Docker Fundamentals For Developers

Introduction

A Gentle Intro to Docker for Developers

Ken Rimple, Director, Training Services, Chariot Solutions ng-conf 2019

Who am I?

- Ken Rimple, Director of Training / Mentoring @ Chariot Solutions
  - http://chariotsolutions.com/training
  - Training in React, Angular, other techs
  - @RimpleOnTech

Resources

- See chariotsolutions.com/ng-conf-2019 for slide deck, zip file of exercises, links
- See me at the Chariot booth this week if you have questions

Agenda

- Explaining Docker
- Launch Pre-Built Containers
- Extending and managing your own Containers with Dockerfiles
- Composing a stack from multiple files
- Volume and port management
- Separating Networks
- Docker for full-stack developers
- Management Tools and Scaling

Explaining Docker

What Docker Itself Is NOT

- A true cross-platform OS-independent virtual machine
- A scaling system (that's what Kubernetes and others are for)
- Trivial to use like a VM

Some terms

- **`Host`** - The actual machine where you're running Docker
- **`Docker Engine`** - The engine that runs on the host you're running Docker on…
- **`Image`** - A runnable piece of softare defined by a…
- **`Dockerfile`** - A file that contains the definition of your Image (GOTO 10)
- **`Container`** - A running instance of an **`Image`**

Workshop Pre-requisites

- A *working* network (testing…)
- You've installed Docker (for OS X, Windows or Linux)
- You're using Linux containers in Windows (otherwise you may not be able to follow along completely

Docker like a Boss on Windows

- Install **`Windows Scripting Host`**
- Install the Ubuntu WSL image from the Windows Store
- Edit your Windows Ubuntu .bashrc file and add aliases for docker, docker-compose, and other windows executables you want to script
- Windows 10 is HIGHLY recommended

Add these to .bashrc and Docker like a boss

```
alias docker=docker.exe
alias docker-compose=docker-compose.exe
```


A really simple Docker test

- Launches the super-small Alpine linux container
- Executes a shell script (just an echo command)
- Shuts down

Open a command line, and type:

```
docker run alpine echo "it worked"
```

Verifying

Execute the Docker **ps** *command to check what happened*

```
$ docker ps -l
```

output

```
CONTAINER ID  IMAGE  COMMAND           CREATED       STATUS                     ...  NAMES
0014d6035286  alpine "echo 'hi there'" 2 minutes ago Exited (0) 2 minutes ago
angry_lumiere
```

What is Alpine Linux?

A super-small Linux container based on **musl libc** and **busybox**. Use Alpine images where possible to save space and skip unnecessary code.

Project home is https://alpinelinux.org/

Docker Containers and Images Crash Course

Remember, Two Different Things

- **`Image`** - The launchable, configured unit of software
- **`Container`** - A running instance of an **`Image`**

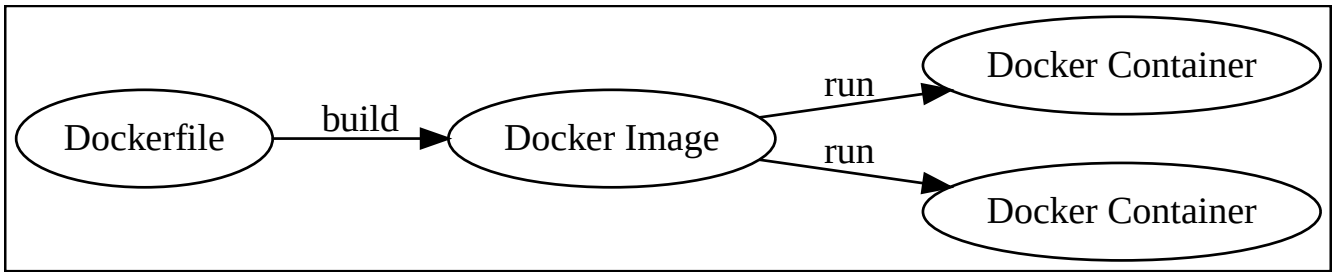

Figure 1. Images are launched as Containers

Containers can be configured with additional information

We'll start by launching pre-built images

Container IDs and names

- Your container has an ID and one more names
- The name is a long hex string, e.g. **`0014d6035286`**
- You can type in the beginning few digits to find it such as **`0014`**

```
$ docker info 0014
```

