespace*, and with your own username.

This means that you cannot push an image called `web`.

It has to be called `my_docker_hub_login/web`.

Name your image properly

Here are different ways to ensure that your image has the right name.

Of course, in the examples below, replace `my_docker_hub_login` with your actual login on the Docker Hub.

- If you have previously built the `web` image, you can re-tag it:

```
$ docker tag web my_docker_hub_login/web
```

- Or, you can also rebuild it from scratch:

```
$ docker build -t my_docker_hub_login/web \
  git://github.com/docker-training/
staticweb.git
```


Pushing a Docker image to the Docker Hub

Now that the image is named properly, we can push it:

```
$ docker push my_docker_hub_login/web
```

You will be prompted for a user name and password.

(Unless you already did `docker login` earlier.)

```
Please login prior to push:
Username: my_docker_hub_login
Password: *********
Email: ...
Login Succeeded
```

You will login using your Docker Hub name, account and email address you created earlier in the training.

Your authentication credentials

Docker stores your credentials in the file `~/.dockercfg`.

```
$ cat ~/.dockercfg
```

Will show you the authentication details.

```
{
  "https://index.docker.io/v1/": {
    "auth":"amFtdHVyMDE6aTliMUw5ckE=",
    "email":"education@docker.com"
  }
}
```


More about pushing an image

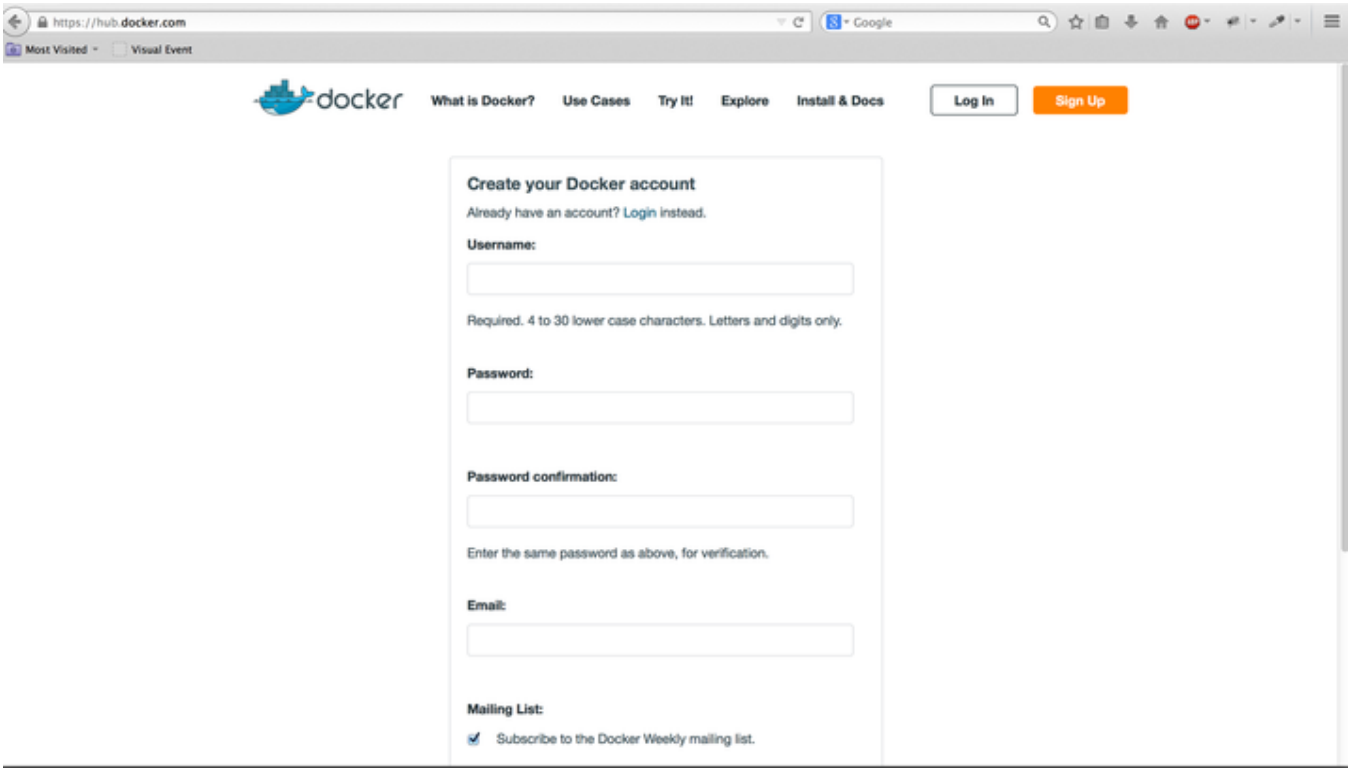
- If the image doesn't exist on the Docker Hub, a new repository will be created.
- You can push an updated image on top of an existing image. Only the layers which have changed will be updated.
- When you pull down the resulting image, only the updates will need to be downloaded.

Viewing our uploaded image

Let's sign onto the [Docker Hub](https://hub.docker.com/) and review our uploaded image.

Browse to:

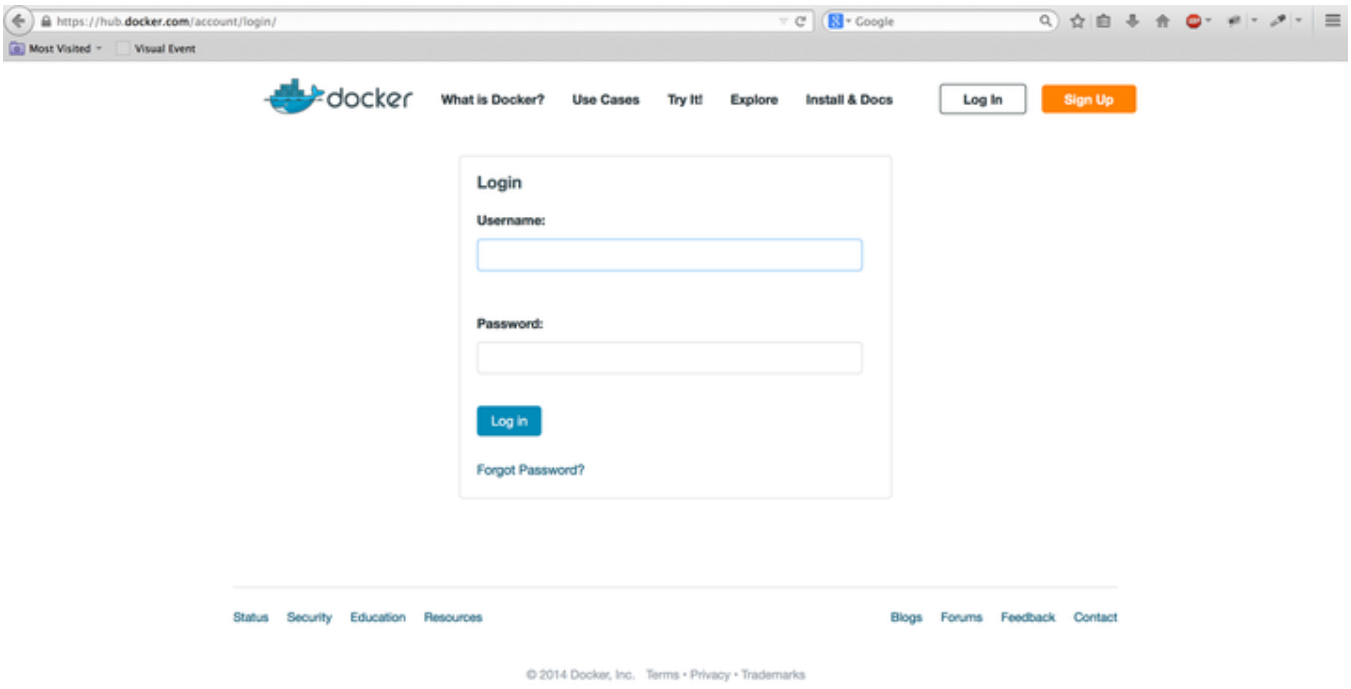
```
https://hub.docker.com/
```



Logging in to the Docker Hub

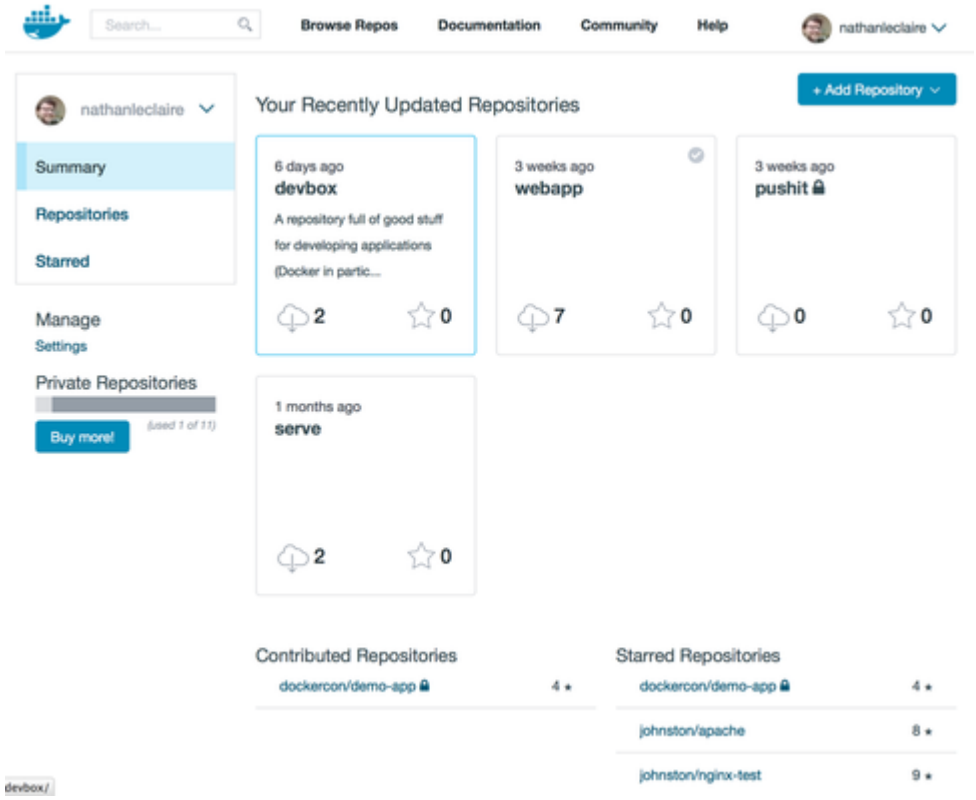
Now click the `Login` link and fill in the `Username` and `Password` fields.

And clicking the `Log In` button.



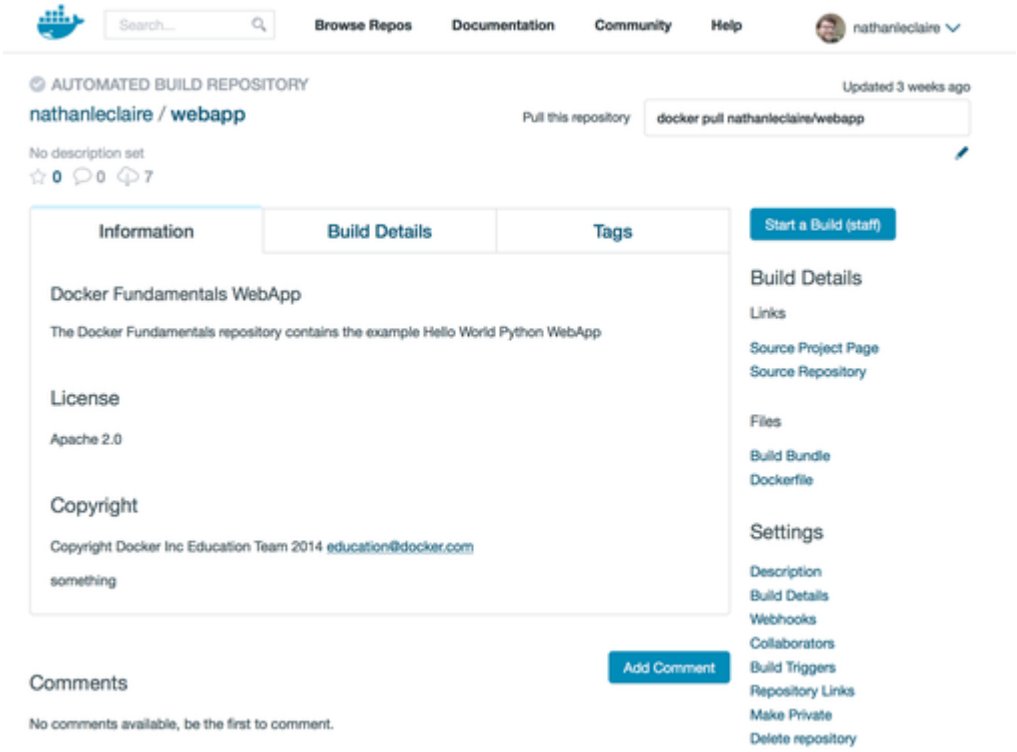
Your account screen

This is the master account screen. Here you can see your repositories and recent activity.



Review your webapp repository

Click on the link to your `my_docker_hub_login/web` repository.



- You can see the basic information about your image.
- You can also browse to the `Tags` tab to see image tags, or navigate to a link in the "Settings" sidebar to configure the repo.

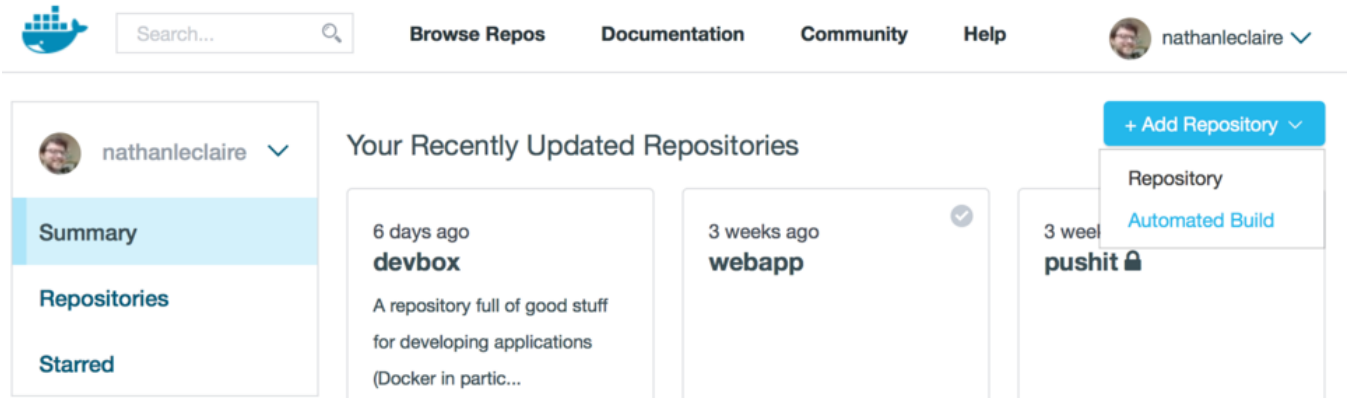
Automated Builds

In addition to pushing images to Docker Hub you can also create special images called *Automated Builds*. An *Automated Build* is created from a `Dockerfile` in a GitHub repository.

This provides a guarantee that the image came from a specific source and allows you to ensure that any downloaded image is built from a `Dockerfile` you can review.

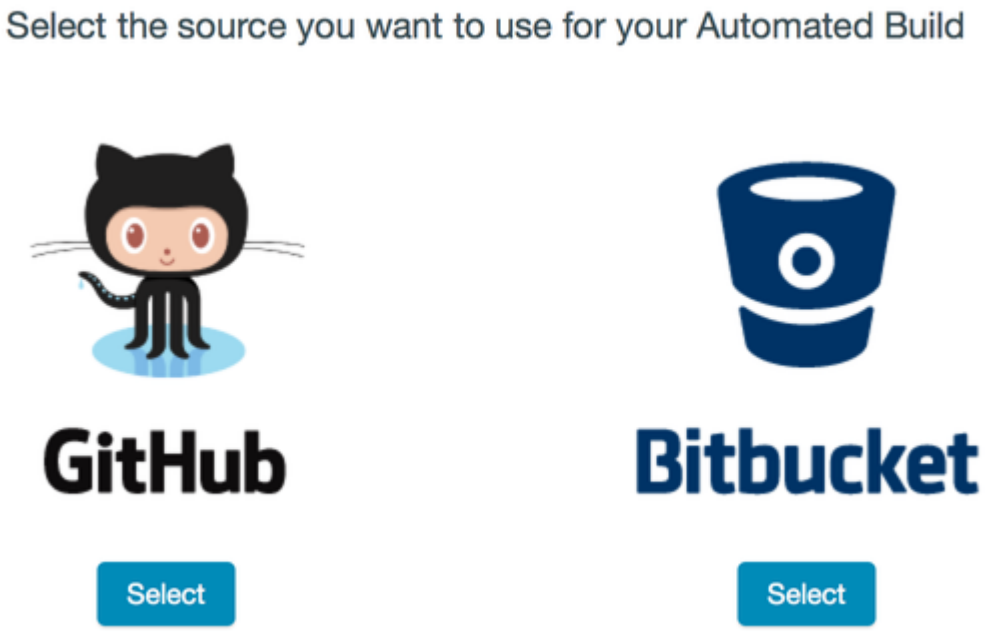
Creating an Automated build

To create an *Automated Build* click on the `Add Repository` button on your main account screen and select `Automated Build`.



