Chapter 4: Working with Docker Images

-Docker or **Open Container Initiative (OCI) images** **provide basis** for everything you **deploy** and **run** with Docker. To launch container, you must download a public image or create your own. You can think image as a single asset that represents the filesystem for container. Every image consists of one or more **linked filesystem layers** that have a direct one-to-one **mapping** to each **build step** used to create that image.

-**Images** built up from individual **layers**, they put special **demands** on **Linux kernel**, which must **provide** the drivers that Docker needs to **run storage backend**. The storage backend **communicates** with **Linux filesystem** to **build and manage layers** combine into image. The primary storage backend supported: Overlay21, B-Tree File System (Btrfs), Device Mapper. Each storage backend provide a **fast copy-on-write** (CoW) **system** for image management.

# 1. Anatomy of a Dockerfile

-To create custom image, you use **Dockerfile**: a file describes all the **steps** that are required to create an image and contained **within** the **root directory** of source code repository for your app.



-Each line create a new image layer which contains all of changes that are a result of that command being issued. So when you build new images, Docker will need to build layers that deviate from previous builds, you can reuse all layers that haven’t changed.

-: Although you can build a Node instance from a base linux image, you can explore **Docker Hub** for official images (<https://registry.hub.docker.com/>). It maintains a series of images and tags to determine what versions are available. This code provide with an Ubuntu Linux image running Node 11.11.x

-: **set variables and their default values**, which are **only available** during **image build process**.

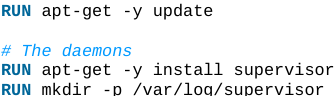
-

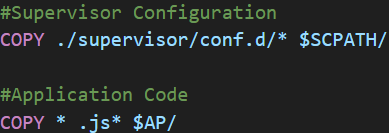
+Applying **labels** allows you to **add metadata** via key/value pairs that can later used to **search** for identify Docker **images** and **containers**

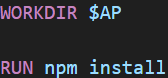
+You can see labels applied to any images using: **docker image inspect**

-: By **default**, Dockers **run all process** as **root** within container, you can USER instruction to change this.

**:** set **shell variables** used by your running application for configuration, to being available during build process.

-: start and create required file structure you need, and install some required software dependencies.

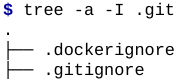
-: copy files from local filesystem into your image. Most often this will include your app code and any required support files. So you no longer need access to local filesystem to access them once the image is built.

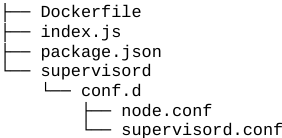
-: Change the working directory in the image for the remaining build instruction and the default process that launches with any resulting containers.

-: defines the command that launches the process that you want to run within the container.

-Note: It’s considered a best practice to try to run only a single process within a container. In example, you use supervisor as a process manager to help improve the resiliency of node app within container and ensure that it stays running. This can useful for troubleshooting during dev so you can restart your service without restarting the whole container.

# 2. Building an Image



  
-.dockerignore: define files and directories don’t want to upload to Docker host when building image.

-supervisord directory contains configuration files for supervisord that you will use to start and monitor the application.

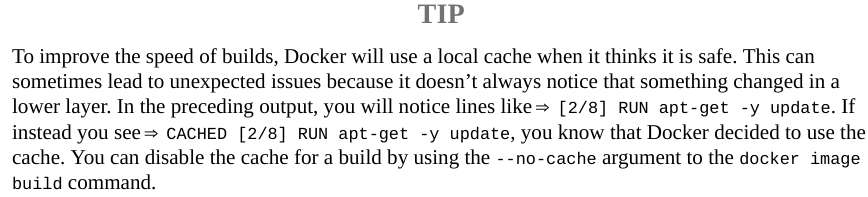
-Each step identified in output maps to a line in Dockerfile, each step creates a new image layer based on previous step. The 1st build will take a few minute as you download base node image. At the end of build command, you see “.”: refer to build context, which tells Docker what files it should upload to server so that it can build our image. “.” Represents the current directory. This build context is what .dockerignore file is filtering.

+-f: point directly to Dockerfile

-Run the build:



+Tip:



+Tip: docker image build = docker build

# 3. Running your Image

-Run image on your Docker host:



+This command tells Docker to create a running container in background from image with example/docker-node-hello:latest tag, then map port 8080 in container to port 8080 on Docker host.

+: verify running containers

+: determine Docker host IP address by examming the entry. Check the value of DOCKER\_HOST env variable. If DOCKER ENDPOINT is set to a Unix socket, IP address is likely 127.0.0.1

## 3.1 Build Arguments

-Inspect the image:



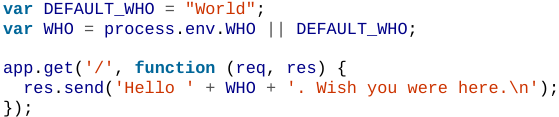
-Change maintainer label: rerun the build and provide a new value:



+Then reinspect: dokcer image inspect…

## 3.2 Environment Variables as Configuration

-In index.js: refers to variable $WHO, which the app uses to determine who the app is going to say:



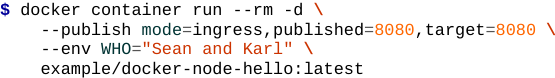
-Configure this app by passing in environment variables when you start it:

+Stop the existing container:

: see containers ID, image, status

: stop container

+Restart after adding --evn:



You can shorten the command:



# 4. Custom Base Images

# 5. Storing Images

## 5.1 Public Registries

## 5.2 Private Registries

## 5.3 Authenticating to a Registry

### 5.3.1 Creating a Docker Hub account

### 5.3.2 Logging into a registry

### 5.3.3 Pushing images into a repository

### 5.3.4 Exploring images in Docker Hub

## 5.4 Running a Private Registry

### 5.4.1 Testing the private registry

# 6. Optimizing Images

## 6.1 Keeping Images Small

### 6.1.1 Testing the private registry

## 6.2 Layers are additive

## 6.3 Utilizing the Layer Cache

## 6.4 Directory Caching

# 7. Troubleshooting Broken Builds

## 7.1 Debugging Pre-BuildKit Images

## 7.2 Debugging BuildKit Images