Remember, *containers* have names and IDs, and *images* have names too

Naming a container

```
$ docker run --name hiya alpine echo "it worked"

# then, to find it
$ docker ps -a -filter "name=hiya"
```


You can't use the same name twice unless you *remove* the older one

Removing a container

Remove containers if named, or to clean them up once done

```
# Remove a named container
$ docker rm hiya

# Remove a container by partial ID
$ docker rm 0fae
```

Only name containers you need to access later

Wha happened to my container?

Look at a container's logs with **`docker logs`**

```
$ docker logs hiya

hi mom
```

Howz about a web server?

Create an HTML file - index.html, then

```
$ cd ex01-http-server/htdocs
$ docker run -p 8080:80 \
  -v "$PWD":/usr/local/apache2/htdocs/ httpd
```

- The **`-v`** flag attaches a *volume* (external directory) to the folder in the container (**`LOCAL:CONTAINER`**)
- Running in the foreground (we'll fix that next).
- Kill with [CTRL-C].

Windows users: **`"$PWD"`** assumes you're running an Ubuntu shell on windows as we recommended before… Otherwise pathing is "fun"…

Run it in the background

*Just add the **-d** flag to your launch process (and name with **--name**)*

```
$ cd ex01-http-server/htdocs
$ docker run --name=myapache -d -p 8080:80 \
  -v "$PWD":/usr/local/apache2/htdocs/ httpd
```

If you don't name it as above, the container will just have its hex ID

Stopping, starting, killing containers

Stopping (pausing) a container (sends SIG)

```
$ docker stop myapache
```

Re-starting a stopped container

```
$ docker start myapache
```

Terminating a container (sends SIGKILL)

```
$ docker kill myapache

# Sending alternate kill signals

$ docker kill -15 myjavaserver
```

You can't remove a container until you kill the instances

What stays around after stopping/killing

- All contents of the container (directories, files)
- Log history
- All is destroyed when the container is removed

Removing a container destroys its data

```
docker rm myapache
```


You have to stop a container before removing it with **`stop`** or the more insistant **`kill`**

Exploring the Apache httpd Image

- A container has a **`Dockerfile`** that defines it
- We can view the **`httpd`** Dockerfile on the Docker Project GitHub

A portion of the Dockerfile for **`httpd`**

```
FROM debian:stretch-slim

... much patching, configuring, etc. ...

COPY httpd-foreground /usr/local/bin/

EXPOSE 80
CMD ["httpd-foreground"]
```

Docker images usually (but not always) have a **`CMD`** or **`ENTRYPOINT`** that kick off when they start

Restarting a container

Restarting a container stops and starts it again

```
docker restart myapache
```

Docker Images

Getting Public Docker Images

- Use **`docker search`** to find an image or search on **`docker.io`**

The AsciiDoctor image is really useful

```
$ docker search asciidoctor
NAME
asciidoctor/docker-asciidoctor
```

Using a Docker Image as a Command Line Tool

Launching the ASCIIDoctor container to convert adoc to html

```
docker run \
  -v "$PWD":/documents/  \
  asciidoctor/docker-asciidoctor asciidoctor source.adoc
```

Produces **source.html** in the current directory and quits immediately.

Creating your own image

- You can build your own Docker images
- Images are based on other images
- You customize your new image with a **Dockerfile**
- You then build it with **docker build** (next slide)

Make your own Apache server

```
FROM httpd:alpine                       ①

COPY htdocs/ /usr/local/apache2/htdocs  ②
```

① The image to pull and extend

② Makes a copy of the files in your local folder **htdocs** and places in the Docker container as **/usr/local/apache2/hdtocs**

The **COPY** command is very finicky. Expect to get it wrong a lot at first. Trust, but verify.