Connecting your GitHub account

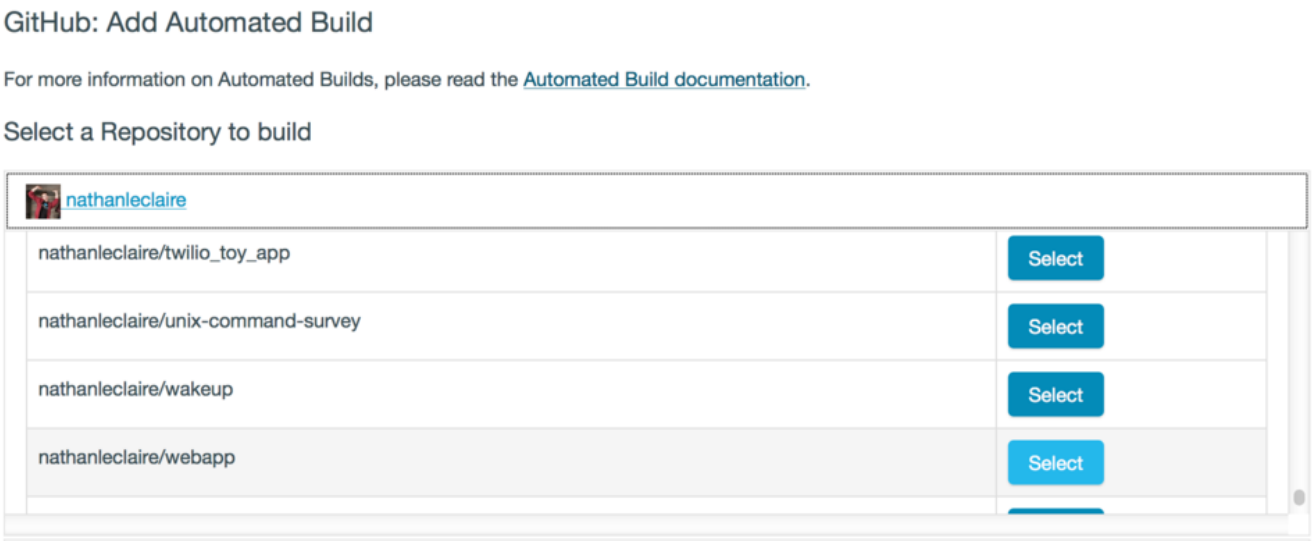
If this is your first *Automated Build* you will be prompted to connect your GitHub account to the Docker Hub.



Select specific GitHub repository

You can then select a specific GitHub repository.

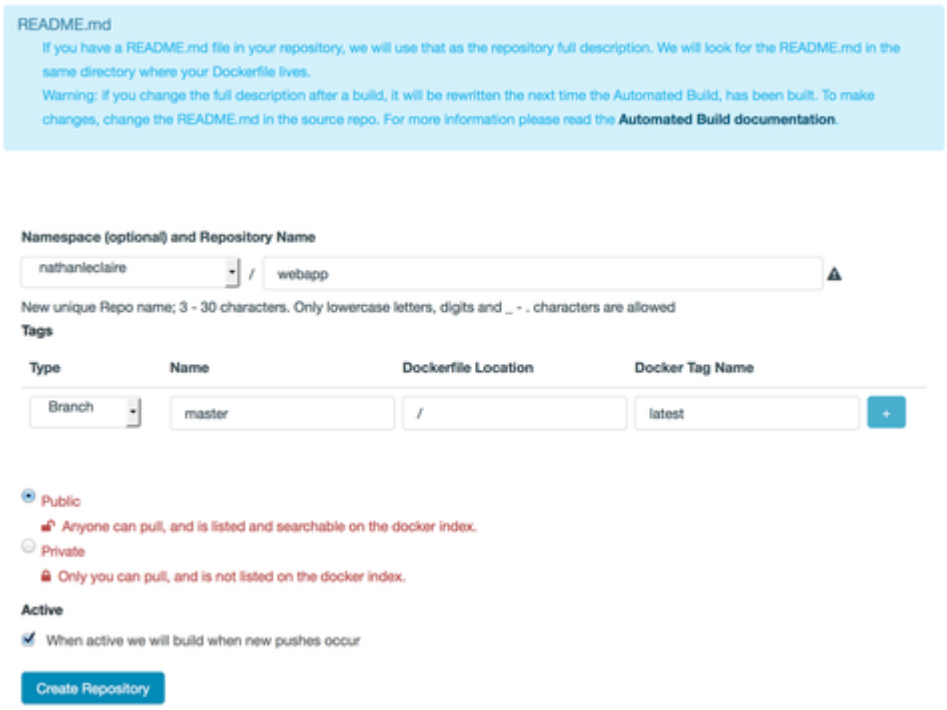
It must contain a `Dockerfile`.



If you don't have a repository with a Dockerfile, you can fork https://github.com/docker-training/staticweb, for instance.

Configuring Automated Build

You can then configure the specifics of your *Automated Build* and click the `Create Repository` button.



Automated Building

Once configured your *Automated Build* will automatically start building an image from the `Dockerfile` contained in your Git repository.

Every time you make a commit to that repository a new version of the image will be built.

Section summary

We've learned how to:

- Pull and push images to the Docker Hub.
- Explore the Docker Hub.
- Understand and create *Automated Builds*.

Local Development Work flow with Docker

Lesson 8: Local Development Workflow with Docker

Objectives

At the end of this lesson, you will be able to:

- Share code between container and host.
- Use a simple local development workflow.

Using a Docker container for local development

Docker containers are perfect for local development.

Let's grab an image with a web application and see how this works.

```
$ docker pull training/namer
```


Our namer image

Our `training/namer` image is based on the Ubuntu image.

It contains:

- Ruby.
- Sinatra.
- Required dependencies.

Adding our source code

Let's download our application's source code.

```
$ git clone https://github.com/docker-training/namer.git
$ cd namer
$ ls
company_name_generator.rb  config.ru  Dockerfile  Gemfile  README.md
```


Creating a container from our image

We've got an image, some source code and now we can add a container to run that code.

```
$ docker run -d -v $(pwd):/opt/namer -w /opt/namer \
-p 80:9292 training/namer rackup
```

We are passing some *flags* as arguments to the `docker run` command to control its behavior:

- The `-d` flag indicates that the container should run in daemon mode (in the background).
- The `-v` flag provides volume mounting inside containers.
- The `-w` flag sets the working directory inside the container.
- The `-p` flag maps port `9292` inside the container to port `80` on the host.

More on these later.

We've launched the application with the `training/namer` image and the `rackup` command.

Mounting volumes inside containers

The `-v` flag mounts a directory from your host into your Docker container. The flag structure is:

```
[host-path]:[container-path]:[rw|ro]
```

- If [host-path] or [container-path] doesn't exist it is created.
- You can control the write status of the volume with the `ro` and `rw` options.
- If you don't specify `rw` or `ro`, it will be `rw` by default.

Checking our new container

Now let us see if our new container is running.

```
$ docker ps
CONTAINER ID  IMAGE                COMMAND CREATED       STATUS
PORTS               NAMES
045885b68bc5  training/namer:latest rackup  3 seconds ago Up 3 seconds
0.0.0.0:80->9292/tcp condescending_shockley
```


Viewing our application

Now let's browse to our web application on:

```
http://<yourHostIP>:80
```

We can see our company naming application.

Making a change to our application

Our customer really doesn't like the color of our text. Let's change it.

```
$ vi company_name_generator.rb
```

And change

```
color: royalblue;
```

To:

```
color: red;
```


Refreshing our application

Now let's refresh our browser:

```
http://<yourHostIP>:80
```

We can see the updated color of our company naming application.

Workflow explained

We can see a simple workflow:

1. Build an image containing our development environment.
   (Rails, Django...)
2. Start a container from that image.
   Use the `-v` flag to mount source code inside the container.
3. Edit source code outside the containers, using regular tools.
   (vim, emacs, textmate...)
4. Test application.
   (Some frameworks pick up changes automatically.
   Others require you to Ctrl-C + restart after each modification.)
5. Repeat last two steps until satisfied.
6. When done, commit+push source code changes.
   (You *are* using version control, right?)

Killing the container

Now we're done let's kill our container.

```
$ docker kill <yourContainerID>
```

And remove it.

```
$ docker rm <yourContainerID>
```


Section summary

We've learned how to:

- Share code between container and host.
- Set our working directory.
- Use a simple local development workflow.

A container to call your own

Lesson 9: A container to call your own

Objectives

At the end of this lesson, you will be able to:

- Understand the different types of containers.
- Start a container.
- See a container's status.
- Inspect a container.
- (Re)Start and attach to a container.

Containers

- Containers are created with the `docker run` command.
- Containers have two modes they run in:
    - Daemonized.
    - Interactive.

Daemonized containers

- Runs in the background.
- The `docker run` command is launched with the `-d` command line flag.
- The container runs until it is stopped or killed.

Interactive containers

- Runs in the foreground.
- Attached a pseudo-terminal, i.e. let you get input and output from the container.
- The container also runs until its controlling process stops or it is stopped or killed.

Launching a container

Let's create a new container from the `ubuntu` image:

```
$ docker run -i -t ubuntu:12.04 /bin/bash
root@268e59b5754c:/#
```

- The `-i` flag sets Docker's mode to interactive.
- The `-t` flag creates a pseudo terminal (or PTY) in the container.
- We've specified the `ubuntu:12.04` image from which to create our container.
- We passed a command to run inside the container, `/bin/bash`.
- That command has launched a Bash shell inside our container.
- The hexadecimal number after `root@` is the container's identifier.
  (The actual ID is longer than that. Docker truncates it for convenience, just like git or hg will show shorter ID instead of full hashes.)

Inside our container

```
root@268e59b5754c:/#
```

Let's run a command.

```
root@268e59b5754c:/# uname -rn
268e59b5754c 3.10.40-50.136.amzn1.x86_64
```

Now let's exit the container.

```
root@268e59b5754c:/# exit
```

After we run `exit` the container stops.

Check the kernel version and hostname again, *outside* the container:

```
[docker@ip-172-31-47-238 ~]$ uname -rn
ip-172-31-47-238.ec2.internal 3.10.40-50.136.amzn1.x86_64
```

The kernel version is the same. Hostname is different.

Container status

You can see container status using the `docker ps` command.

e.g.:

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

What? What happened to our container?

- The `docker ps` command only shows running containers.

Last container status

Since the container has stopped we can show it by adding the `-l` flag. This shows the last run container, running or stopped.

```
$ docker ps -l
CONTAINER ID  IMAGE         COMMAND   CREATED       STATUS  PORTS  NAMES
a2d4b003d7b6  ubuntu:12.04 /bin/bash 5 minutes ago Exit 0         sad_pare
```


The status of all containers

We can also use the `docker ps` command with the `-a` flag. The `-a` flag tells Docker to list all containers both running and stopped.

```
$ docker ps -a
CONTAINER ID        IMAGE                              COMMAND
CREATED             STATUS
PORTS                                NAMES
a2d4b003d7b6        ubuntu:12.04                       /bin/bash              5 minutes
ago       Exit 0
sad_pare
c870de4523bf        nathanleclaire/devbox:latest       /bin/bash -l           16 hours
ago         Up 32 minutes                                                        b2d
acc65c24dceb        training/webapp:latest             python -m SimpleHTTP   16 hours
ago         Up 41 minutes                   5000/tcp, 0.0.0.0:49154->8000/tcp
furious_perlman
833daa3d9708        training/webapp:latest             python -m SimpleHTTP   16 hours
ago         Up 44 minutes
sharp_lovelace
6fb68e7b7451        nathanleclaire/devbox:latest       /bin/bash -l           16 hours
ago         Up 22 minutes                                                        bar
```


What does `docker ps` tell us?

We can see a lot of data returned by the `docker ps` command.

```
CONTAINER ID  IMAGE         COMMAND   CREATED       STATUS  PORTS  NAMES
a2d4b003d7b6  ubuntu:12.04 /bin/bash 5 minutes ago Exit 0         sad_pare
```

Let's focus on some items:

- `CONTAINER ID` is a unique identifier generated by Docker for our container.
  You can use it to manage the container (e.g. stop it, examine it...)
- `IMAGE` is the image used to create that container.
  We did `docker run ubuntu`, and Docker selected `ubuntu:12.04`.
- `COMMAND` is the exact command that we asked Docker to run: `/bin/bash`.
- You can name your containers (with the `--name` option).
  If you don't, Docker will generate a random name for you, like `sad_pare`.
  That name shows up in the `NAMES` column.

Get the ID of the last container

In the next slides, we will use a very convenient command:

```
$ docker ps -l -q
ee9165307acc
```

- `-l` means "show only the last container started".
- `-q` means "show only the short ID of the container".

This shows the ID (and only the ID!) of the last container we started.

Inspecting our container

You can also get a lot more information about our container by using the `docker inspect` command.

```
$ docker inspect $(docker ps -l -q) | less
[{
  "ID": "<yourContainerID>",
  "Created": "2014-03-15T22:05:42.73203576Z",
  "Path": "/bin/bash",
  "Args": [],
  "Config": {
    "Hostname": "<yourContainerID>",
    "Domainname": "",
    "User": "",
  . . .
}]
```

(See? The full ID of the container is longer, as promised earlier!)

Inspecting something specific

We can also use the `docker inspect` command to find specific things about our container, for example:

```
$ docker inspect --format='{{.State.Running}}' $(docker ps -l -q)
false
```

Here we've used the `--format` flag and specified a single value from our inspect hash result. This will return its value, in this case a Boolean value for the container's status.

(Fun fact: the argument to `--format` is a template using the Go `text/template` package.)

Being lazy

- We used `$(docker ps -q -l)` in the previous examples.
- That's a lot to type!
- From now on, we will use the ID of the container.
- In the following examples, you should use the ID of *your* container.

```
$ docker inspect <yourContainerID>
```

- Wait, that's still a lot to type!
- Docker lets us type just the first characters of the ID.

```
$ docker inspect a2d4
```


Restarting our container

You can (re-)start a stopped container using its ID.

```
$ docker start <yourContainerID>
<yourContainerID>
```

Or using its name.

```
$ docker start sad_pare
sad_pare
```

The container will be restarted using the same options you launched it with.

Attaching to our restarted container

Once the container is started you can attach to it. In our case this will attach us to the Bash shell we launched when we ran the container initially.

```
$ docker attach <yourContainerID>
root@<yourContainerID>:/#
```

Note: if the prompt is not displayed after running `docker attach`, just press "Enter" one more time. The prompt should then appear.

You can also attach to the container using its name.

```
$ docker attach sad_pare
root@<yourContainerID>:/#
```

If we ran `exit` here the container would stop again because the `/bin/bash` process would be ended.

You can detach from the running container using `<CTRL+p><CTRL+q>`.

Starting and attaching shortcut

There's also a shortcut we can use that combines the `docker start` and `docker attach` commands.

```
$ docker start -a <yourContainerID>
```

The `-a` flag combines the function of `docker attach` when running the `docker start` command.

Section summary

We've learned how to:

- Understand the different types of containers.
- Start a container.
- See a container's status.
- Inspect a container.
- (Re)Start and attach to a container.

Container Networking Basics

Lesson 10: Container Networking Basics

Objectives

At the end of this lesson, you will be able to:

- Expose a network port.
- Manipulate container networking basics.
- Find a container's IP address.
- Stop and remove a container.

Container Networking Basics

Now we've seen the basics of a daemonized container let's look at a more complex (and useful!) example.

To do this we're going to build a web server in a daemonized container.

Running our web server container

Let's start by creating our new container.

```
$ docker run -d -p 80 training/webapp python -m SimpleHTTPServer 80
72bbff4d768c52d6ce56fae5d45681c62d38bc46300fc6cc28a7642385b99eb5
```

- We've used the `-d` flag to daemonize the container.
- The `-p` flag exposes port `80` in the container.
- We've used the `training/webapp` image, which happens to have Python.
- We've used Python to create a web server.

Checking the container is running

Let's look at our running container.

```
$ docker ps
CONTAINER ID  IMAGE  COMMAND  CREATED  STATUS  PORTS                    NAMES
72bbff4d768c  ...    ...      ...      ...     0.0.0.0:49153->80/tcp  ...
```

- The `-p` flag maps a random high port, here `49153` to port `80` inside the container. We can see it in the `PORTS` column.
- The other columns have been scrubbed out for legibility here.

The `docker port` shortcut

We can also use the `docker port` command to find our mapped port.

```
$ docker port <yourContainerID> 80
0.0.0.0:49153
```

We specify the container ID and the port number for which we wish to return a mapped port.

We can also find the mapped network port using the `docker inspect` command.

```
$ docker inspect -f "{{ .HostConfig.PortBindings }}" <yourContainerID>
{"80/tcp":[{"HostIp":"0.0.0.0","HostPort":"49153"}]}
```


Viewing our web server

We open a web browser and view our web server on `<yourHostIP>:49153`.

Container networking basics

- You can map ports automatically.
- You can also manually map ports.

```
$ docker run -d -p 8080:80 training/webapp \
python -m SimpleHTTPServer 80
```

The `-p` flag takes the form:

```
host_port:container_port
```

This maps port `8080` on the host to port `80` inside the container.

- Note that this style prevents you from spinning up multiple instances of the same image (the ports will conflict).
- Containers also have their own private IP address.

Finding the container's IP address

We can use the `docker inspect` command to find the IP address of the container.

```
$ docker inspect --format '{{ .NetworkSettings.IPAddress }}' <yourContainerID>
172.17.0.3
```

- We've again used the `--format` flag which selects a portion of the output returned by the `docker inspect` command.
- The default network used for Docker containers is 172.17.0.0/16.
  If it is already in use on your system, Docker will pick another one.

Pinging our container

We can test connectivity to the container using the IP address we've just discovered. Let's see this now by using the `ping` tool.

```
$ ping 172.17.0.3
64 bytes from 172.17.0.3: icmp_req=2 ttl=64 time=0.085 ms
64 bytes from 172.17.0.3: icmp_req=2 ttl=64 time=0.085 ms
64 bytes from 172.17.0.3: icmp_req=2 ttl=64 time=0.085 ms
```


Stopping the container

Now let's stop the running container.

```
$ docker stop <yourContainerID>
```


Removing the container

Let's be neat and remove our container too.

```
$ docker rm <yourContainerID>
```


Section summary

We've learned how to:

- Expose a network port.
- Manipulate container networking basics.
- Find a container's IP address.
- Stop and remove a container.

NOTE: Later on we'll see how to network containers without exposing ports using the `link` primitive.

Working with Volumes

Lesson 11: Working with Volumes

Objectives

At the end of this lesson, you will be able to:

- Create containers holding volumes.
- Share volumes across containers.
- Share a host directory with one or many containers.

Working with Volumes

Docker volumes can be used to achieve many things, including:

- Bypassing the copy-on-write system to obtain native disk I/O performance.
- Bypassing copy-on-write to leave some files out of `docker commit`.
- Sharing a directory between multiple containers.
- Sharing a directory between the host and a container.
- Sharing a *single file* between the host and a container.

Volumes are special directories in a container

Volumes can be declared in two different ways.

