# Creating Docker Images

# Getting Demos and Lab Code

- We've provided various demos and labs here:
  - http://olsensoft.com/DockerOpenShift/DoshDev.zip

- You must download the zip into your Linux box, and unzip
  - To do this, run the following commands in your Linux box:

```
wget http://olsensoft.com/DockerOpenShift/DoshDev.zip -O temp.zip; \
unzip temp.zip; \
rm temp.zip
```

# Contents

# 1. Creating a Simple Docker Image

- Overview
- Defining a Dockerfile
- Understanding the Dockerfile
- Our Python program
- Building the image
- Viewing images in the local Docker registry
- Running a container

# Overview

- In this section, we're going to show how to create a simple Docker image
  - Remember, a Docker image is a "black box" executable package
  - It includes everything needed to run an application

- In our case, we'll create a Docker image containing:
  - The Python runtime
  - Our Python program
  - A command to execute the Python program

- For the demo code, see this folder:
  - `PythonImage1`

# Defining a Dockerfile

- In order to define a Docker image, you define a special file named `Dockerfile` (by default)
  - Specifies build instructions, telling Docker how to build an image

- Here's the Dockerfile in the demo folder, `PythonImage1`

```
FROM python:3.8.5

COPY ./main.py app.py

ENTRYPOINT ["python", "app.py"]
```

- See following slides for an explanation
  - Also see https://docs.docker.com/engine/reference/builder/

# Understanding the Dockerfile (1 of 3)

```
FROM python:3.8.5

COPY ./main.py app.py

ENTRYPOINT ["python", "app.py"]
```

- A Dockerfile must start with a `FROM` instruction
  - Specifies the "base image" from which you are building
  - Our image is based on Python version 3.8.5
  - If you want to create an image from scratch, use `FROM SCRATCH`

- When you run this Dockerfile to build your image…
  - Docker will see if you've already downloaded Python into your local Docker registry
  - If not, Docker will automatically download it from Docker Hub

# Understanding the Dockerfile (2 of 3)

```
FROM python:3.8.5

COPY ./main.py app.py

ENTRYPOINT ["python", "app.py"]
```

- A Dockerfile specifies files and folders to copy into the Docker image
  - Use the `COPY` instruction to copy files and folders into the image
  - In our example, we copy `main.py` into the image
  - Inside the image, the Python file will be named `app.py`

# Understanding the Dockerfile (3 of 3)

```
FROM python:3.8.5

COPY ./main.py app.py

ENTRYPOINT ["python", "app.py"]
```

- The `ENTRYPOINT` instruction specifies what to actually execute inside the Docker image
  - In our example, we run the Python interpreter and tell it to run our Python program
  - We've used the "exec form" of the `ENTRYPOINT` instruction here, which is JSON array syntax

- The `ENTRYPOINT` instruction also supports "shell form" syntax, and executes it in `/bin/sh -c`

```
ENTRYPOINT python app.py
```

# Our Python Program

- Here's our Python program ☺

```
print ("Hello from my uber cool Python app #1")
```

# Building the Image (1 of 2)

- Use the `docker build` command to build a Docker image

```
docker build -t  mypython1 .
```

- `-t`
  - Specifies the tag name for the image
- `.`
  - Specifies the current directory as the build context
  - You can also specify a path, URL, or tarball
- Note:
  - Assumes the dockerfile is called `Dockerfile` (use `-f` if different)

# Building the Image (2 of 2)

- Here's how it looks when we use `docker build`

```
docker build -t mypython1 .
```

```
$ docker build -t mypython1 .
Sending build context to Docker daemon  3.072kB
Step 1/3 : FROM python:3.8.5
3.8.5: Pulling from library/python
d6ff36c9ec48: Pull complete
c958d65b3090: Pull complete
edaf0a6b092f: Pull complete
80931cf68816: Pull complete
7dc5581457b1: Pull complete
87013dc371d5: Pull complete
dbb5b2d86fe3: Pull complete
4cb6f1e38c2d: Pull complete
c2df8846f270: Pull complete
Digest: sha256:bc765f71aaa90648de6cfa356ec201d50549031a244f48f8f477f386517c5d1b
Status: Downloaded newer image for python:3.8.5
 ---> a7cda474cef4
Step 2/3 : COPY ./main.py app.py
 ---> b58c54a9abac
Step 3/3 : ENTRYPOINT ["python", "app.py"]
 ---> Running in 0d17e4db3be2
Removing intermediate container 0d17e4db3be2
 ---> 465145c39144
Successfully built 465145c39144
Successfully tagged mypython1:latest
```

# Viewing Images in the Local Docker Registry

- You can view images in the Docker registry

```
docker image ls
```

```
$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
mypython1           latest              465145c39144        3 minutes ago       882MB
python              3.8.5               a7cda474cef4        4 days ago          882MB
```