Building a Dockerfile into an image

Use **docker build <dirname>**

Build the Docker file in your current folder

```
$ docker build .

Sending build context to Docker daemon   7.68kB
Step 1/2 : from httpd:alpine
 ---> 1d42e93b142c
Step 2/2 : COPY htdocs/ /usr/local/apache2/htdocs
 ---> 18d3646c191b
Successfully built 18d3646c191b
```

Name Those Images…

- Highly recommended to name your image

```
docker build -n myapache-image .
```

- If the dockerfile is not called **`Dockerfile`**, point to it with **`-f filename`**

```
docker build -f Dockerfile.myapache \
   -n myapache-image .
```

Always try to name your images if you manage them later

Now you can spin them up by name

```
docker run \
  --name=myapache-container
  -d myapache-image
..

docker kill myapache-container
docker image rm myapache-image

# alternate syntax
docker rmi myapache-image
```

Name your images and containers for easy scripting and management

Dockerfile in-depth

How does the Dockerfile configure your image?

- Lets you manage, create directories in the image
- Lets you **`COPY`** files from outside of the image into the image
- Lets you install updates to the software the image is based on
- Lets you expose ports and map them to ports on your OS
- Lets you define a starting command (**`CMD`**) or entrypoint script (**`ENTRYPOINT`**) for execution on startup

Dockerfile: Managing Directories and Files in an Image

Use the **`COPY`** *command to copy files*

```
# Copy everything from the current host directory
# to the current image directory

COPY . .

# Copy everything from the local "htdocs" directory
# into the container's "/usr/local/apache2/htdocs"
# directory.

WORKDIR /usr/local/apache2/htdocs
COPY htdocs/ ./
```

COPY command gotchas

- The **`COPY`** command doesn't always do recursive file copying.
- You must append a trailing slash to a target if you're copying a directory from the source
- Using **`COPY dir/* .`** *flattens* the files in the target
- You may end up writing some rather verbose copy statements to copy subdirectories, then the parent directory, then individual files

Show the sample from my app

Dockerfile: Running Scripts

Use the **RUN** *command*

```
RUN mkdir -p /usr/local/server/http

RUN /usr/local/bin/generate-something
```

Each command in a Dockerfile creates an intermediate image if it changes the filesystem. The final image is kept.

Configuring the container's startup actions

- **ENTRYPOINT** - command and parameters that run after the container starts
- **CMD** - used to provide defaults for executing a container

Configuring the Entrypoint

Entrypoints execute when the container starts

```
 ..
COPY ./entrypoint.sh /usr/local/bin/

WORKDIR /usr/local/bin

ENTRYPOINT ["./entrypoint.sh"]
```

Build a shell script if you want full control

Example: Pandoc

- Pandoc converts files from one format to another (HTML to PDF, MARKDOWN TO ASCIIDOC, etc)
- It can be difficult to configure natively
- So we can install it in a Docker linux container with its dependencies

```
FROM 'debian:buster-slim'
RUN mkdir /documents
RUN apt update -y && apt install -y pandoc texlive
WORKDIR /documents
ENTRYPOINT ["/usr/bin/pandoc"]
CMD ["--help"]
```

This is **ex02-pandoc** in your labs

To Run the Pandoc Container

```
docker run -it -v"$PWD":/documents/ \
    pandoc \
    *.markdown \
    -o output.pdf
```

- This example runs the **pandoc** command line tool and mounts the current directory as /documents/ in the container
- It converts all markdown files into a single PDF, **output.pdf**
- If you pass the container **--help** or leave out any command, it will print the **pandoc** help entry

Volumes allow for extra-container persistence

- Use a volume to
  - Bind a local host directory to a directory in the container for direct access to the files (database files, app files for dev reload)
  - Access a logical named volume at runtime without mapping directly to a file system (named volume, typically used in the **docker-compose** utility instead of Docker itself)
  - Map a temporary directory for writing data faster than writing to a container

Mapping a host directory revisited

Map the local directory htdocs to a server directory

```
# Map the local directory ./htdocs
# to the server on the container
-v "$PWD/htdocs":/usr/local/apache2/htdocs
```


Use the newer --mount flag (preferred)

```
--mount type=bind,source=htdocs,target=/usr/local/apache2/htdocs
```

-v will always create the directory if it doesn't exist, but **--mount** will not create it.

