

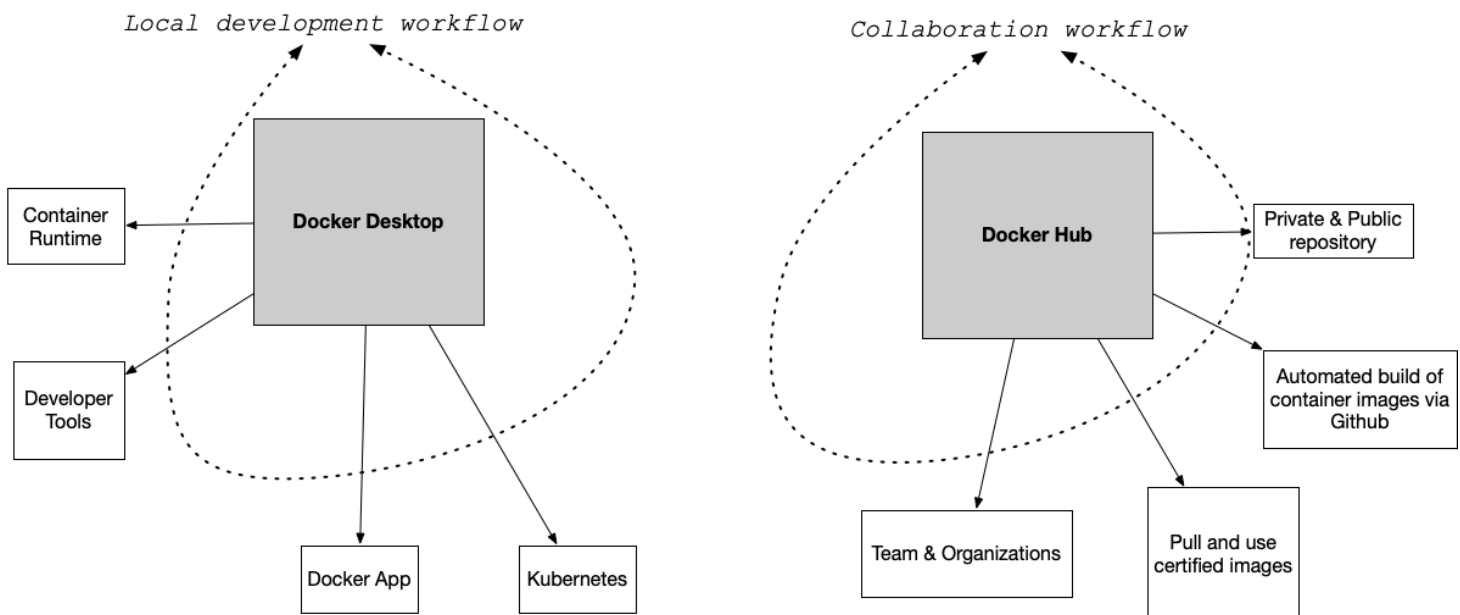
Containers: Docker

Containers: Docker

Getting started with Docker

There are two main components of Docker: [Docker Desktop](#) and [Docker Hub](#).

Docker Product Ecosystem



Docker Desktop Overview

The desktop application contains the container runtime which allows containers to execute. The Docker App itself orchestrates the local development workflow including the ability to use [Kubernetes](#), which is an open-source system for managing containerized applications that came out of Google.

Docker Hub Overview

So what is Docker Hub and what problem does it solve? Just as the [git](#) source code ecosystem has local developer tools like [vim](#), [emacs](#), [Visual Studio Code](#) or [XCode](#) that work with it, Docker Desktop works with Docker containers and allows for local use and development.

When collaborating with [git](#) outside of the local environment, developers often use platforms like [Github](#) or [Gitlab](#) to communicate with other parties and share code. [Docker Hub](#) works in a similar way. Docker Hub allows developers to share docker containers that can serve as a base image for building new solutions.

These base images can be built by experts and certified to be high quality: i.e. the [official Python developers have a base image](#). This allows a developer to leverage the expertise of the true expert on a particular software component and improve the overall quality of their container. This is a similar concept to using a library developed by another developer versus writing it yourself.

