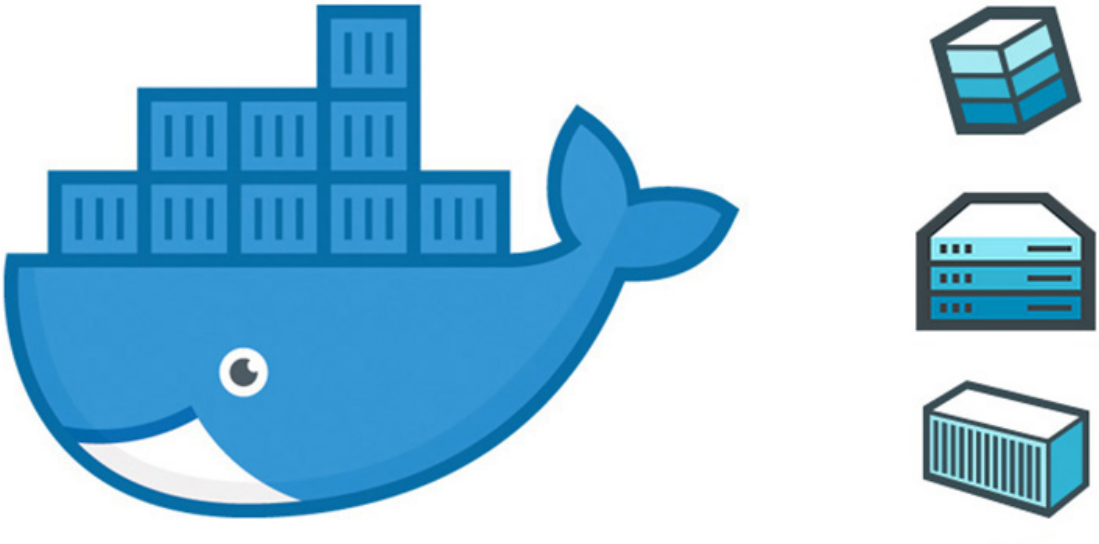
Franklin Smith

Automating Dev Ops with Docker Automation Script



Docker is a platform for engineers, developers, and sysadmins to create, develop, run, and deploy applications with containers. Using Linux containers to launch applications is called containerization. Containers and growing in popularity due to the fact that they make it possible to get more apps running on the same old servers and its easy to ship and package programs.

Within this automated shell script/tutorial, you will have:
Set up your docker environment, build an image and run as one container, scale your app to run multiple containers, distribute your app across a cluster, stack your services by adding a backend database, and deploy your app to production.

Part 1 Preparing Docker Environment

Step1.

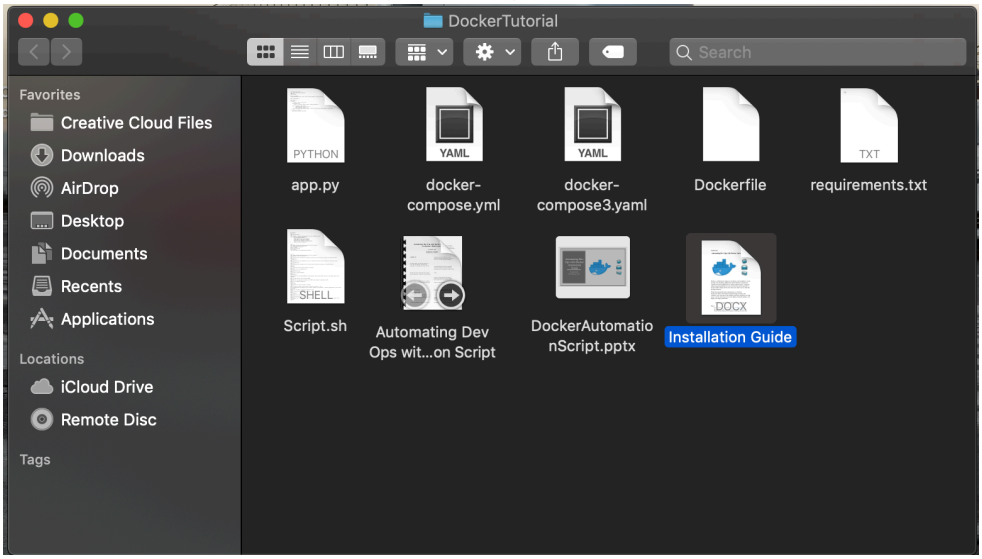
Create an account on Docker Hub https://hub.docker.com/

Install Docker https://docs.docker.com/engine/installation/

Verify Docker Daemon is running, by opening on your local machine and running the whale application

Download the necessary required files and folder from https://github.com/fsmith503/ComputerScience/tree/master/FranklinSmithDockerTutorialScript

Below are the example folder and files you should see in the folder after you clone the respository from github or download the foler.