- Within a `Dockerfile`, with a `VOLUME` instruction.

```
VOLUME /var/lib/postgresql
```

- On the command-line, with the `-v` flag for `docker run`.

```
$ docker run -d -v /var/lib/postgresql
training/postgresql
```

In both cases, `/var/lib/postgresql` (inside the container) will be a volume.

Volumes bypass the copy-on-write system

Volumes act as passthroughs to the host filesystem.

- The I/O performance on a volume is exactly the same as I/O performance on the Docker host.
- When you `docker commit`, the content of volumes is not brought into the resulting image.
- If a `RUN` instruction in a `Dockerfile` changes the content of a volume, those changes are not recorded neither.

Volumes can be shared across containers

You can start a container with *exactly the same volumes* as another one.

The new container will have the same volumes, in the same directories.

They will contain exactly the same thing, and remain in sync.

Under the hood, they are actually the same directories on the host anyway.

```
$ docker run --name alpha -t -i -v /var/log ubuntu bash
root@99020f87e695:/# date >/var/log/now
```

In another terminal, let's start another container with the same volume.

```
$ docker run --volumes-from alpha ubuntu cat /var/log/now
Fri May 30 05:06:27 UTC 2014
```


Volumes exist independently of containers

If a container is stopped, its volumes still exist and are available.

In the last exemple, it doesn't matter if container `alpha` is running or not.

Data containers

A *data container* is a container created for the sole purpose of referencing one (or many) volumes.

It is typically created with a no-op command:

```
$ docker run --name wwwdata -v /var/lib/www busybox true
$ docker run --name wwwlogs -v /var/log/www busybox true
```

- We created two data containers.
- They are using the `busybox` image, a tiny image.
- We used the command `true`, possibly the simplest command in the world!
- We named each container to reference them easily later.

Using data containers

Data containers are used by other containers thanks to `--volumes-from`.

Consider the following (fictious) example, using the previously created volumes:

```
$ docker run -d --volumes-from wwwdata --volumes-from wwwlogs webserver
$ docker run -d --volumes-from wwwdata ftpserver
$ docker run -d --volumes-from wwwlogs pipestash
```

- The first container runs a webserver, serving content from `/var/lib/www` and logging to `/var/log/www`.
- The second container runs a FTP server, allowing to upload content to the same `/var/lib/www` path.
- The third container collects the logs, and sends them to logstash, a log storage and analysis system.

Managing volumes yourself (instead of letting Docker do it)

In some cases, you want a specific directory on the host to be mapped inside the container:

- You want to manage storage and snapshots yourself.
  (With LVM, or a SAN, or ZFS, or anything else!)
- You have a separate disk with better performance (SSD) or resiliency (EBS) than the system disk, and you want to put important data on that disk.
- You want to share your source directory between your host (where the source gets edited) and the container (where it is compiled or executed).

Wait, we already met the last use-case in our example development workflow!

Sharing a directory between the host and a container

```
$ docker run -t -i -v /src/webapp:/var/www/html/webapp ubuntu /bin/bash
```

This will mount the `/src/webapp` directory into the container at `/var/www/html/webapp`.

It defaults to mounting read-write but we can also mount read-only.

```
$ docker run -t -i -v /src/webapp:/var/www/html/webapp:ro ubuntu /bin/bash
```

Those volumes can also be shared with `--volumes-from`.

Share a volume between the host and multiple containers

Let's see how to put both pieces together.

```
$ docker run --name data -v /tmp/volume:/volume busybox true
$ docker run --volumes-from data busybox touch /volume/hello
$ ls -l /tmp/volume/hello
-rw-r--r-- 1 root root 0 May 29 23:27 /tmp/volume/hello
```

- We created a data container named `data`, for `/volume`, mapped to `/tmp/volume` on the host.
- We created another container, using this volume, and creating a file there.
- From the host, we checked that the file appeared as expected.

Checking volumes defined by an image

Wondering if an image has volumes? Just use `docker inspect`:

```
$ # docker inspect training/datavol
[{
  "config": {
    . . .
    "Volumes": {
        "/var/webapp": {}
    },
    . . .
}]
```


Checking volumes used by a container

To look which paths are actually volumes, and to what they are bound, use `docker inspect` (again):

```
 $ docker inspect cadb250b18be
 [{
   "ID": "<yourContainerID>3d483d3e1a647edbd5683fd6aeae0dd87810d5f523e2d7cb781c",
 . . .
   "Volumes": {
      "/var/webapp": "/var/lib/docker/vfs/dir/
f4280c5b6207ed531efd4cc673ff620cef2a7980f747dbbcca001db61de04468"
   },
   "VolumesRW": {
      "/var/webapp": true
   },
 }]
```

- We can see that our volume is present on the file system of the Docker host.

Sharing a single file between the host and a container

The same `-v` flag can be used to share a single file.

```
$ echo 4815162342 > /tmp/numbers
$ docker run -t -i -v /tmp/numbers:/numbers ubuntu bash
root@274514a6e2eb:/# cat /numbers
4815162342
```

- All modifications done to `/numbers` in the container will also change `/tmp/numbers` on the host!

It can also be used to share a *socket*.

```
$ docker run -t -i -v /var/run/docker.sock:/docker.sock ubuntu bash
```

- This pattern is frequently used to give access to the Docker socket to a given container.

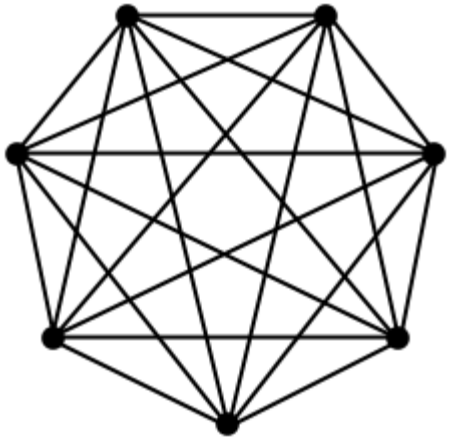
Warning: when using such mounts, the container gains access to the host. It can potentially do bad things.

Section summary

We've learned how to:

- Create containers holding volumes.
- Share volumes across containers.
- Share a host directory with one or many containers.

Connecting Containers



Lesson 12: Connecting containers

Objectives

At the end of this lesson, you will be able to:

- Launch named containers.
- Create links between containers.
- Use names and links to communicate across containers.
- Use these features to decouple app dependencies and reduce complexity.

Connecting containers

- We will learn how to use names and links to expose one container's port(s) to another.
- Why? So each component of your app (e.g., DB vs. web app) can run independently with its own dependencies.

What we've got planned

- We're going to get two images: a PostgreSQL database image and a Ruby on Rails application image.
- We're going to start containers from each image.
- We're going to link the containers running our Rails application and database using Docker's link primitive.

Our PostgreSQL database image

Let's start by pulling down our database image.

```
$ docker pull training/postgres
```

And reviewing it.

```
$ docker images training/postgres
REPOSITORY         TAG    IMAGE ID     CREATED           VIRTUAL SIZE
training/postgres  latest 4b3bbfac8bad About an hour ago 335.4 MB
```


The `training/postgres` Dockerfile

Let's look at the Dockerfile that built this image.

It's located in a repository online:

https://github.com/docker-training/postgres/blob/master/Dockerfile

The `training/postgres` Dockerfile

```
FROM ubuntu:14.04
MAINTAINER Docker Education Team <education@docker.com>

ENV PG_VERSION 9.3
RUN apt-get update
RUN apt-get -y install postgresql postgresql-client postgresql-contrib

RUN echo "host    all             all             0.0.0.0/0 trust" >> /etc/
postgresql/$PG_VERSION/main/pg_hba.conf
RUN echo "listen_addresses='*'" >> /etc/postgresql/$PG_VERSION/main/postgresql.conf

RUN service postgresql start && \
 su postgres sh -c "createuser -d -r -s docker" && \
 su postgres sh -c "createdb -O docker docker" && \
 su postgres sh -c "psql -c \"GRANT ALL PRIVILEGES ON DATABASE docker to docker;\""

EXPOSE 5432
CMD ["su", "postgres", "-c", "/usr/lib/postgresql/$PG_VERSION/bin/postgres -D /var/
lib/postgresql/$PG_VERSION/main/ -c config_file=/etc/postgresql/$PG_VERSION/main/
postgresql.conf"]
```


The `training/postgres` Dockerfile in detail

- Based on the `ubuntu:14.04` base image.
- Specifies a number of environment variables.
- Installs the required packages.
- Configures the PostgreSQL database settings.
- Exposes port `5432`.
- Runs the PostgreSQL database when a container is launched from the image.

More about our `EXPOSE` instruction

We saw EXPOSE earlier. We were exposing port `22` to allow SSH connections. Here we see it again.

```
EXPOSE 5432
```

This time we will expose a port to our Rails application in another container, not to the host. But the `Dockerfile` syntax is identical.

Launch a container from the `training/postgres` image.

Let's launch a container from the `training/postgres` image.

```
$ docker run -d --name database training/postgres
```

Let's check the container is running:

```
$ docker ps -l
CONTAINER ID IMAGE                    COMMAND               CREATED
STATUS         PORTS    NAMES
491b16146e5a training/postgres:latest  su postgres -c /usr/  20 seconds ago Up 20
seconds  5432/tcp database
```

- Our container is launched and running a PostgreSQL database.
- Using the `--name` flag we've given it a name: `database`. Container names are unique. We're going to use that name shortly.

Our Rails application image

Let's start by pulling down our Rails application image.

```
$ docker pull training/notes
```

And reviewing it.

```
$ docker images training/notes
REPOSITORY         TAG    IMAGE ID     CREATED           VIRTUAL SIZE
training/notes  latest    7618ecf3ead8 About an hour ago 450.6 MB
```


The `training/notes` Dockerfile

Let's look at the Dockerfile that build this image.

```
FROM ubuntu:14.04
MAINTAINER Docker Education Team <education@docker.com>

RUN apt-get update
RUN apt-get install -y ruby ruby-dev libpq-dev build-essential
RUN gem install sinatra bundler --no-ri --no-rdoc

ADD . /opt/dockernotes

WORKDIR /opt/dockernotes
RUN bundle install

EXPOSE 3000
CMD bundle exec rails s
```


The `training/notes` Dockerfile in detail

- Based on the `ubuntu` base image but using the `14.04` tag to get Ruby 1.9.3.
- Installs the required packages.
- Adds the Rails application itself to the `/opt/dockernotes` directory.
- Exposes port `3000`.
- Runs Ruby on Rails when a container is launched from the image.

Launch a container from the `training/notes` image.

Let's launch a container from the `training/notes` image.

Rather than run the application we're going to try to setup its database schema.

```
$ docker run --name rails training/notes rake db:create db:migrate
```

And we discover that our container has failed to start.

```
could not connect to server: No such file or directory
Is the server running locally and accepting connections on Unix domain socket "/var/
rakeunningun/postgresql/.s.PGSQL.5432"?
```

Without access to a database the Rails application will fail.

Names are unique

Remember we learned earlier that container names are unique.

You can only have one container named `rails` at a time.

So let's delete our container so we can use that name again.

```
$ docker rm rails
```


Launch and link a container

Let's try again but this time we'll link our container to our existing PostgreSQL container.

The special `rake` command below is a one-time process that needs to be done in all Rails applications to initialize the database.

```
$ docker run -i -t  --name rails --link database:db training/notes \
  rake db:create db:migrate
==  CreateNotes: migrating
-- create_table(:notes)
-> 0.0101s
==  CreateNotes: migrated (0.0103s)
```

Woot! That's more like it.

- The `--link` flag connects one container to another.
- We specify the name of the container to link to, `database`, and an alias for the link `db`.

Names are still unique

We started the previous container with `--name rails`.

So once again, we should clean up after ourselves.

```
$ docker rm rails
```


More about our link

The link provides a secure tunnel between containers. On our `database` container port `5432` has been exposed to the linked container.

Docker will automatically set environment variables in our container, to indicate connection information.

Let's see that information:

```
$ docker run --rm --link database:db training/notes env
HOSTNAME=4d6f59f6e3a1
DB_NAME=/sharp_poincare/db
DB_PORT_5432_TCP_ADDR=172.17.0.2
DB_PORT=tcp://172.17.0.2:5432
DB_ENV_LANG=en_US.UTF-8
DB_PORT_5432_TCP=tcp://172.17.0.2:5432
DB_ENV_LOCALE=en_US
DB_ENV_LANGUAGE=en_US.UTF-8
DB_ENV_LC_ALL=en_US.UTF-8
DB_PORT_5432_TCP_PORT=5432
DB_PORT_5432_TCP_PROTO=tcp
DB_ENV_PG_VERSION=9.2
```

- Each variables is prefixed with the link alias: `db`.
- Includes connection information PLUS any environment variables set in the `database` container via `ENV` instructions.
- `--rm` removes the container after it exits, as this is not a long running container.

Starting our Rails application

Now we've configured our database let's start the application itself in a fresh container:

```
$ docker run -d -p 80:3000 --name rails --link database:db training/notes
```

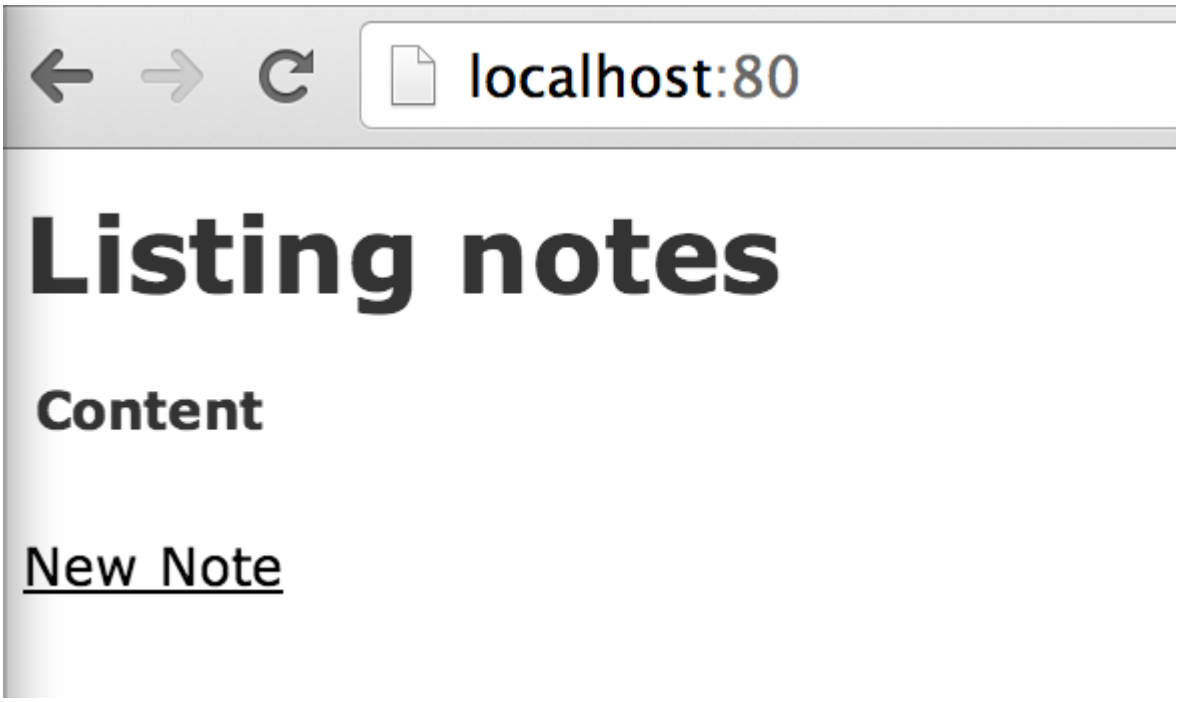
Now let's check the container is running.

```
$ docker ps -l
CONTAINER ID IMAGE                COMMAND             CREATED       STATUS
PORTS                NAMES
02552fa0d0ed training/notes:latest /bin/sh -c bundle ex 2 minutes ago Up 2 minutes
0.0.0.0:80->3000/tcp elegant_thompson
```


Viewing our Rails application

Finally, let's browse to our application and confirm it's working.

```
http://<yourHostIP>
```



Tidying up

Finally let's tidy up our application and database.

```
$ docker kill rails database
. . .
$ docker rm rails database
```

- We can use the container names to stop and remove them.
- We removed them so we can re-use their names later if we want. (Remember container names are unique!)

Section summary

We've learned how to:

- Launch named containers.
- Create links between containers.
- Use names and links to communicate across containers.
- Use these features to decouple app dependencies and reduce complexity.

Using Docker for testing

Lesson 13: Using Docker for testing

Objectives

At the end of this lesson, you will be able to:

- Dockerize a Flask (Python) web application and its tests
- Integrate this Dockerized application with Jenkins for CI
- Use Web Hooks with Automated Builds to automate testing and deployment.

Docker for software testing

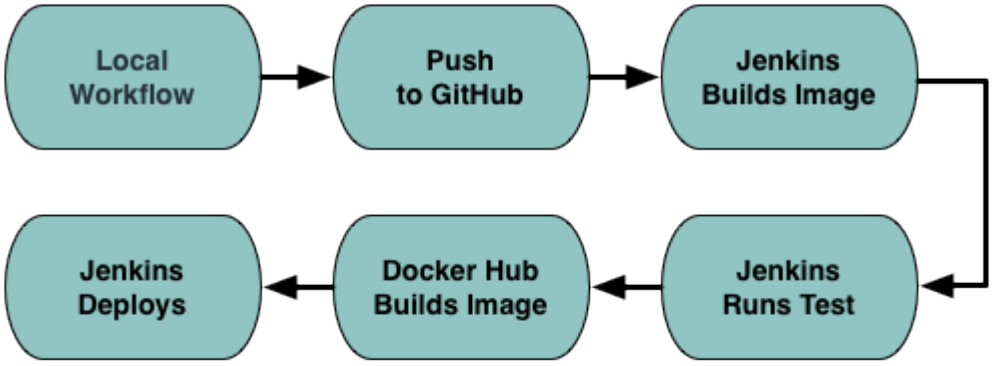
One of Docker's popular use cases is to better enable testing.

We're going to see how to integrate Docker into a Continuous Integration workflow.

To do this we're going to assume that every time we update an application that we want to rebuild its image and run the application's tests.

What we want

1. Every time our code changes we want our image rebuilt.
2. Every time our code changes we want the tests run against our new image.
3. If the tests are good, deploy the new code.