Why Docker Containers vs Virtual Machines?

What is the difference between a container and a virtual machine? Here is a breakdown:

- **Size:** Containers are much smaller than Virtual Machines (VM) and run as isolated processes versus virtualized hardware. VMs can be GBs while containers can be MBs.
- **Speed:** Virtual Machines can be slow to boot and take minutes to launch. A container can spawn much more quickly typically in seconds.
- **Composability:** Containers are designed to be programmatically built and are defined as source code in an Infrastructure as Code project (IaC). Virtual Machines are often replicas of a manually built system. Containers make IaC workflows possible because they are defined as a file and checked into source control alongside the project's source code.

Real-World Examples of Containers

What problem do [Docker format containers](#) solve? In a nutshell, the operating system runtime can be packaged along with the code, and this solves a particularly complicated problem with a long history. There is a famous meme that goes "It works on my machine!". While this is often told as a joke to illustrate the complexity of deploying software, it is also true. Containers solve this exact problem. If the code works in a container, then the container configuration can be checked in as code. Another way to describe this concept is that the actual Infrastructure is treated as code. This is called IaC (Infrastructure as Code).

Here are a few specific examples:

Developer Shares Local Project

A developer can work on a web application that uses `flask` (a popular Python web framework). The installation and configuration of the underlying operating system is handled by the Docker container file. Another team member can checkout the code and use `docker run` to run the project. This eliminates what could be a multi-day problem of configuring a laptop correctly to run a software project.

Data Scientist shares Jupyter Notebook with a Researcher at another University

A data scientist working with [jupyter](#) style notebooks wants to share a complex data science project that has multiple dependencies on C, Fortran, R, and Python code. They package up the runtime as a Docker container and eliminate the back and forth over several weeks that occurs when sharing a project like this.

A Machine Learning Engineer Load Tests a Production Machine Learning Model

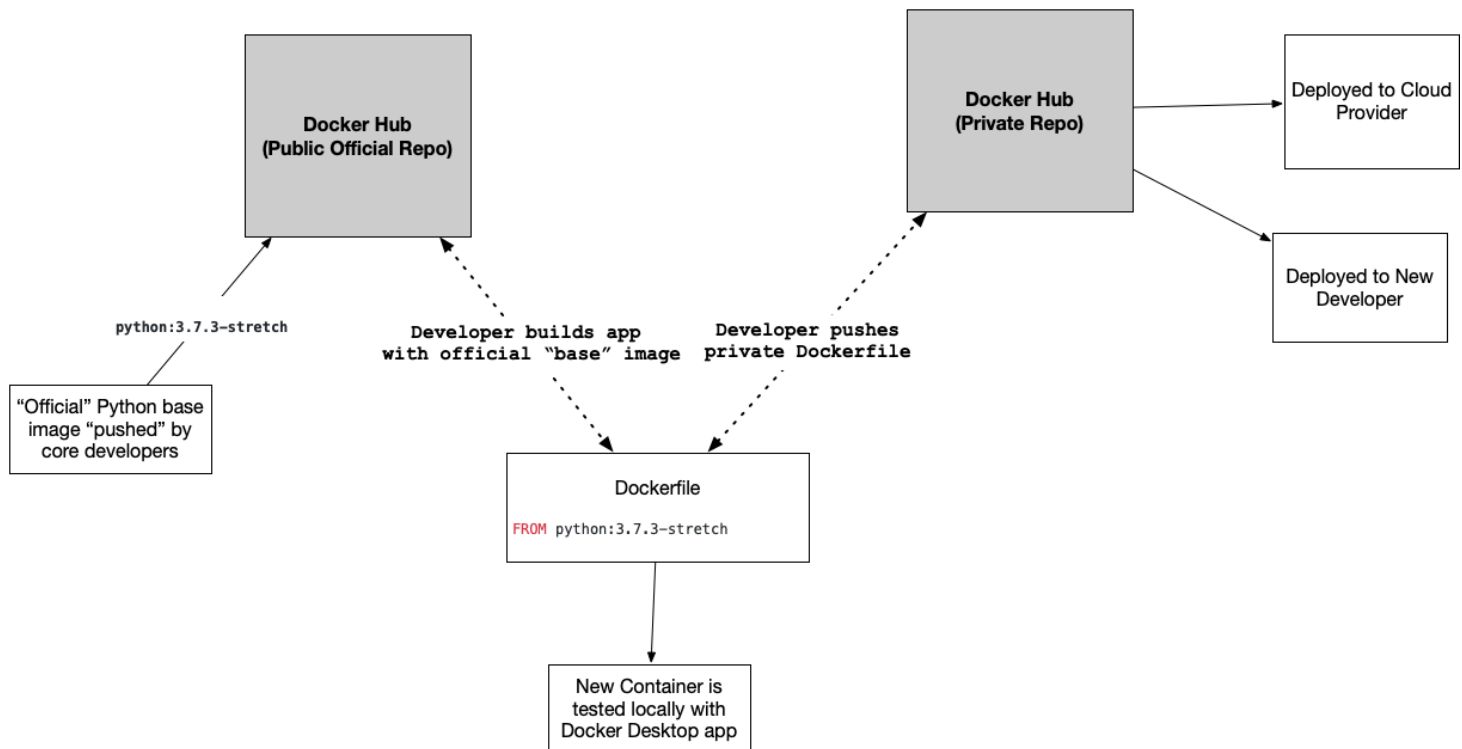
A Machine learning engineer has been tasked with taking a new model and deploying it to production. Previously, they were concerned about how to accurately test the accuracy of the new model before committing to it. The model recommends products to paying customers and, if it is inaccurate, it costs the company a lot of money. Using containers, it is possible to deploy the model to a fraction of the customers, only 10%, and if there are problems, it can be quickly reverted. If the model performs well, it can quickly replace the existing models.