Open the “Terminal” Application on your computer

Change directory into the folder “DockerTutorial”

If you are unfamillilar with terminal commands, or how to do this, you can google it.

On my local machine I execute this step by doing:

$ cd Desktop

$ cd DockerTutorial

At this point you can execute the shell script by the following:
$ sh Script.sh

At this point you can execute the commands for the tutorial yourself by follow all of the instructions below. This is an exclusive step and the user should only choose one **OR** the other in the tutorial process. The shell script execution **OR** the commands.

Step2.

$ docker --version

This verifies a supported version of docker

Step3.

$ docker run hello-world

Verifies that the installation works by running the simple docker image, hello-world

Step4.

$ docker image ls

Verifies that the image was downloaded to our machine

Conclusion.

Containers make continuous integration and continuous delivery seamless

- Applications have no system dependencies
- Updates can be pushed to nay part of a distributed application
- Resource density can be optimized
- With docker scaling your application is a matter of spinning up new executables, not running heavy VM hosts.

Part 2 Getting Started With Containers

Step1.

Verify that the docker file Included in tutorial exists and correctly contains the following contents

```
# Use an official Python runtime as a parent image
FROM python:2.7-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

Verify that the requirements.txt file in tutorial exists and correctly contains the following contents

```
Flask
Redis
```

Verify that the app.py file in tutorial exists and correctly contains the following contents

```
from flask import Flask
from redis import Redis, RedisError
import os
import socket

# Connect to Redis
redis = Redis(host="redis", db=0, socket_connect_timeout=2, socket_timeout=2)

app = Flask(__name__)

@app.route("/")
def hello():
    try:
        visits = redis.incr("counter")
    except RedisError:
        visits = "<i>cannot connect to Redis, counter disabled</i>"

    html = "<h3>Hello {name}!</h3>" \
           "<b>Hostname:</b> {hostname}<br/>" \
           "<b>Visits:</b> {visits}"
    return html.format(name=os.getenv("NAME", "world"), hostname=socket.gethostname(), visits=visits)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```

Step2.

$ pip install -r requirements.txt

Installs flask and redis libraries for python

$ ls

Step3.

$ docker build –tag=friendly hello.

$ docker image ls

$ docker run -p 4000:80 friendlyhello

Open web browser or chrome to localhost:4000

Step4.

$ docker login

Login with your docker id, if no id then sign up at hub.docker.com

$ docker tag image username/repository:tag

$ docker image ls

$ docker push username/repository:tag

$docker run -p 4000:80 username/repository:tag

Conclusion.

In the last part you built a python app using a python runtime on your local machine. But that's not good because your machine needs to be in perfect state to run your app perfectly and also match your production environment. With docker you can just grab a portable python runtime as an image, without any install. Thus, your build can include the base python image along your app code, ensuring that the app, its dependencies, and runtime all travel together.

Part 3 Getting Started With Services

Step1.

Verify that the docker-compose.yml file in tutorial exists and correctly contains the following contents

```yaml
version: "3"
services:
  web:
    # replace username/repo:tag with your name and image details
    image: username/repo:tag
    deploy:
      replicas: 5
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
      restart_policy:
        condition: on-failure
    ports:
      - "4000:80"
    networks:
      - webnet
networks:
  webnet:
```

Step2.

$ docker swarm leave –force

Clears out old swarm from previous exercise

$ docker swarm init

Iniitalizes new swarm

$ docker stack deploy -c docker-compose.yml $appname

This command scales the app, you can scale more by changing the replicas value in docker-compose.yml

$ docker service ls

Gets the service ID for your application

$ docker stack services $appname

Views all services associated with your app

$ docker service ps $appname

Lists all of the tasks for the service

$ docker container ls -q

Lists all of the containers on the system

Step3.

$ docker stack ps $appname

Views all tasks of your stack

$ docker stack deploy -c docker-compose.yml $appname

Scales up the app

$ docker stack rm $appname

Takes down the app

$ docker swarm leave – force

Takes down the swarm

$ docker tag $appname $username/$appname:tag

Tags the image

$ docker push $username/$appname/tag

Conclusion.

The above steps allow the user to create and scale your app with docker. This is closer to learning how to run the containers in production.

Part 4 Getting started with swarms

Step1.

Ensure steps 1-3 are completed correctly.

Step2.

$ docker-machine create --driver virtualbox myvm1

$ docker-machine create –driver virtualbox myvm2

These commands create a couple of VMs using docker-machine

Step3.

$ docker-machine ls

List your VM machines and gets their IP addresses

Enter IP address of vm1

Enter token into terminal

Step4.

Built in script command initializes swarm and makes first VM the mangager.