The plan

- Use the Python Flask `webapp` web application we used earlier in the training.
- Look at the `Dockerfile` for that application.
- Look at the test suite for that application.
- Create a Automated Build for that application.
- Configure a Build Trigger that Jenkins will call when tests have passed.
- Configure a Web Hook that Jenkins can use to trigger a deployment of the image.

Why?

This flow is designed to use all of the power of Docker Hub to distribute and maintain your repositories and images. It allows you to maintain the integrity and availability of images to all of your hosts. It also allows you to easily utilize any future Docker Hub features.

- We recommend using Docker Hub's repositiories as a known-good source of images for your deployments.
- The CI system (Jenkins in this case) should run tests inside of the same environments as development and production.
- Docker Hub should build all images that will be deployed to production environments.
- The CI system should be able to block Docker Hub from building an image if there is a test failure.
- The deployment system (also Jenkins in this case) should only deploy images that are built on Docker Hub.

Our web application

Remember the Python Flask web application, originally named `webapp`, that we used earlier?

We're going to re-use this application as our example of using Docker for continous integration.

The webapp Dockerfile

Let's look at the `Dockerfile` for that application again.

```
FROM ubuntu:12.04
MAINTAINER Docker Education Team <education@docker.com>
RUN apt-get update
RUN apt-get install -y -q curl \
python-all python-pip wget
ADD ./webapp /opt/webapp/
WORKDIR /opt/webapp
RUN pip install -r requirements.txt
EXPOSE 5000
CMD ["python", "app.py"]
```

You can see the `Dockerfile` and the application here

The webapp Dockerfile explained

Our `Dockerfile` is pretty simple:

- We use an Ubuntu 12.04 image as the base.
- We install some required prerequisites like Python.
- We add the contents of the `webapp` directory that holds our application code.
- We specify the working directory of the image as `/opt/webapp`.
- We install the applications Python-based dependencies using `pip`.
- We expose port 5000 to serve our application on.
- Finally, our application is launched using `python app.py`.

Our Flask tests

Inside our application, in the `tests.py` file, we've also got a simple Python test for our Flask application.

```
from app import app

import os
import unittest

class AppTestCase(unittest.TestCase):

    def test_root_text(self):
        tester = app.test_client(self)
        response = tester.get('/')
        assert 'Hello world!' in response.data

if __name__ == '__main__':
    unittest.main()
```

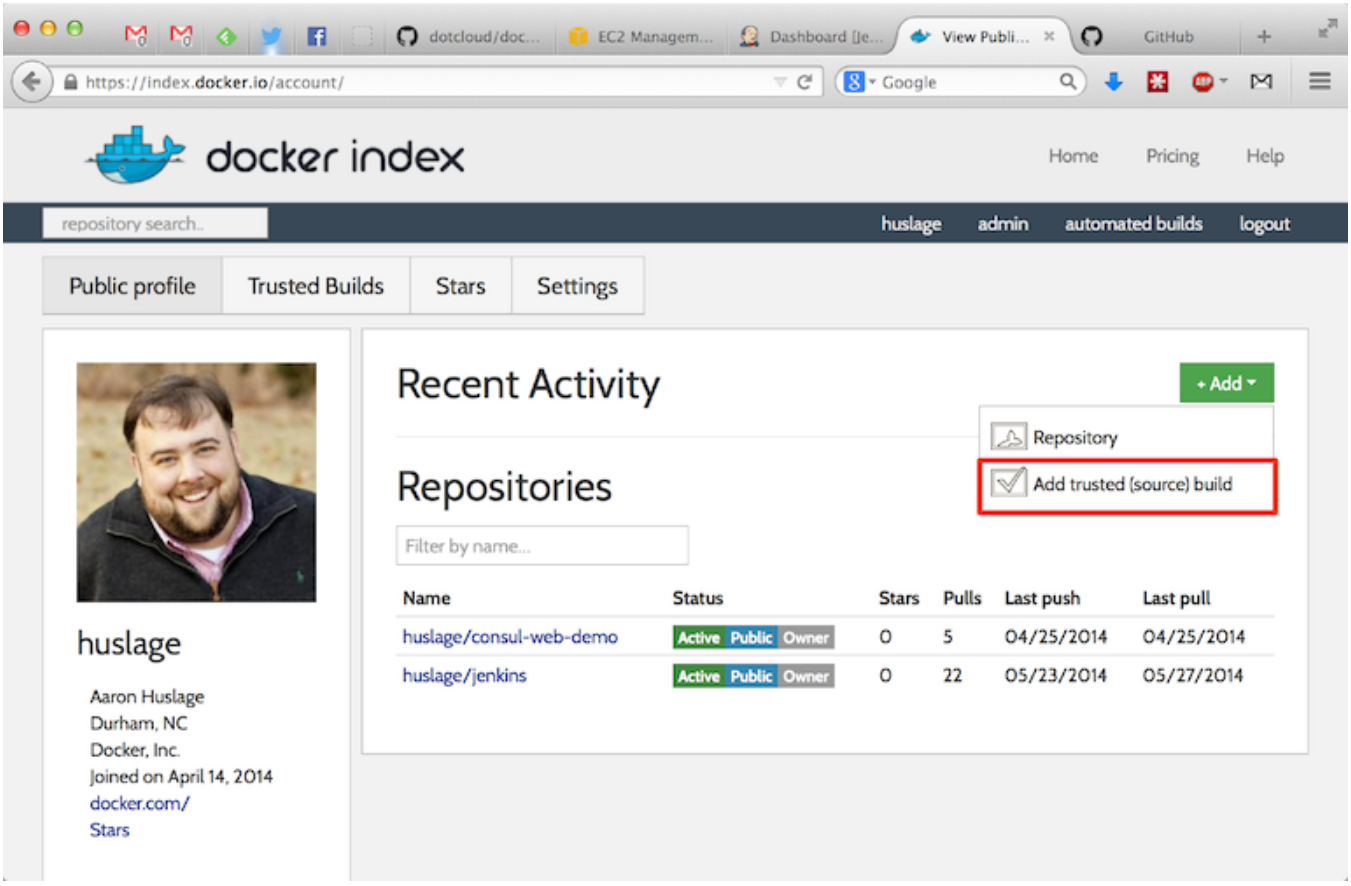

Forking our GitHub repository

We need a place to put our code changes that will trigger this whole flow. Let's fork the docker-training/webapp repository.

Go to http://github.com/docker-training/webapp. Click `Fork` in the upper-right corner. This will create a new repository under your account to hold changes that you make.

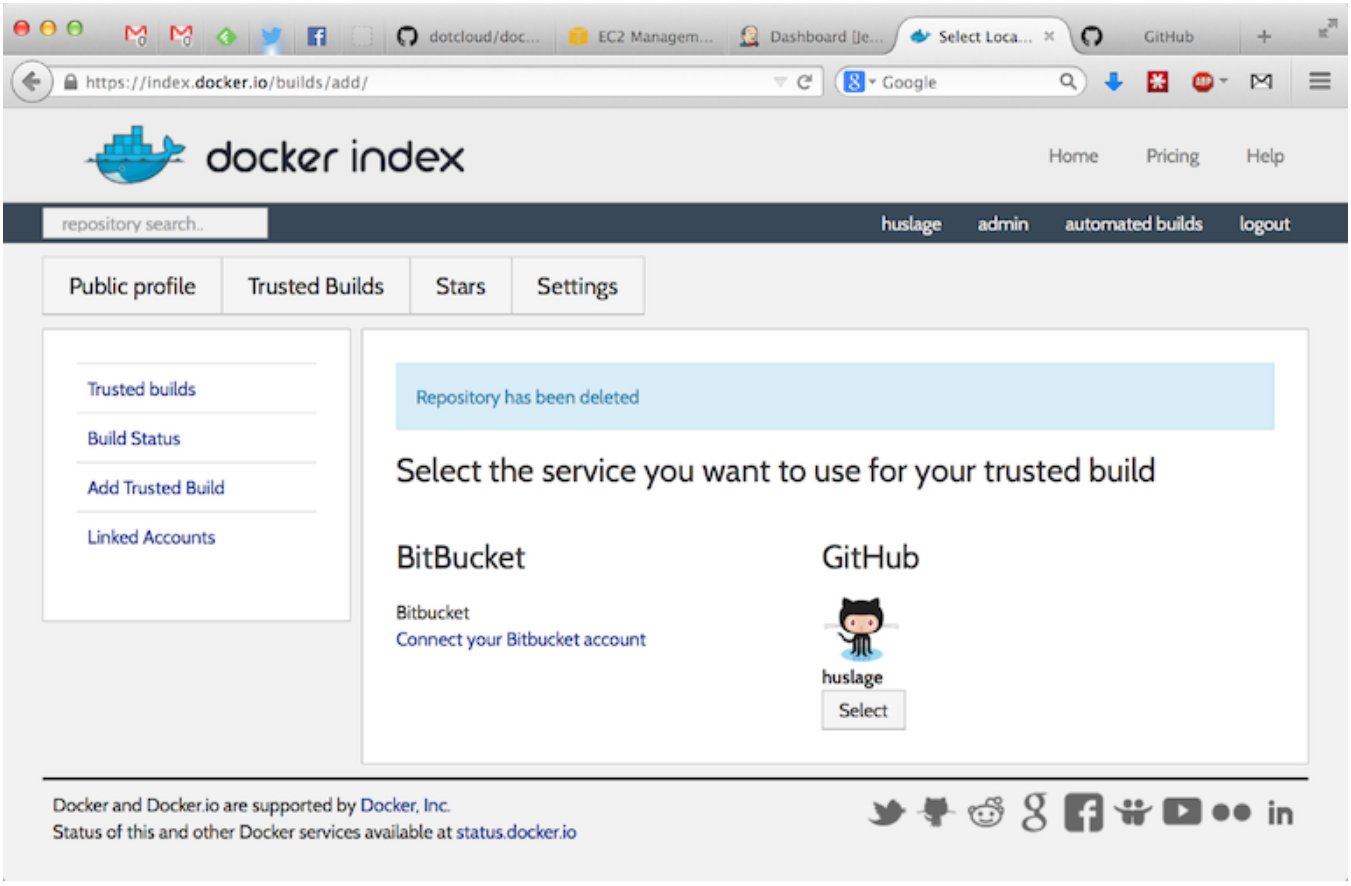
Adding a new Automated Build

Go to your Docker Hub profile page and click `Add automated (source) build`



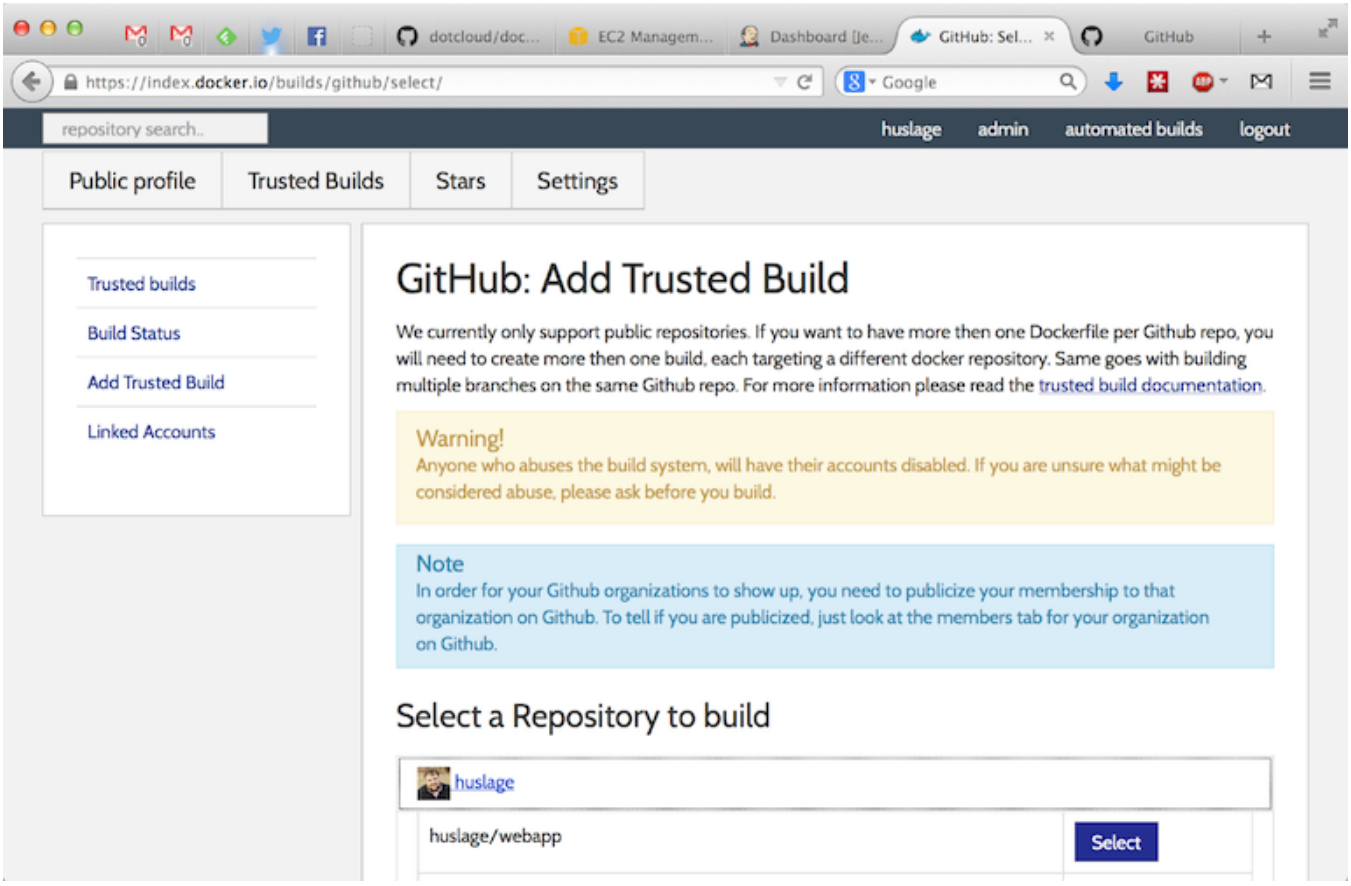
Selecting the Source Code Service

Click on `GitHub` for the service you want to use.



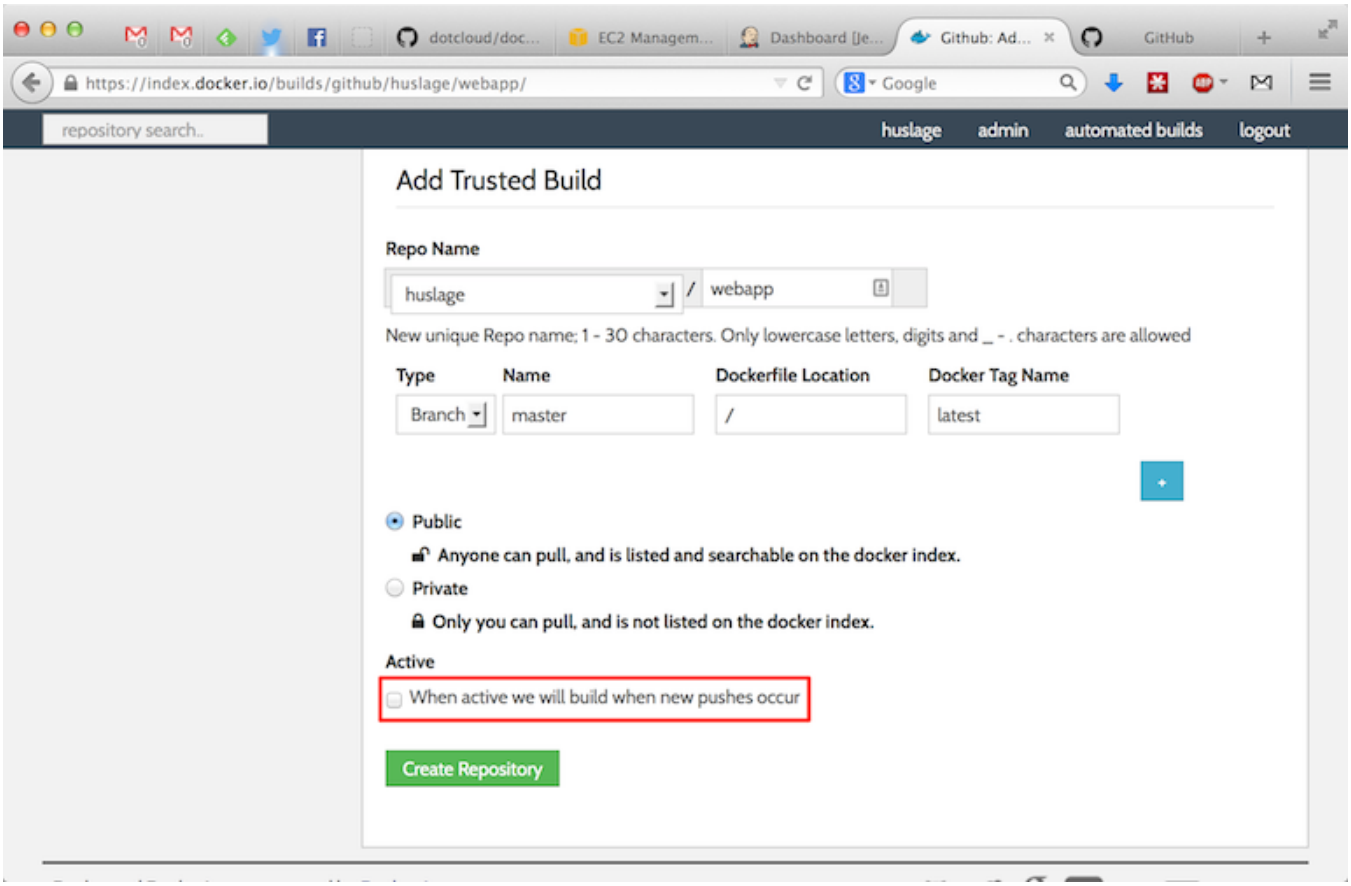
Selecting the repository

We then select the `webapp` repository from your GitHub Account.