Running Docker Containers

Using "base" images

One of the advantages of the Docker workflow for developers is the ability to use certified containers from the “official” development teams. In this diagram a developer uses the official Python base image which is developed by the core Python developers. This is accomplished by the `FROM` statement which loads in a previously created container image.

Docker “Base” Image Workflow



As the developer makes changes to the `Dockerfile`, they test locally, then push the changes to a private Docker Hub repo. After this the changes can be used by a deployment process to a Cloud or by another developer.

Common Issues Running a Docker Container

There are a few common issues that crop up when starting a container or building one for the first time. Let’s walk through each problem and then present a solution for them.

- What goes in a `Dockerfile` if you need to [write to the host filesystem](#)? In the following example the `docker volume` command is used to create a volume and then later it is mounted to the container.

```

> /tmp docker volume create docker-data
docker-data
> /tmp docker volume ls
DRIVER          VOLUME NAME
local          docker-data
> /tmp docker run -d \
  --name devtest \
  --mount source=docker-data,target=/app \

```

```
ubuntu:latest
6cef681d9d3b06788d0f461665919b3bf2d32e6c6cc62e2dbab02b05e77769f4
```

- How do you [configure logging](#) for a Docker container?

You can configure logging for a Docker container by selecting the type of log driver, in this example `json-file` and whether it is blocking or non-blocking. This example shows a configuration that uses `json-file` and `mode=non-blocking` for an ubuntu container. The `non-blocking` mode ensure that the application won't fail in a non-deterministic manner. Make sure to read the [Docker logging guide](#) on different logging options.

```
> /tmp docker run -it --log-driver json-file --log-opt mode=non-blocking ubuntu
root@551f89012f30:/#
```

- How do you map ports to the external host?

The Docker container has an internal set of ports that [must be exposed to the host and mapped](#). One of the easiest ways to see what ports are exposed to the host is by running the `docker port <container name>` command. Here is an example of what that looks like against a `foo` named container.

```
$ docker port foo
7000/tcp -> 0.0.0.0:2000
9000/tcp -> 0.0.0.0:3000
```

What about actually mapping the ports? You can do that using the `-p` flag as shown. You can read more about [Docker run flags here](#).

```
docker run -p 127.0.0.1:80:9999/tcp ubuntu bash
```

- What about configuring Memory, CPU and GPU?