Sends command to vm using docker-machine ssh, instructs myvm1 to become a swarm mangager with docker swarm init

Step5.

Built in script command to second VM via ssh has the second VM join the new swarm as a worker.

Now you have created your first swarm

$ docker-machine ls

Lists the nodes and machines in the swarm

Step6.

Built in docker command that joins the second VM to the swarm

$ docker-machine ssh myvm1 “docker node ls”

Views the nodes in the swarm

Step7.

$ docker-machine env myvm1

Configuring the shell to talk to the first vm

$ eval $(docker-machine env myvm1)

Configures the shell to talk to the first vm

$ docker-machine ls

Lists machines

Verifys that myvm1 is now the active machine with asterisk in active column

Step8.

$ docker stack deploy -c docker-copost.yml $appName

Now the app is deployed on a swarm cluster

$ docker stack ps $appName

Listing services

Step9.

$ eval $(docker-machine env -u)

Unsets the docker-machine environment variables in current shell

Step10.

$ docker-machine ls

Checks statuses of machines

Two webbrowsers open with the two addresses of the VMS.

Webpages should be working upon refresh

Conclusion.

Now you have successfully learned about swarms, how nodes in them can be managers or workers, created a swarm, and deployed an application on it.

Part 5 Getting started with Stacks

Part1.

$ docker-machine ls

Verifying machines you set up in part 4 are running and ready

Part2.

$docker swarm init

$ docker stack deploy -c docker-compose.yml friendlyhello

Part3.

$ docker-machine ls

Try to open visualizer

Part4.

$ docker stack ps friendlyhello

Checking the stack

Part5.

$ docker-machine ssh myvm1 mkdir ./data

Create data directory on the manager

Part6.

$ docker stack deploy -c docker-compose.yml friendlyhello

Part7.

$docker service ls

Verify three services are running as expected

Conclusion.

In this part of the tutorial you learned that stacks are inter-connected services that run together. To add more services to your stack you insert them in your compose file. You used directories to make a place for storing your data. This allows the application to survive then the container is down or redeployed.

Part 6 Deploying Your App

Step1.

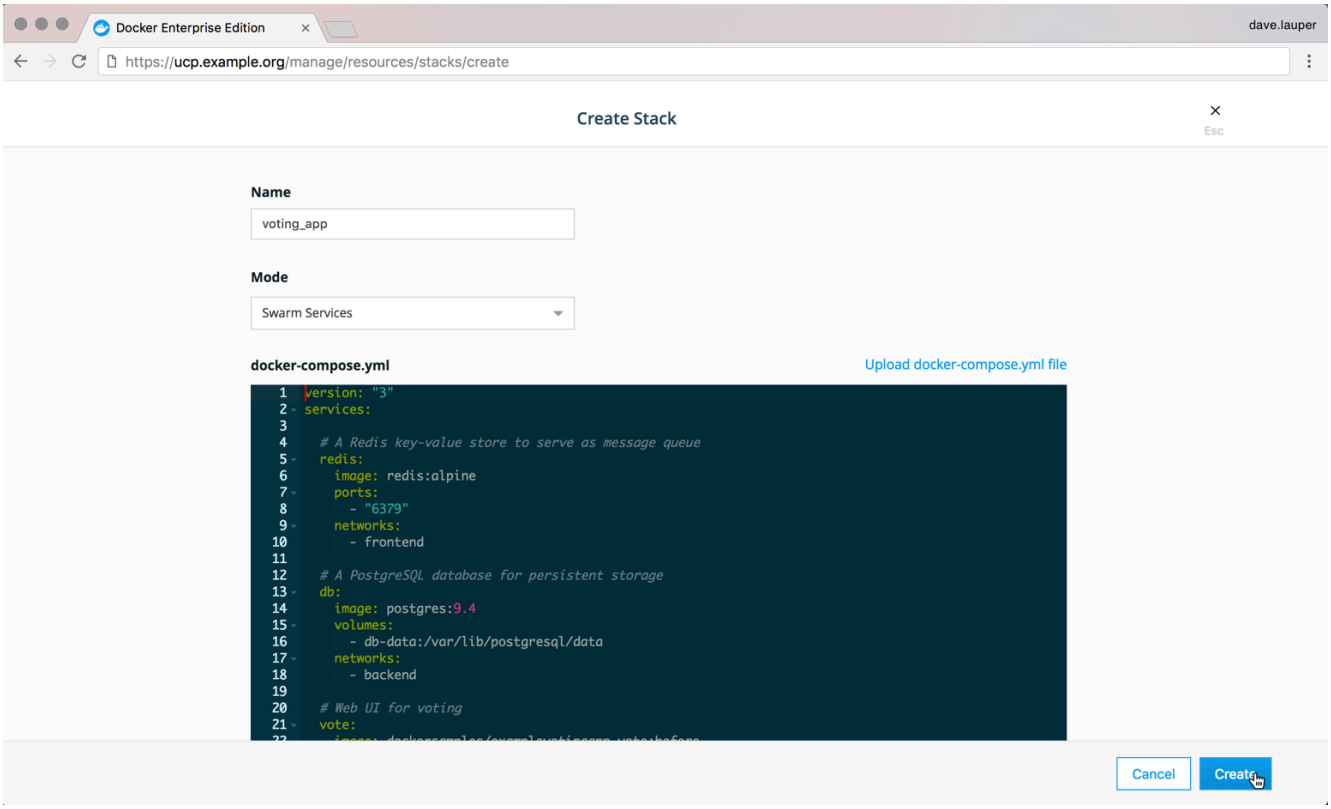
Get Docker Enterprise for your servers OS from Docker Hub