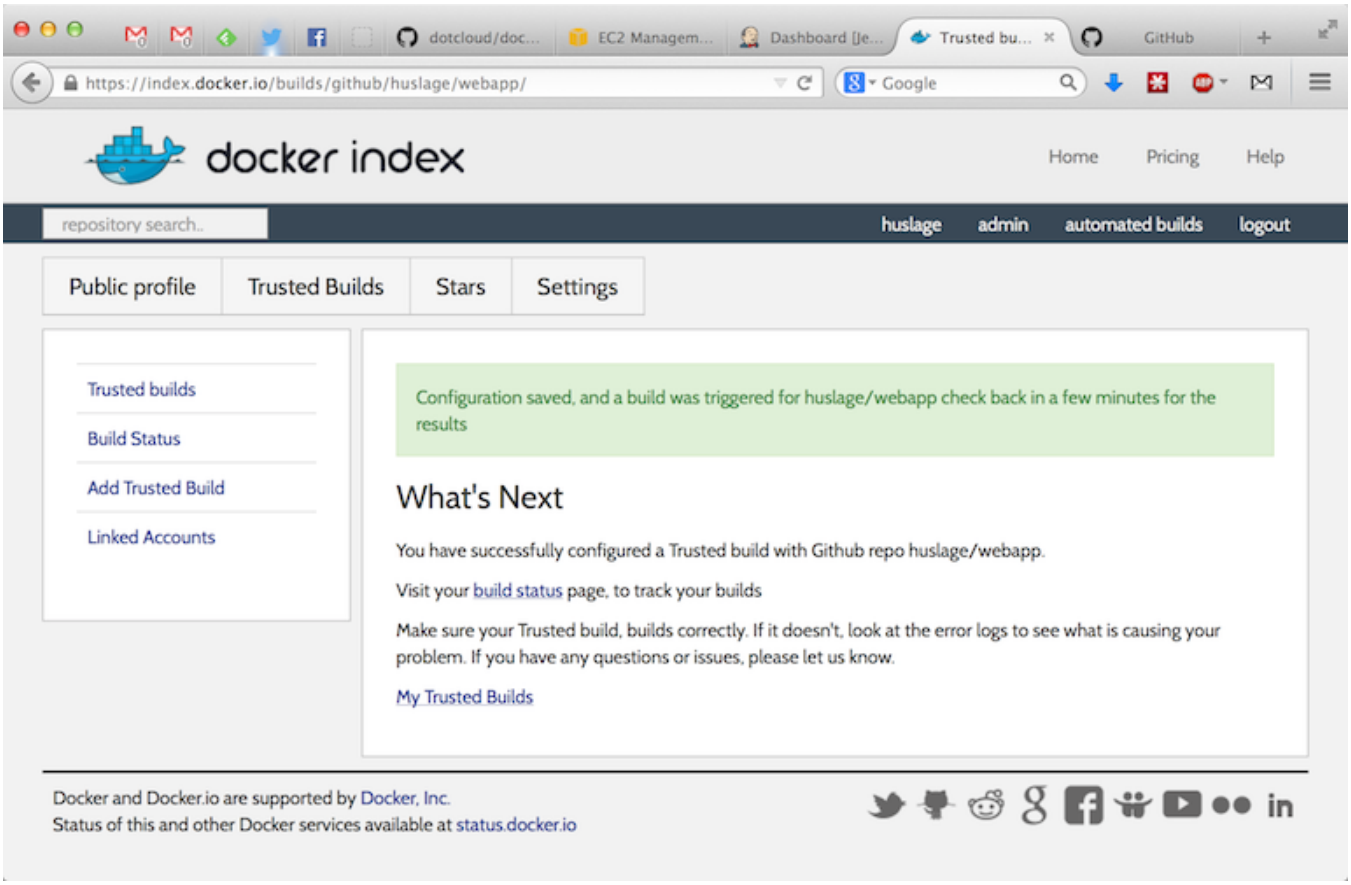
Configuring the build

We then configure our Automated Build and click the `Create Repository` link. Make sure to un-check the `When active we will build when new pushes occur` box.



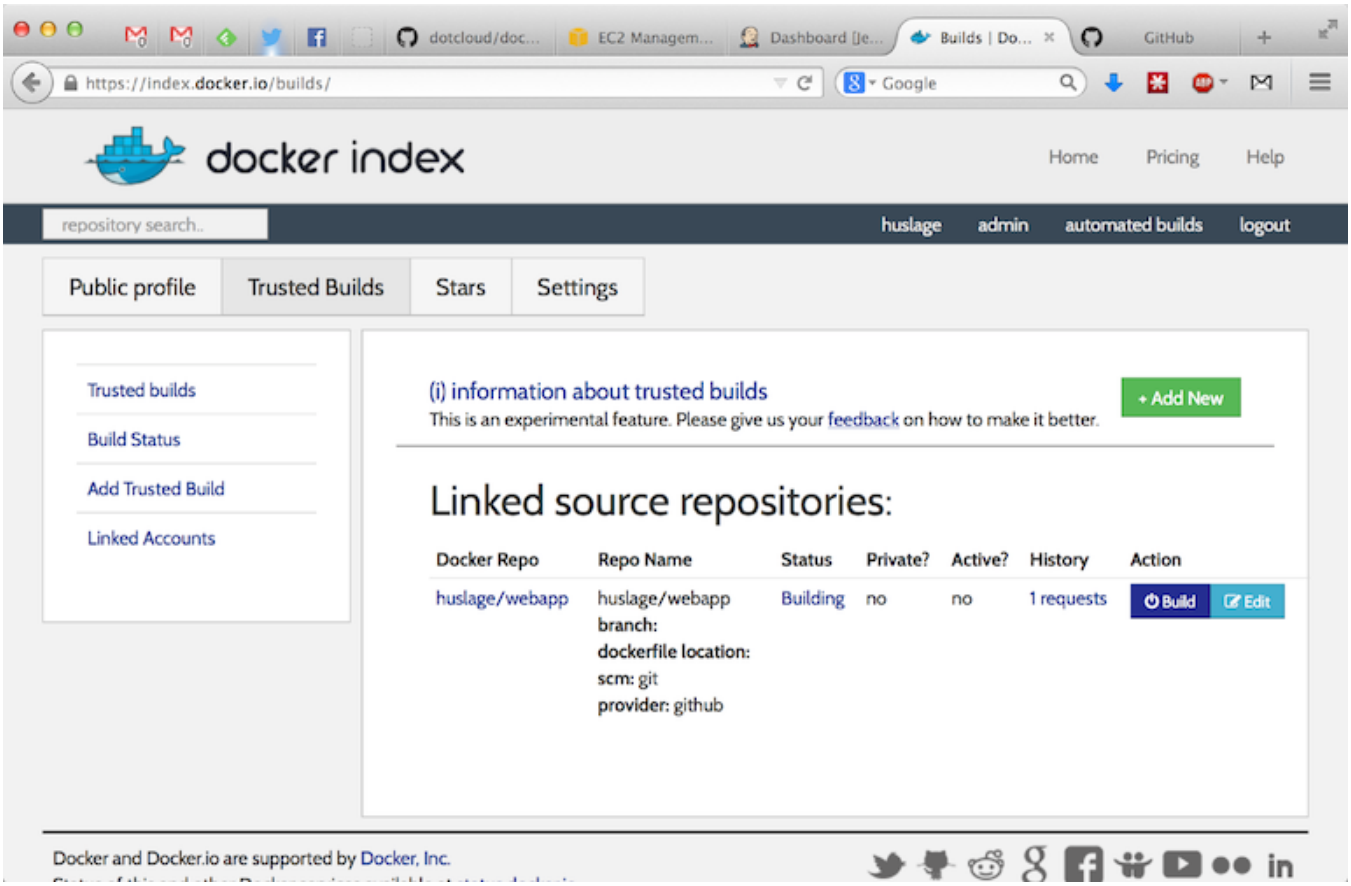
Success

This is what success looks like. Drink it in.



Build status page

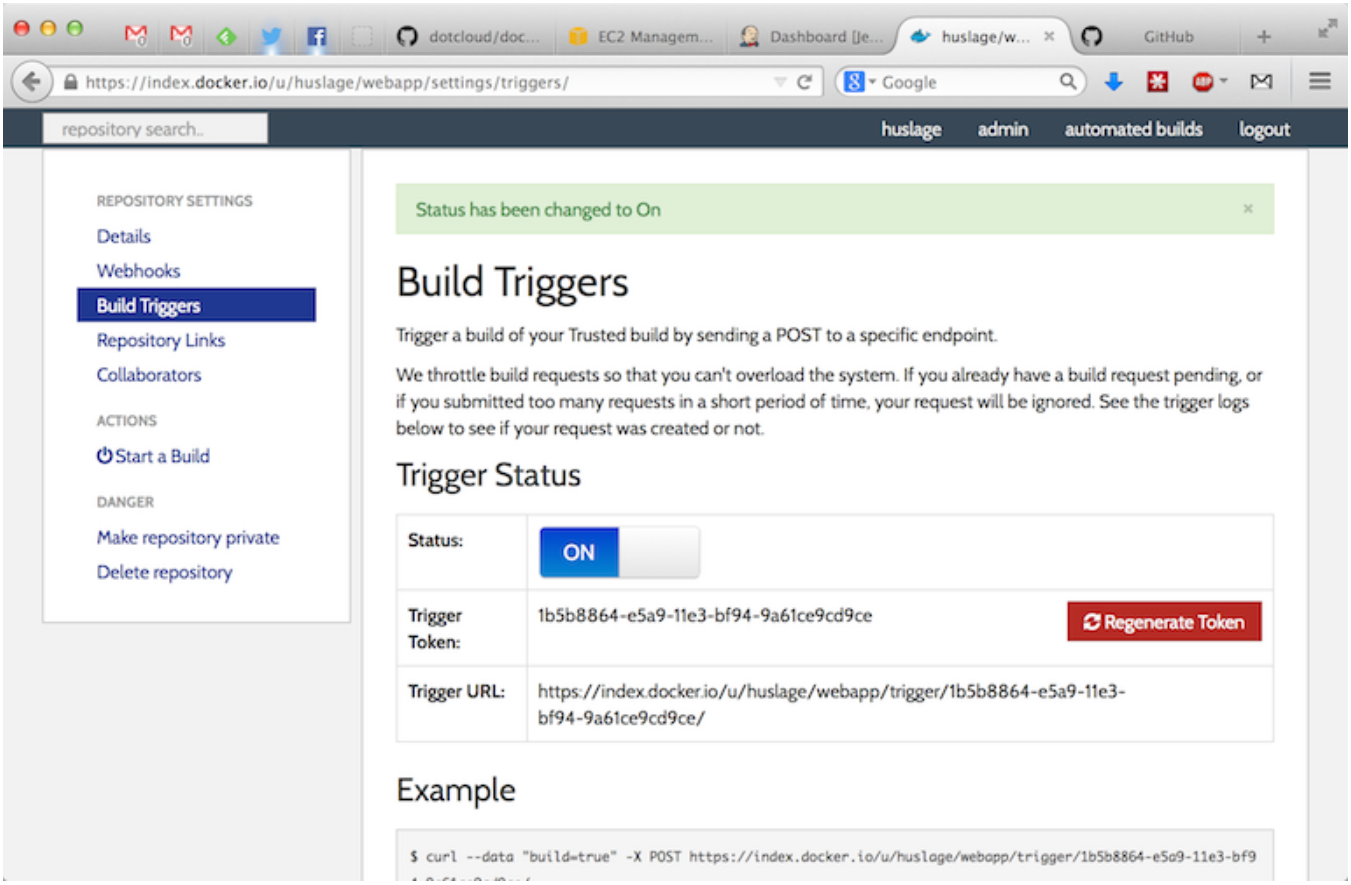
If you click on `Automated Builds` on the left sidebar, you can view the status of your build. When you create a new build, we automatically run it the first time. How handy!



Configure a build trigger

Now we need a way to tell Docker Hub when to build the container.

Go to the repository's page, click `Settings` and `Build Triggers`. Slide the switch to `On` and copy the URL that's generated.



Run Jenkins

We can use an existing Docker image to run Jenkins, `training/jenkins`. Let's run a Jenkins instance now from this image.

```
$ docker run --privileged -d --name=jenkins -p 8080:8080 -e PORT=4444 \
 -e DOCKERHUB_ID=<Your Docker Hub ID> \
 -e DOCKERHUB_EMAIL=<Your Docker Hub Email> \
 -e GITHUB_ID=<Your GitHub ID> \
 training/jenkins
```

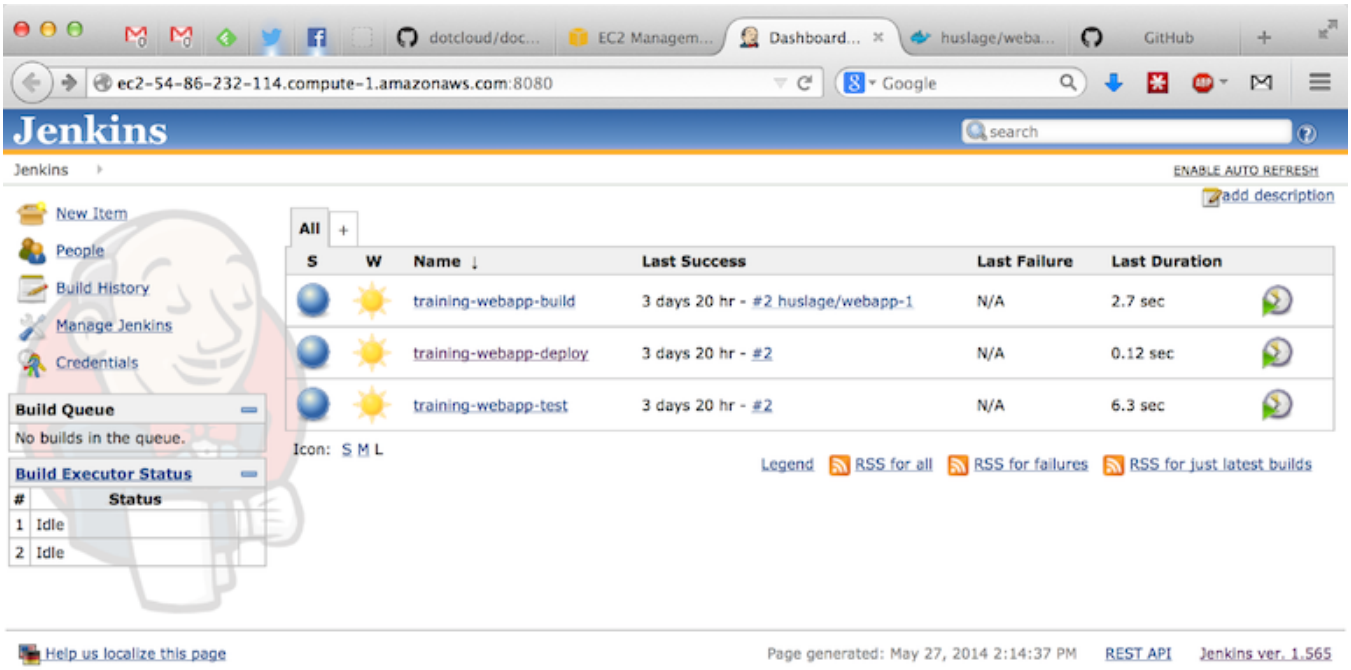
This will launch a container from our `training/jenkins` image and bind it to port `8080` on our local host. The `-e` options pass environment variables to the script that runs Jenkins. They configure the built-in jobs so that you don't have to.

This image runs docker inside of the container, so it needs the `--privileged` flag to run.

P.S. If you're interested you can see the `Dockerfile` that built this image.

Configure Jenkins

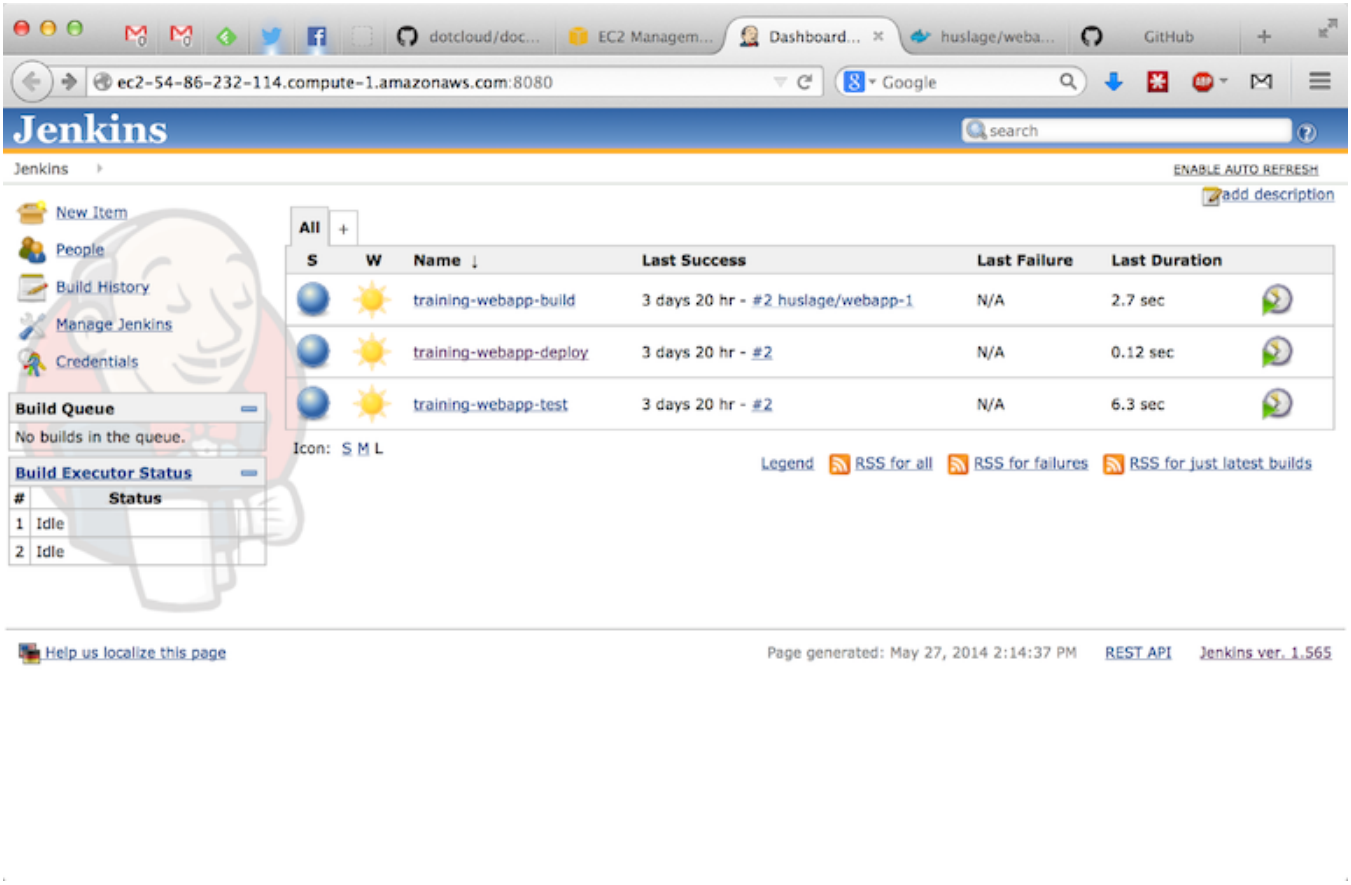
Our Jenkins image is already pre-populated with jobs that will run our Webapp tests. Go to the URL of your Jenkins instance