You can configure docker `run` to accept flags for setting Memory, CPU and GPU. You can read [more about it here](#) in the official documentation. Here is a brief example of setting the CPU.

```
docker run -it --cpus=".25" ubuntu /bin/bash
```

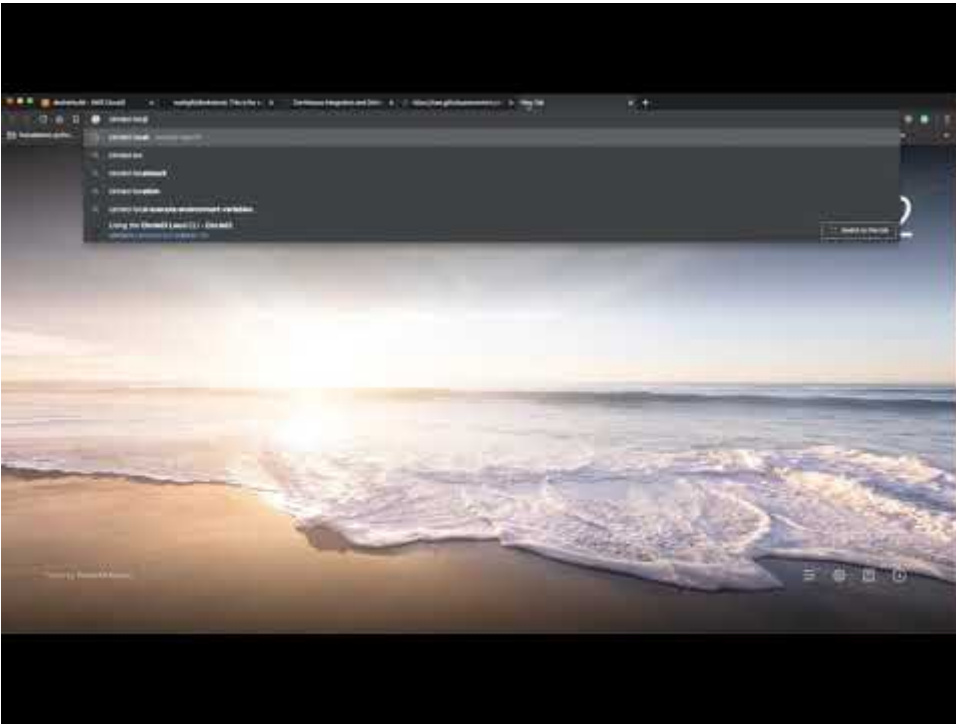
This tells this container to use at max only 25% of the CPU every second.

[TO DO: Docker GPU example]

Container Registries

Build containerized application from Zero on AWS Cloud9

Screencast



- [Docker Project Source Code](#)

1. Launch AWS Cloud9
2. Create Github repo
3. Create ssh keys and upload to Github
4. Git clone
5. Create structure
6. Create a local python virtual environment and source **MUST HAVE!**: `python 3 -m venv ~/.dockerproj && source ~/.dockerproj/bin/activate`

- `Dockerfile`

```
FROM python:3.7.3-stretch

# Working Directory
WORKDIR /app

# Copy source code to working directory
COPY . app.py /app/

# Install packages from requirements.txt
# hadolint ignore=DL3013
RUN pip install --upgrade pip &&\
    pip install --trusted-host pypi.python.org -r requirements.txt
```

- `requirements.txt`
- `Makefile`

```
setup:
    python3 -m venv ~/.dockerproj

install:
    pip install --upgrade pip &&\
```

```

    pip install -r requirements.txt

test:
    #python -m pytest -vv --cov=myrepolib tests/*.py
    #python -m pytest --nbval notebook.ipynb

validate-circleci:
    # See https://circleci.com/docs/2.0/local-cli/#processing-a-config
    circleci config process .circleci/config.yml

run-circleci-local:
    # See https://circleci.com/docs/2.0/local-cli/#running-a-job
    circleci local execute

lint:
    hadolint Dockerfile
    pylint --disable=R,C,W1203 app.py

all: install lint test