Characteristics of Bind Mounts

- Very fast (because they are native to the Host OS)
- Host-specific (so they aren't 100% cross-platform for pathing, etc)
- Not easy to backup

Named Volumes

- Managed with **docker volume** or in **docker-compose**
- Creates a volume on the Host FS (or virtual machine in the case of OS X or Windows)

Creating with **docker volume**

```
docker volume create dbfiles
```

Stacks and Composition

Composing a Stack of Docker Images

- Enterprise applications are comprised of a set of related services
- Normally developers stay away from tiered systems and run everything "locally"
- Docker tries to make it easy for developers to run closer to the production architecture

Example - Wordpress Deployment

Figure 2. Wordpress needs at least two servers

We'll use Docker Images for each server

New Tool: docker-compose

- **`docker-compose`** launches a set of related docker containers
  - The **`docker-compose.yml`** file lists the services and relates them together
  - Images can be pulled from public definitions or built with **`Dockerfiles`**
  - The **`docker-compose`** command launches all services together or can launch a subset based on their names

A docker-compose.yml file sample for Wordpress

Skeleton of compose file

```
version: '3.7'

services:
  wordpress:
    image: wordpress
    ...
  db:
    image: mysql:5.7
    ...
```

The `wordpress` container definition

Wordpress has a prefab image ready to go

```
wordpress:
    image: wordpress
    restart: always
    ports:
      - "8080:80"
    environment:
      WORDPRESS_DB_HOST: db
      WORDPRESS_DB_USER: exampleuser
      WORDPRESS_DB_PASSWORD: examplepass
      WORDPRESS_DB_NAME: exampledb
```

Options used

- **`image: wordpress`** - pulls the latest wordpress Docker image
- **`restart: always`** - restarts whenever the container fails for any reason
- **`ports`** - exposes ports (such as port **`80`**) from the container in the host (**`8080`**)
- **`environment`** - sends variables into the docker container where they are used to pass parameters to the Wordpress engine

The `db` container definition

*The container name **db** exposes it as that name on the virtual network*

```
db:
    image: mysql:5.7
    restart: always
    environment:
      MYSQL_DATABASE: exampledb
      MYSQL_USER: exampleuser
      MYSQL_PASSWORD: examplepass
      MYSQL_RANDOM_ROOT_PASSWORD: '1'
```

Managing the stack

- **`docker-compose up`** - boot up the stack
- **`docker-compose down`** - tear down the stack
- **`docker-compose start`** - start up the stack (if stopped)
- **`docker-compose stop`** - stop the stack (if running)

Kicking off the Wordpress Docker Stack

```
docker-compose up -d
```

Using **`-d`** detaches the process so it runs in the background

Getting the container status

Containers are named after the project (dir by default) and container base name

```
$ docker-compose ps

           Name                         Command              State          Ports

ex04-wordpress_db_1          docker-entrypoint.sh mysqld     Up      3306/tcp, 33060/tcp
ex04-wordpress_wordpress_1   docker-entrypoint.sh apach      Up      0.0.0.0:8080->80/tcp
```

Advanced users can spin up a swarm / cluster of nodes and so they are automatically named with a instance number as an appended value (etc...)