Configure Jenkins

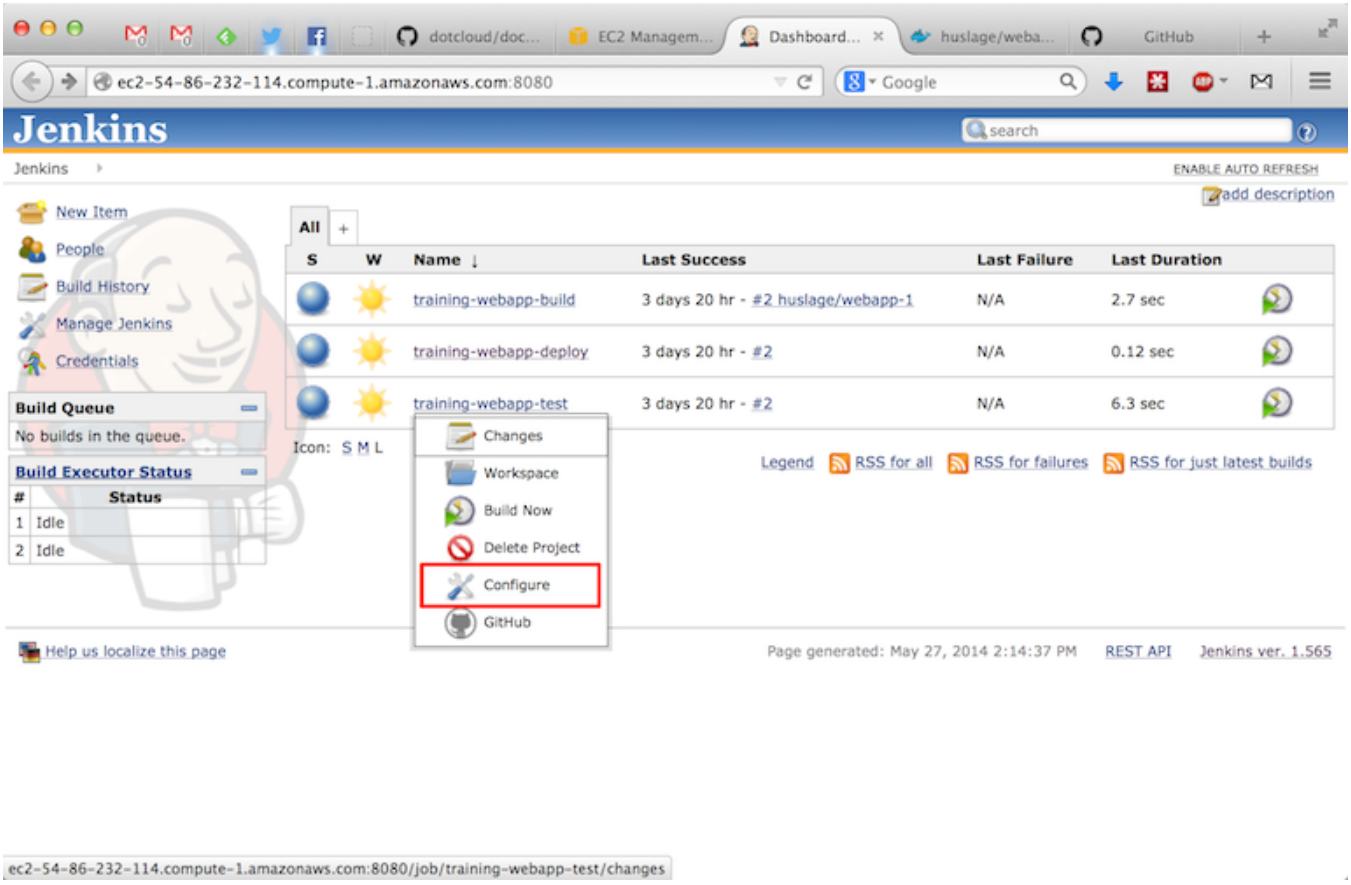
These three jobs will:

1. Build a container from the `webapp` GitHub repository we forked earlier.
2. Run the tests inside that container and return the results.
3. Trigger a build.
4. Simulate the deployment of the code.



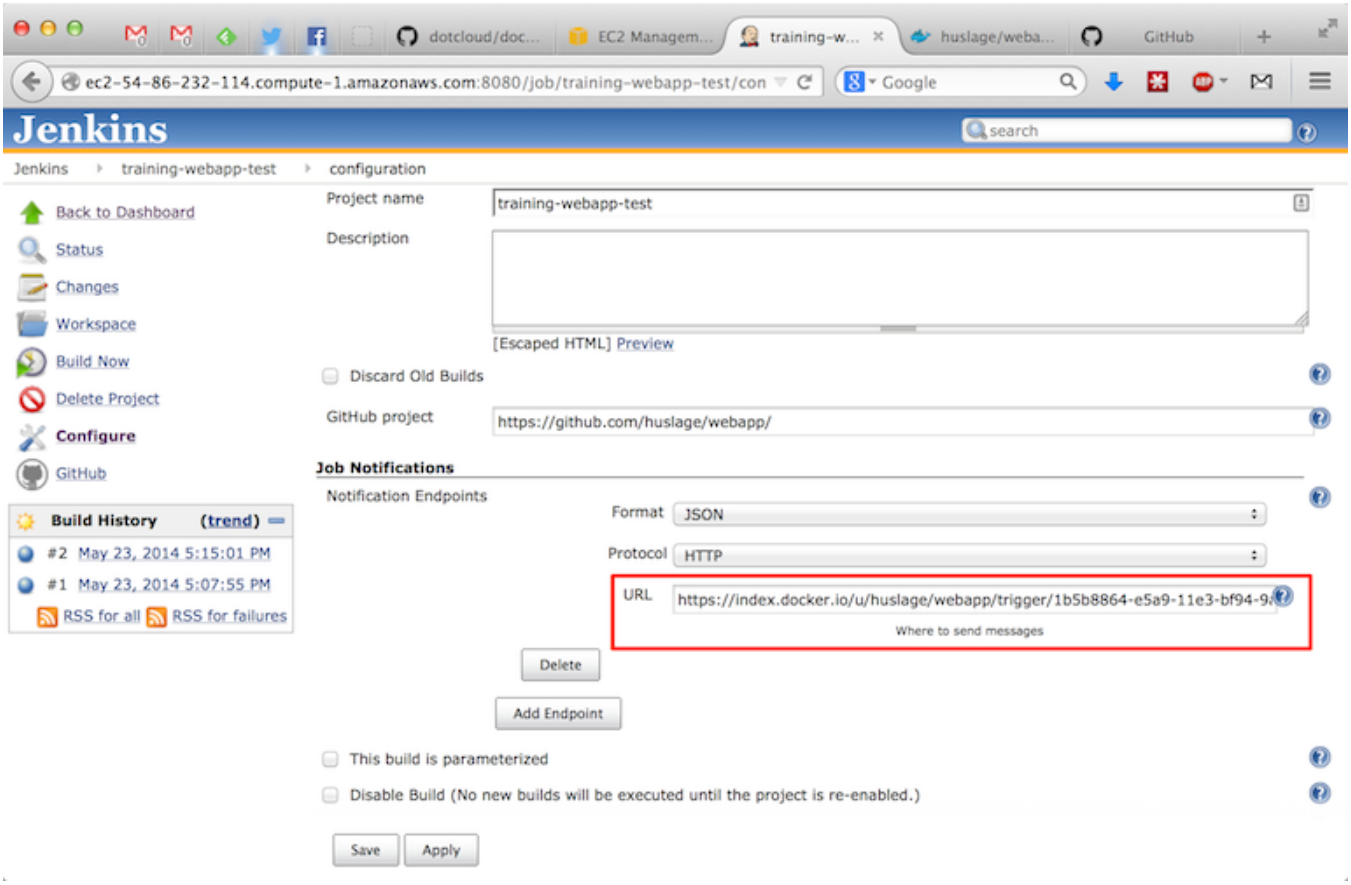
Configuring Test Job

We need to paste that Build Trigger into the `training-webapp-test` job so that it will trigger a build after the tests pass. Hover over the job name and pull down the menu. Select `Configure`.



Configuring Test Job

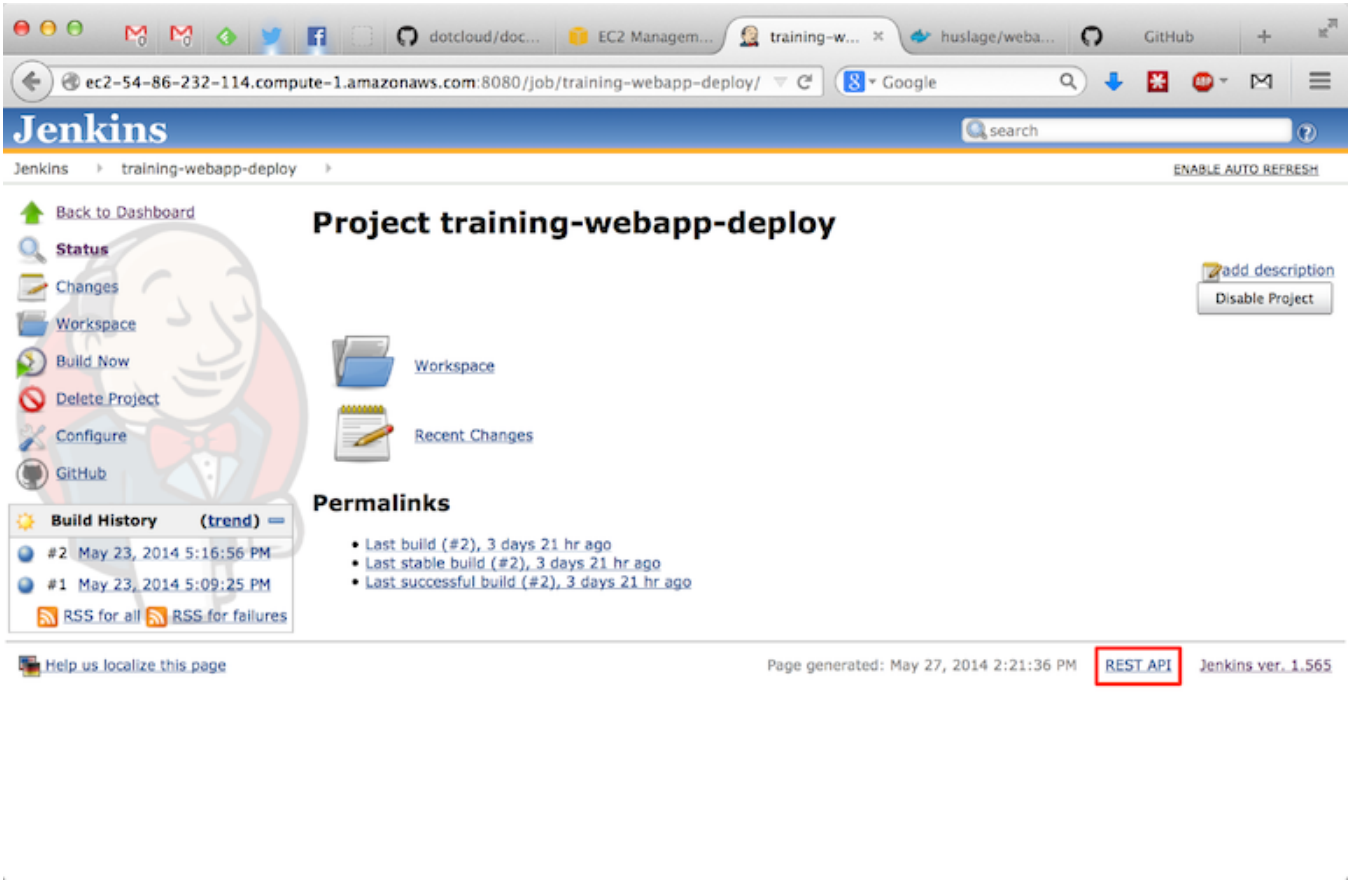
Paste the Build Trigger URL under `Job Notifications`



Click `Save` at the bottom of the page. Return to the Jenkins Dashboard.

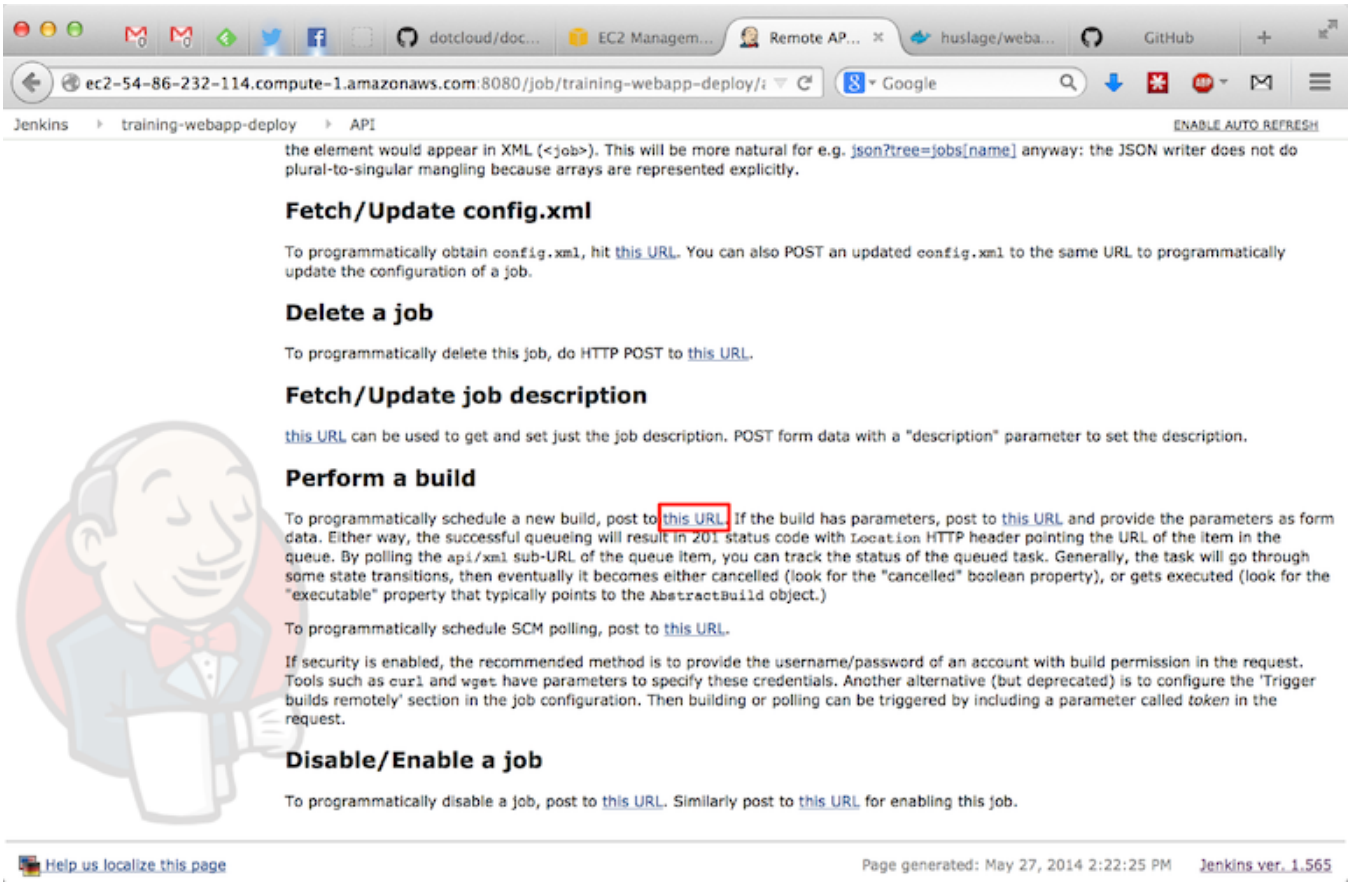
Configuring Deploy Job

We're almost done. Now we need to set up a trigger for the deployment step. From the Jenkins Dashboard click the `training-webapp-deploy` job. In the bottom-right of the page you will see a link for the `REST API`. Click that link.