```

- `app.py`

1. Install hadolint (you may want to become root: i.e. `sudo su -` run this command then exit by typing `exit`).

```

wget -O /bin/hadolint https://github.com/hadolint/hadolint/releases/download/v1
      chmod +x /bin/hadolint

```

1. Create cirleci config

```

# Python CircleCI 2.0 configuration file
#
# Check https://circleci.com/docs/2.0/language-python/ for more details
#
version: 2
jobs:
  build:
    docker:
      # Use the same Docker base as the project
      - image: python:3.7.3-stretch

    working_directory: ~/repo

    steps:
      - checkout

      # Download and cache dependencies
      - restore_cache:
          keys:
            - v1-dependencies-
            # fallback to using the latest cache if no exact match is found
            - v1-dependencies-

      - run:

```

```

name: install dependencies
command: |
    python3 -m venv venv
    . venv/bin/activate
    make install
    # Install hadolint
    wget -O /bin/hadolint https://github.com/hadolint/hadolint/releases
    chmod +x /bin/hadolint

- save cache:
  paths:
    - ./venv
  key: vl-dependencies-

# run lint!
- run:
  name: run lint
  command: |
    . venv/bin/activate
    make lint

```

1. Install [local circleci](#) (optional)
2. setup requirements.txt

```

pylint
click

```

1. Create app.py

```

#!/usr/bin/env python
import click

@click.command()
def hello():
    click.echo('Hello World!')

if __name__ == '__main__':
    hello()

```

1. Run in container

```
docker build --tag=app .
```

```
docker run -it app bash
```

1. test app in shell

REMEMBER Virtualenv: `python3 -m venv ~/.dockerproj && source ~/.dockerproj/bin/activate`

`python app.py` or `chmod +x && ./app.py`

1. Test local circleci and local make lint and then configure circleci.

```
ec2-user:~/environment $ sudo su -  
[root@ip-172-31-65-112 ~]# curl -fLs https://circle.ci/cli | bash  
Starting installation.  
Installing CircleCI CLI v0.1.5879  
Installing to /usr/local/bin  
/usr/local/bin/circleci
```

1. Setup [Docker Hub Account](#) and deploy it!
2. To deploy you will need something like this (bash script)

```
#!/usr/bin/env bash
```