Watching logs in the stack

Use the **logs** *command*

```
docker-compose logs

# watch them
docker-compose logs -f

# watch for a certain container
docker-compose logs -f db
```

Special Features

Specifying container dependencies

- Each container can depend on one or more other containers in the stack
- Use the **depends_on** property

This is an array in YAML

```
services:

  wordpress:
    image: wordpress
    restart: always
    depends_on:
      - db
  ...

  db:
    ...
```

Doesn't wait for *logical* startup, just that **db** is launched!

Handling ports and connectivity

Exposing to other containers

use **expose** *to make port available to other containers*

```
# Expose port 80 of the machine 's Virtual IP
  wordpress:
    expose:
      - "80"
      - "443"
```

This does not expose them to the **host**

Creating Networks

- By default there is one virtual network for the whole stack
- Docker containers can live in different virtual networks
- Networks are firewalled off of each other
- Containers can be attached to one or more networks to form a full network stack too

Creating networks - example

Two networks in a container

```
# at the bottom of the compose file
networks:
  - frontend
  - backend
```

This is a powerful feature and keeps you from accidentally exposing the whole stack

Assiging Networks

Connect the containers to networks

```
web:
  networks:
    - frontend
appserver:
  networks:
    - frontend
    - backend
db:
   networks:
      -backend
```

This is a LOT easier than working with physical servers and firewall rules

Exposing a service to the host

Use the **`ports`** *property*

```
wordpress:
  ports:
     - "22"
     - "8080:80”
     - “8443:443”
```

YAML parses any numeric lower than 60 as "Base 60" so quote all ports

Volumes

- Managing volumes is much easier in **`docker-compose`**
- Volumes are contextual to the use of the image
- You can create linked volumes and named volumes easily

Linked Volumes

- These are "punches" outside to your filesystem
- They should be relative to your compose file
- Great for externalizing log files, source code

Linked volume examples

```
myApp:
  volumes:
    - ./source:/usr/app/source
    - ../config/certs:/app/certs
```

Named Volumes

- Named volumes are persistent, external filesystems
- They are MUCH faster to write to than docker containers
- But they are handled by docker

Setting up a named volume for a database

```
db:
  container: mysql:5.7
  volumes:
    - data

volumes:
  - data
```

These are created if they don't yet exist

Environment and Config

- There are several types of variables in Docker
  - **`ARG`** - a variable defined during the build stage of a **`Dockerfile`**
  - **`ENV`** - a variable from the running "environment" of the host

ARG variables are used by Dockerfiles primarily and are not available in the running container

Why ARGs?

- Bake in labels during builds (visible with docker inspect)
- Assign specific but configurable versions of software when installing with **`apt-get`**
- Example

Setting ARGs (DON'T PUT PASSWORDS HERE)

```
# in Dockerfile (enables ARG)
ARG BUILDVERSION
LABEL buildVersion=$BUILDVERSION

# in compose
myApp:
  build:
    context: myApp
    args:
     buildVersion: 1023
```

Why ENVs?

- Assign database passwords to container when launching
- Set specific flags in running containers rather than for the image (DEVELOPMENT, PRODUCTION)

Do not check .env files into your developer version control system

Wrapup

- Be sure to visit https://chariotsolutions.com/ngconf-docker` for links to resources
- Also stop by at our booth at ng-conf to say 'hi' and discuss Docker or Angular in more detail
- Contact me at **krimple@chariotsolutions.com** or on Twitter at **@RimpleOnTech**
- Looking for Angular or React training? Check us out at https://chariotsolutions.com/training
- Our consultants blog and speak about many topics across the full stack. Check out our blog at https://chariotsolutions.com/blog for more information.
- I've been running a podcast, the Chariot TechCast, for about 10 years now. Interviews with open source luminaries as well as developers in the field are available at https://chariotsolutions.com/techcast or on iTunes or Google as "Chariot TechCast".
- We also run a conference every year, Philly Emerging Technologies for the Enterprise, and our materials are coming online soon on our YouTube page. Look for the Philly ETE 2019 playlist for details at: https://www.youtube.com/user/ChariotSolutions/playlists

Most of all, ENJOY ng-conf 2019!!!