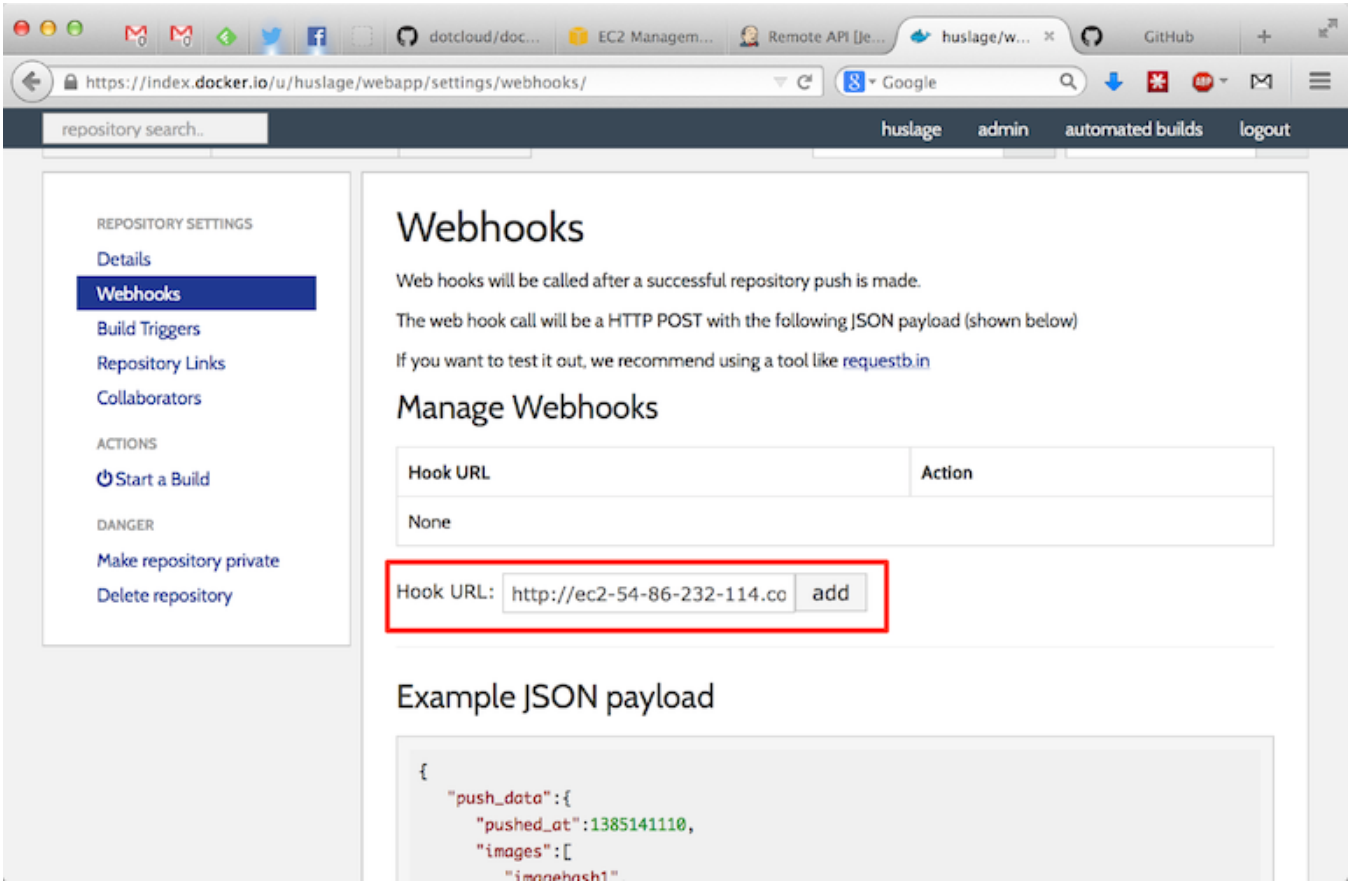


Get the build link URL

Scroll down a bit to the `Perform a build` section. Copy the link where it says `Post to this URL`.



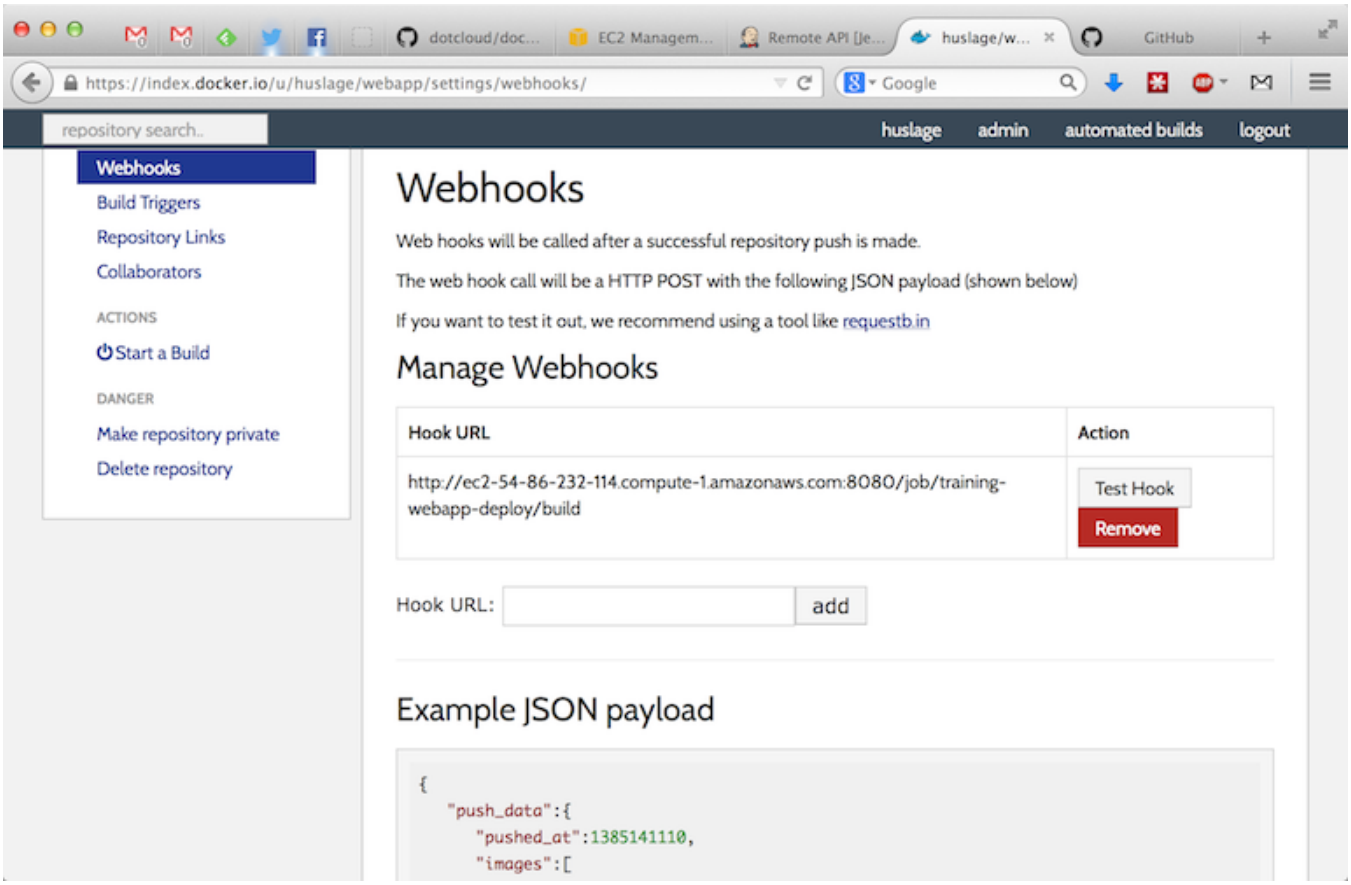
Go back to your `webapp` Docker Hub repository Settings and click `Webhooks` on the left sidebar. Paste the URL you just copied into the `Hook URL` box.



Click `Add`

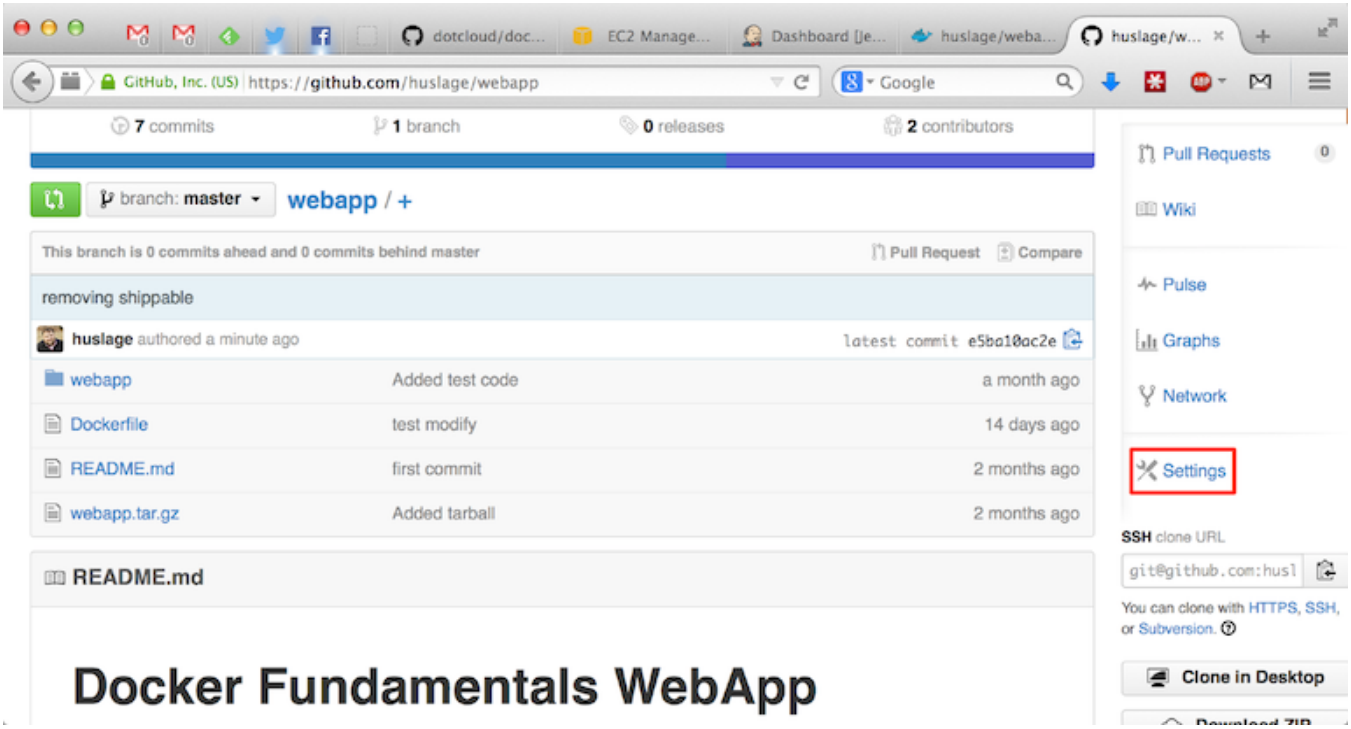
More Success!

This is what your expertise has gained you. Amazing!



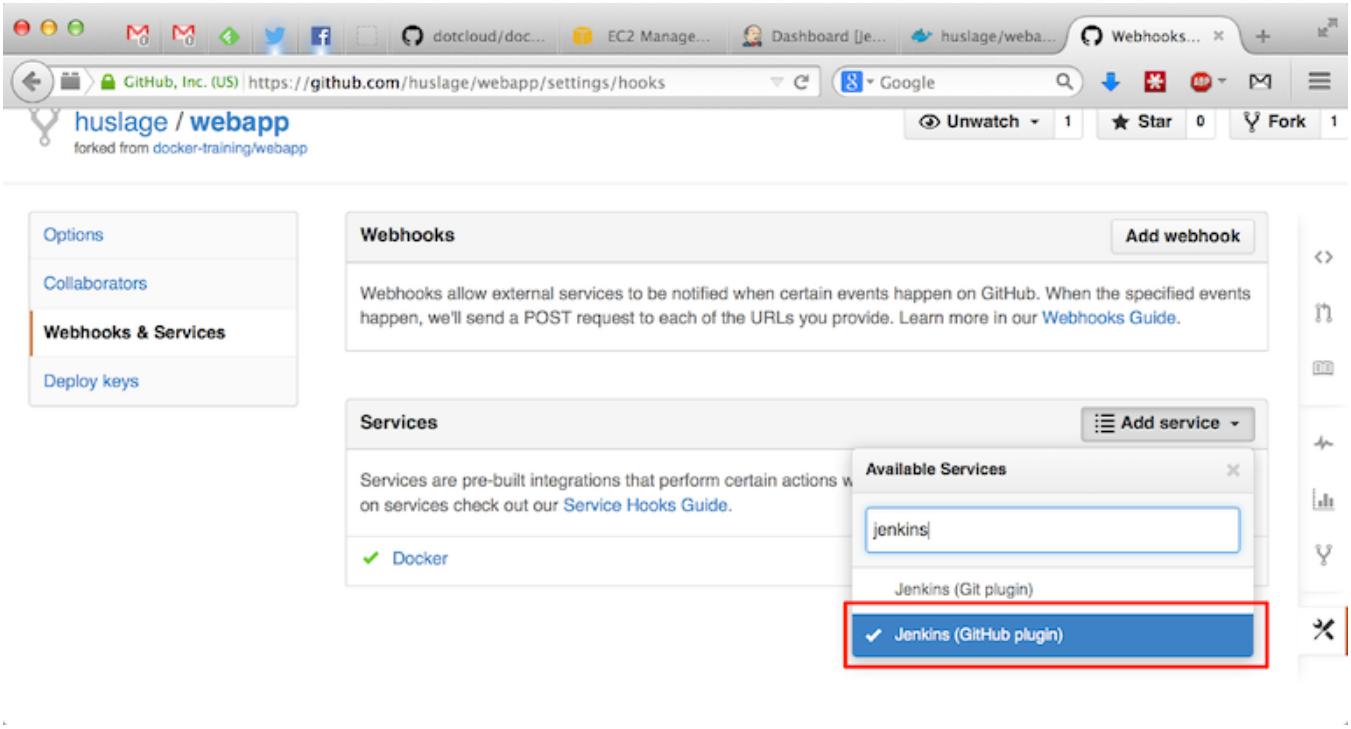
Adding GitHub hook

Now we get to the crux of everything. When we push new code to GitHub, we want to trigger all of this action. Head over to your GitHub fork of the `webapp` and click `Settings` in the right-hand sidebar.



Make Jenkins go!

Click on `Webhooks & Services` in the left-hand sidebar. Click on `Add Service` and type `jenkins` into the search box. Select `Jenkins (GitHub plugin)` from the list.



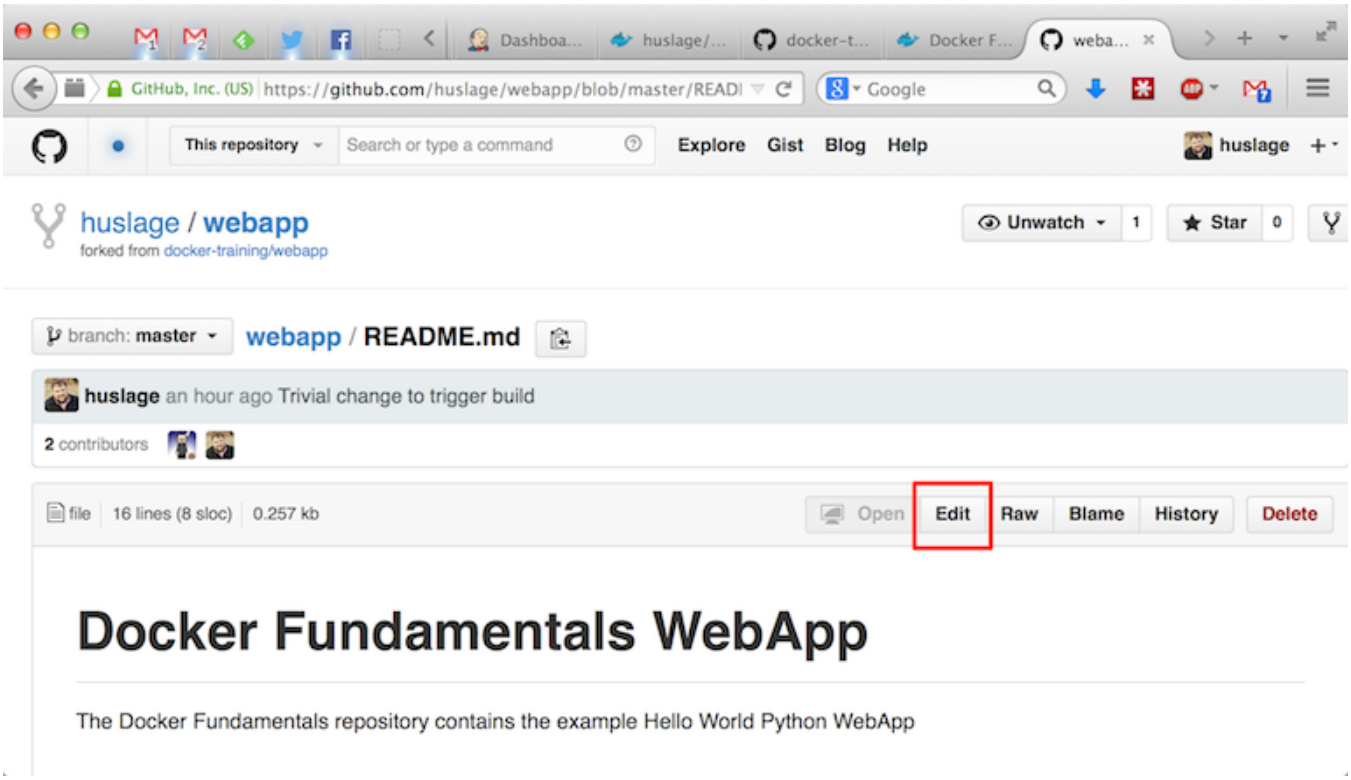
Add that URL

Type the URL of your Jenkins instance into the `Jenkins Hook URL` box. Add `/github-webhook/` to the end of it.

Click `Add Service` at the end.