https://hub.docker.com/search?offering=enterprise&type=edition

Follow the instructions to install Docker Enterprise on your own host.
https://docs.docker.com/datacenter/install/linux/

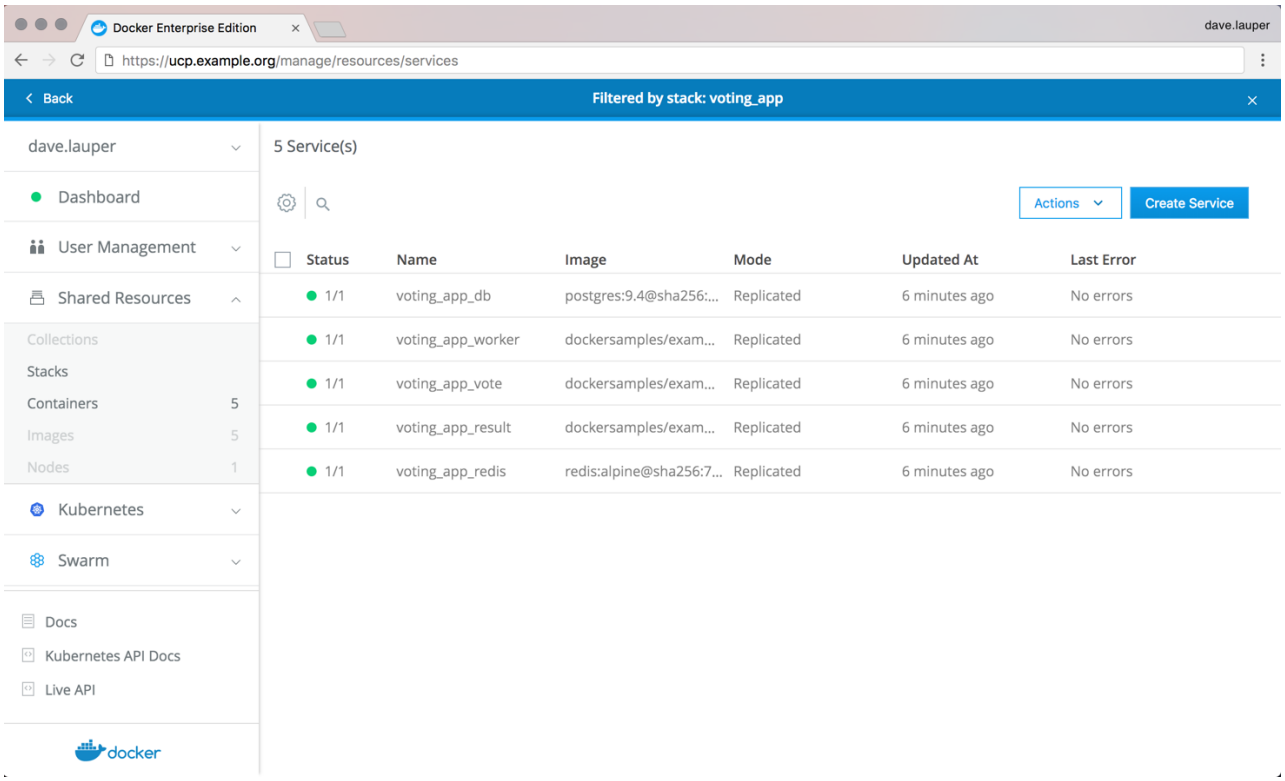
Step2.

Now you have completed step1 and you have Docker Enterprise is running, its time to deploy your Compose file from directly within the UI.



Step3.

After its running, you can modify anything about your application that you want, including editing the compose file.



Docker Tutorial Finished

Now you have accomplished a full-stack, development to depoloyment tutorial of the entire Docker platform application. Between the tutorial, command line, and shell script you can execute and read which gives you a full understanding of containers, images, services, swarms, stacks and scaling.