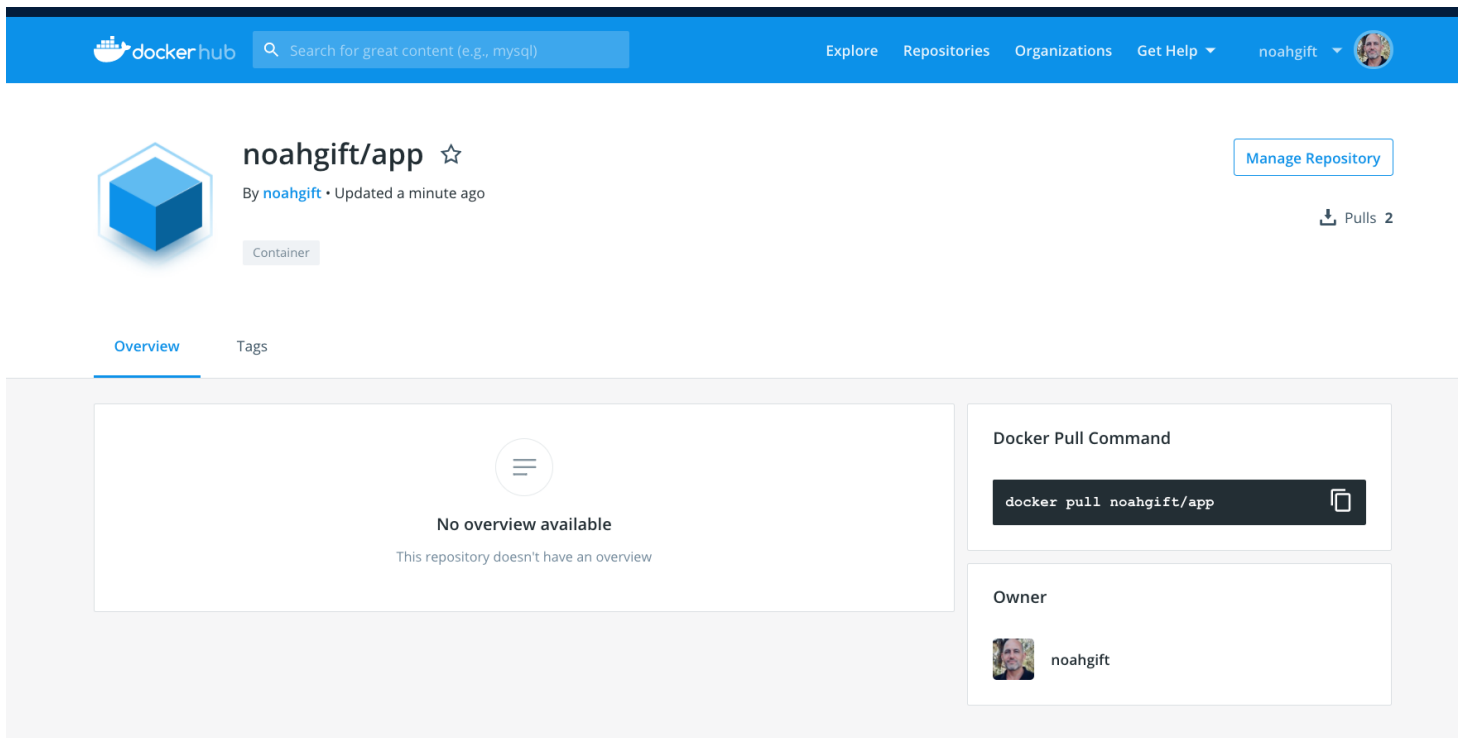
This tags and uploads an image to Docker Hub

Change to your repo

This is a sample script you could change do this or something like this: `touch push-docker.sh`
`&& chmod +x push-docker.sh && ./push-docker.sh`.

```
#!/usr/bin/env bash  
# This tags and uploads an image to Docker Hub  
  
#Assumes this is built  
#docker build --tag=app .  
  
dockerpath="noahgift/app"  
  
# Authenticate & Tag  
echo "Docker ID and Image: $dockerpath"  
docker login &&\  
    docker image tag app $dockerpath  
  
# Push Image  
docker image push $dockerpath
```

Any person can "pull now" `docker pull noahgift/app`



The screenshot shows the Docker Hub interface for the repository `noahgift/app`. The repository is owned by `noahgift` and was updated a minute ago. It is a container image. The page has tabs for `Overview` and `Tags`. The `Overview` tab is selected, but it displays a message: "No overview available. This repository doesn't have an overview." On the right, there is a "Manage Repository" button and a "Pulls 2" indicator. Below the message, there is a "Docker Pull Command" section showing the command `docker pull noahgift/app` and an "Owner" section showing the user `noahgift`.

An advanced version is here: <https://github.com/noahgift/container-revolution-devops-microservices/tree/master/demos/flask-sklearn>

Exercise

- Topic: Create Hello World Container in AWS Cloud9 and Publish to Docker Hub
- Estimated time: 20-30 minutes
- People: Individual or Final Project Team
- Slack Channel: #noisy-exercise-chatter
- Directions:
 - Part A: Build a hello world Docker container in AWS Cloud9 that uses the official Python base image. You can use the [sample command-line tools](#) in this repository for ideas.
 - Part B: Create an account on Docker Hub and publish there
 - Part C: Share your Docker Hub container in slack
 - Part D: Pull down another students container and run it
 - (Optional for the ambitious): Containerize a flask application and publish