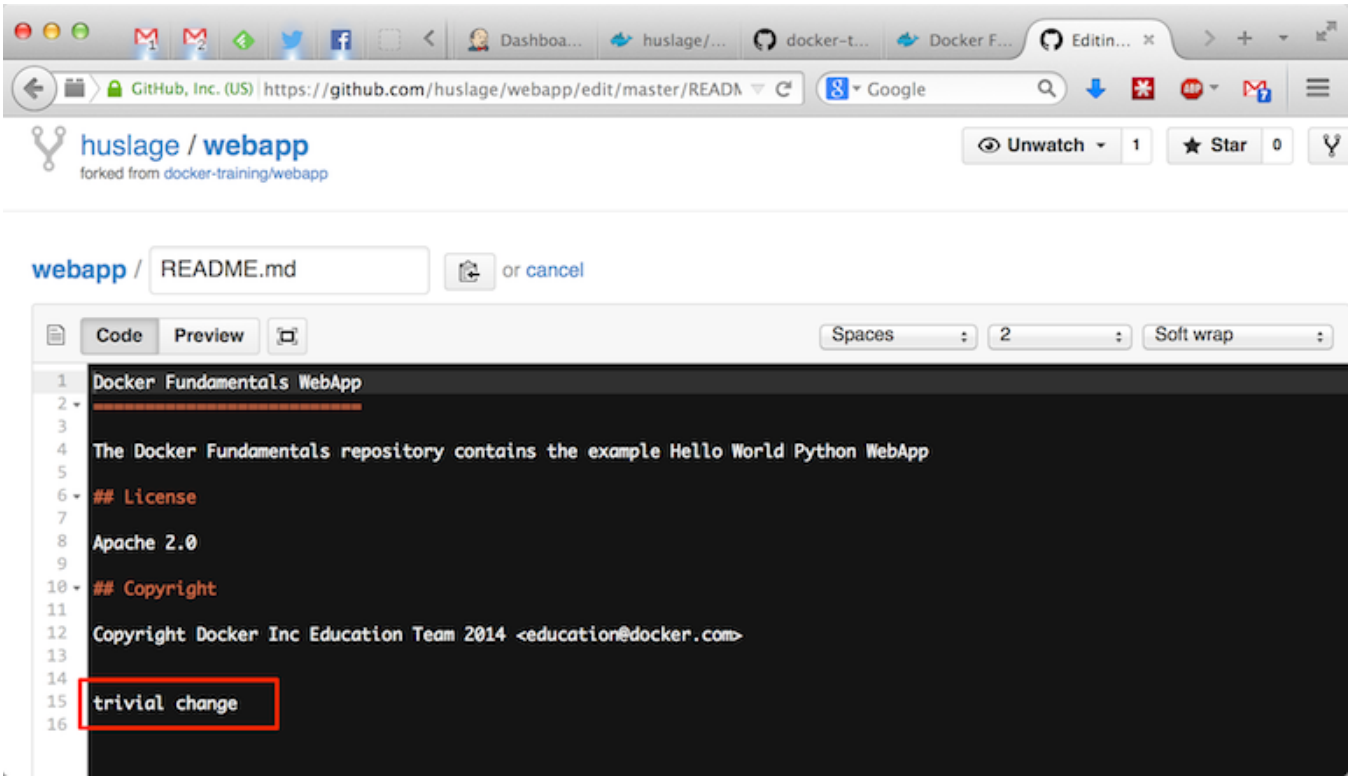
Make some changes

Now we need to make a change to the repository. From the repository home page, click on `README.md`. Then click `Edit` on the top of the next page.



Trivial changes are best

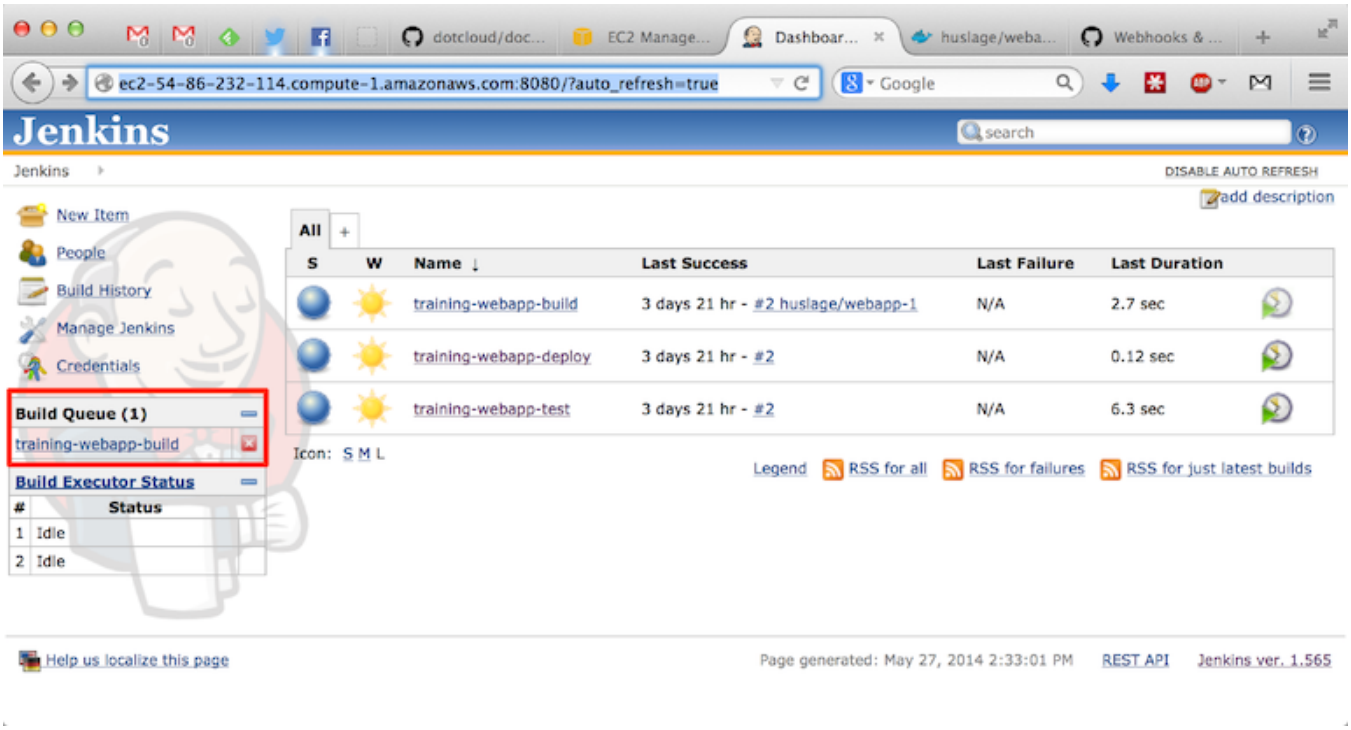
When the edit box comes up, add some text to the bottom of the file. Then scroll to the bottom of the page and click `Commit Changes`.



The Jenkins job will be triggered by this action!

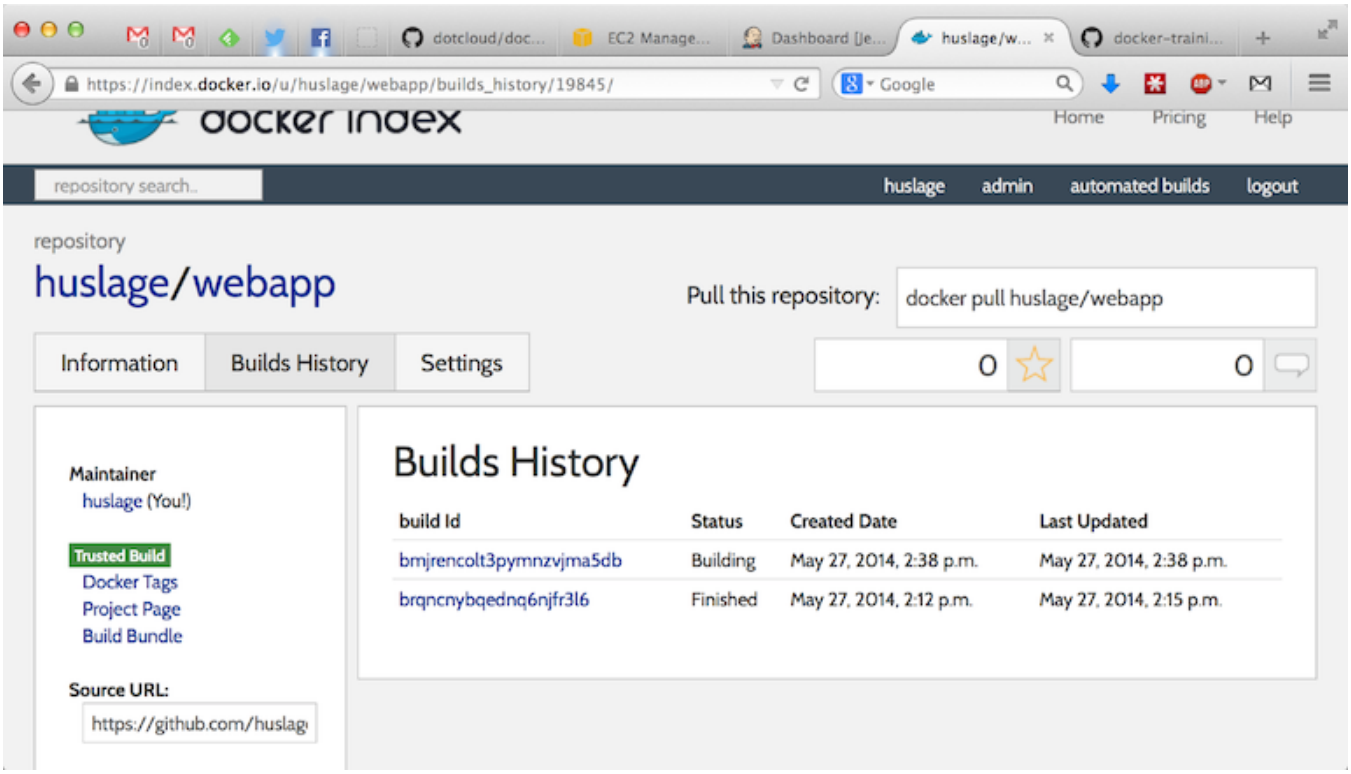
Watch it go

Go back to your Jenkins dashboard. You will see a job running or queued.



Automation in Action

When the `training-webapp-test` job finishes, you can look at the `Build Status` page on your Automated Build. It should build fine.



Section summary

We've learned how to:

- Dockerize a Flask (Python) web application and its tests
- Integrate this Dockerized application with Jenkins for CI
- Use Web Hooks with Automated Builds to automate testing and deployment.

Securing Docker with TLS

Lesson 14: Securing Docker with TLS

Objectives

At the end of this lesson, you will be able to:

- Understand how Docker uses TLS to secure and authorize remote clients
- Create a TLS Certificate Authority
- Create TLS Keys
- Sign TLS Keys
- Use these keys with Docker

Why should I care?

- Docker does not have any access controls on its network API unless you use TLS!

What is TLS

- TLS is Transport Layer Security.
- The protocol that secures websites with `https` URLs.
- Uses Public Key Cryptography to encrypt connections.
- Keys are signed with Certificates which are maintained by a trusted party.
- These Certificates indicate that a trusted party believes the server is who it says it is.
- Each transaction is therefor encrypted *and* authenticated.

How Docker Uses TLS

- Docker provides mechanisms to authenticate both the server the client to *each other*.
- Provides strong authentication, authorization and encryption for any API connection over the network.
- Client keys can be distributed to authorized clients

Environment Preparation

- You need to make sure that OpenSSL version 1.0.1 is installed on your machine.
- Make a directory for all of the files to reside.
- Make sure that the directory is protected and backed up!
- *Treat these files the same as a root password.*

Creating a Certificate Authority

First, initialize the CA serial file and generate CA private and public keys:

```
$ echo 01 > ca.srl
$ openssl genrsa -des3 -out ca-key.pem 2048
$ openssl req -new -x509 -days 365 -key ca-key.pem -out ca.pem
```

We will use the `ca.pem` file to sign all of the other keys later.

Create and Sign the Server Key

Now that we have a CA, we can create a server key and certificate signing request. Make sure that `CN` matches the hostname you run the Docker daemon on:

```
$ openssl genrsa -des3 -out server-key.pem 2048
$ openssl req -subj '/CN=**<Your Hostname Here>**' -new -key server-key.pem -out
server.csr
$ openssl rsa -in server-key.pem -out server-key.pem
```

Next we're going to sign the key with our CA:

```
$ openssl x509 -req -days 365 -in server.csr -CA ca.pem -CAkey ca-key.pem \
  -out server-cert.pem
```


Create and Sign the Client Key

```
$ openssl genrsa -des3 -out client-key.pem 2048
$ openssl req -subj '/CN=client' -new -key client-key.pem -out client.csr
$ openssl rsa -in client-key.pem -out client-key.pem
```

To make the key suitable for client authentication, create a extensions config file:

```
$ echo extendedKeyUsage = clientAuth > extfile.cnf
```

Now sign the key:

```
$ openssl x509 -req -days 365 -in client.csr -CA ca.pem -CAkey ca-key.pem \
  -out client-cert.pem -extfile extfile.cnf
```


Configuring the Docker Daemon for TLS

- By default, Docker does not listen on the network at all.
- To enable remote connections, use the `-H` flag.
- The assigned port for Docker over TLS is `2376`.

```
$ sudo docker -d --tlsverify
--tlscacert=ca.pem --tlscert=server-cert.pem
--tlskey=server-key.pem -H=0.0.0.0:2376
```

Note: You will need to modify the startup scripts on your server for this to be permanent! The keys should be placed in a secure system directory, such as /etc/docker.

Configuring the Docker Client for TLS

If you want to secure your Docker client connections by default, you can move the key files to the `.docker` directory in your home directory. Set the `DOCKER_HOST` variable as well.

```
$ cp ca.pem ~/.docker/ca.pem
$ cp client-cert.pem ~/.docker/cert.pem
$ cp client-key.pem ~/.docker/key.pem
$ export DOCKER_HOST=tcp://:2376
```

Then you can run docker with the `--tlsverify` option.

```
$ docker --tlsverify ps
```


Section Summary

We learned how to:

- Create a TLS Certificate Authority
- Create TLS Keys
- Sign TLS Keys
- Use these keys with Docker

The Docker API

Lesson 15: The Docker API

Objectives

At the end of this lesson, you will be able to:

- Work with the Docker API.
- Create and manage containers with the Docker API.
- Manage images with the Docker API.

Introduction to the Docker API

So far we've used Docker's command line tools to interact with it. Docker also has a fully fledged RESTful API you can work with.

The API allows:

- To build images.
- Run containers.
- Manage containers.

Docker API details

The Docker API is:

- Broadly RESTful with some commands hijacking the HTTP connection for STDIN, STDERR, and STDOUT.
- The API binds locally to `unix:///var/run/docker.sock` but can also be bound to a network interface.
- Not authenticated by default.
- Securable with certificates.

Testing the Docker API

Let's start by using the `info` endpoint to test the Docker API.

```
$ curl --silent -X GET http://localhost:2375/info | python -mjson.tool
{
  "Containers": 27,
  "Debug": 0,
  "Driver": "aufs",
  "DriverStatus": [
    [
        "Root Dir",
        "/var/lib/docker/aufs"
    ],
    [
        "Dirs",
        "144"
    ]
  ],
  "ExecutionDriver": "native-0.1",
  "IPv4Forwarding": 1,
  "Images": 90,
  "IndexServerAddress": "https://index.docker.io/v1/",
  "InitPath": "/usr/bin/docker",
  "KernelVersion": "3.8.0-29-generic",
  "NFd": 19,
  "NGoroutines": 19,
  "SwapLimit": 0
}
```

- This endpoint returns basic information about our Docker host.

Creating a new container via the API

Now let's see how to create a container with the API.

```
$ curl -X POST -H "Content-Type: application/json" \
  http://localhost:2375/containers/create \
  -d '{
    "Hostname":"",
    "Cmd":"[echo hello world]",
    "User":"",
    "Memory":0,
    "MemorySwap":0,
    "AttachStdin":false,
    "AttachStdout":true,
    "AttachStderr":true,
    "PortSpecs":null,
    "Privileged": false,
    "Tty":false,
    "OpenStdin":false,
    "StdinOnce":false,
    "Env":null,
    "Dns":null,
    "Image":"ubuntu",
    "Volumes":{},
    "VolumesFrom":"",
    "WorkingDir":""
  }'
  {"Id":"<yourContainerID>","Warnings":null}
```

- You can see the container ID returned by the API.

Inspecting our launched container

We can also inspect our freshly launched container.

```
$ curl --silent -X POST \
  http://localhost:2375/containers/<yourContainerID>/json
{
  "Args": [],
  "Config": {
    "AttachStderr": true,
    "AttachStdin": false,
    "AttachStdout": true,
    "Cmd": [
        "[echo hello world]"
    ],
    . . .
}
```

- It returns the same hash the `docker inspect` command returns.

Stopping a container

We can also stop a container using the API.

```
$ curl --silent -X POST \
  http://localhost:2375/containers/<yourContainerID>/stop
```

- If it succeeds it will return a HTTP 204 response code.

Working with images

We can also work with Docker images.

```
$ curl -X GET http://localhost:2375/images/json?all=0
[
  {
    "Created": 1396291095,
    "Id": "cccdc2d2ec497e814793e8bd952ae76d5d552c8bb7ed927db54aa65579508ffd",
    "ParentId": "9cd978db300e27386baa9dd791bf6dc818f13e52235b26e95703361ec3c94dc6",
    "RepoTags": [
        "training/datavol:latest"
    ],
    "Size": 0,
    "VirtualSize": 204371253
  },
  {
    "Created": 1396117401,
    "Id": "d4faa2107ddab5b22e815759d9a345f1381562ad44d1d95235347d6b006ec713",
    "ParentId": "439aa219e271671919a52a8d5f7a8e7c2b2950c639f09ce763ac3a06c0d15c22",
    . . .
  }
]
```

- Returns a hash of all images.

Searching the Docker Hub for an image

We can also search the Docker Hub for specific images.

```
$ curl -X GET http://localhost:2375/images/search?term=training
[
  {
    "description": "",
    "is_official": false,
    "is_trusted": true,
    "name": "training/namer",
    "star_count": 0
  },
  {
    "description": "",
    "is_official": false,
    "is_trusted": true,
    "name": "training/postgres",
    "star_count": 0
  }
]
```

This returns a list of images and their metadata.

Creating an image

We can then add one of these images to our Docker host.

```
$ curl -i -v -X POST \
  http://localhost:2375/images/create?fromImage=training/namer
  {"status":"Pulling repository training/namer"}
```

This will pull down the `training/namer` image and add it to our Docker host.

Section summary

We've learned how to:

- Work with the Docker API.
- Create and manage containers with the Docker API.
- Manage images with the Docker API.

Course Conclusion

Course Summary

During this class, we:

- Installed Docker.
- Launched our first container.
- Learned about images.
- Got an understanding about how to manage connectivity and data in Docker containers.
- Learned how to integrate Docker into your daily work flow

Questions & Next Steps

Still Learning:

- Docker homepage
- Docker blog
- Docker documentation
- Docker Getting Started Guide
- Docker code on GitHub
- Docker Forge - collection of Docker tools, utilities and services.
- Docker mailing list
- Docker on IRC: irc.freenode.net and channels `#docker` and `#docker-dev`
- Docker on Twitter
- Get Docker help on StackOverflow

Thank You