# Running a Container

- You can run containers based on the image

```
docker run mypython1
```

```
$ docker run mypython1
Hello from my uber cool Python app #1
```

# 2. A Closer Look at Dockerfile Instructions

- Overview
- The `ARG` instruction
- The `RUN` instruction
- The `WORKDIR` instruction
- The `COPY` and `ADD` instructions
- The `EXPOSE` instruction
- The `ENTRYPOINT` instruction

# Overview

- In this section we're going to look at a more interesting Dockerfile, to explore additional Docker techniques

```
FROM python:3.8.5

ARG PYTHON_MAIN_FILE

RUN mkdir /app

WORKDIR /app

COPY ./requirements.txt /app
COPY ${PYTHON_MAIN_FILE} /app/main.py

RUN pip install --trusted-host pypi.python.org -r requirements.txt

EXPOSE 5000

ENTRYPOINT ["python", "main.py"]
```

- For the demo code, see this folder:
  - `PythonImage2`

# The ARG Instruction

- The ARG instruction allows you to specify build arguments for the Dockerfile, to make it more flexible

```
…
ARG PYTHON_MAIN_FILE
…
COPY ${PYTHON_MAIN_FILE} /app/main.py
…
```

- You pass argument values into the Dockerfile as part of the Docker build command
  - Via the `--build-arg` flag

```
docker build -t mypython2 \
--build-arg PYTHON_MAIN_FILE=server.py \
.
```

# The RUN Instruction

- The RUN instruction allows you to run any valid Linux command, e.g. to make a directory in the image

```
RUN mkdir /app
```

- Remember each statement in a Dockerfile creates a separate layer in the union filesystem
  - So it's best to combine multiple Linux commands into a single RUN statement, to minimize the number of layers

```
RUN mkdir /app && cd /app
```

```
RUN mkdir /app
RUN cd /app
```

# The WORKDIR Instruction

- The WORKDIR instruction sets the working directory inside the image
  - All activity that happens inside the image after this statement will use this as the working directory

```
WORKDIR /app
```

- Note that WORKDIR is different to cd
  - WORKDIR sets the context across layers in the image filesystem
  - But cd only applies to the current layer, not subsequent layers

```
WORKDIR /app
RUN touch logfile.txt      # Creates /app/logfile.txt
```

```
RUN cd /app
RUN touch logfile.txt      # Creates /logfile.txt
```

# The COPY and ADD Instructions (1 of 2)

- The COPY instruction copies files and folders from the host filesystem into the image filesystem

```
COPY ./requirements.txt /app
COPY ${PYTHON_MAIN_FILE} /app/main.py
```

- The ADD instruction is similar, but more flexible
  - Allows you to specify URLs and tarballs
  - As well as files and folders, of course

```
ADD mytarball.tar /app
ADD http://olsensoft.com/mydata /data
```

# The COPY and ADD Instructions (2 of 2)

- It's common practice to copy all of the contents of a folder on the host system into the image

```
COPY . /app
```

- You can define a `.dockerignore` file to tell Docker to ignore some files and folders during the copying

```
.git

**/*.class

*.md
!readme*.md
```

`.dockerignore`

# The EXPOSE Instruction

- The EXPOSE instruction tells Docker that the container listens on the specified network port(s) at runtime

```
EXPOSE 5000
```

- Note: EXPOSE doesn't actually publish the port
  - It just acts as documentation between the person building the image and the person running the container
  - To publish the port when running the container, use the -p flag

```
docker run -d -p 5000:5000 mypython2
```

# The ENTRYPOINT Instruction

- The ENTRYPOINT instruction tells Docker what to execute in the running container

- Here's the "exec form" of ENTRYPOINT
  - This is generally preferred

```
ENTRYPOINT ["python", "main.py"]
```

- There's also the "shell form" of ENTRYPOINT
  - This executes in a shell, via /bin/sh -c

```
ENTRYPOINT python main.py
```

# Pinging the Python Rest Service

- The containerized Python application is running a REST service, with endpoints such as `api/Products`

- You can ping the REST endpoint as follows:

```
curl localhost:5000/api/Products
```

```
$ curl localhost:5000/api/Products
[{"id": 0, "description": "Swansea City shirt", "price": 55, "unitsInStock": 500}, {"id": 1, "de
scription": "Cardiff City shirt", "price": 1.99, "unitsInStock": 20000}, {"id": 2, "description"
: "Bugatti Divo", "price": 4000000, "unitsInStock": 2}, {"id": 3, "description": "Carving Skis",
 "price": 350, "unitsInStock": 75}, {"id": 4, "description": "Ski Boots", "price": 150, "unitsIn
Stock": 150}, {"id": 5, "description": "55in OLED HDTV", "price": 1800, "unitsInStock": 100}]
```

# 3. Parameterizing Docker Containers

- The CMD instruction
- Running a container with default arguments
- Running a container and overriding arguments

# The CMD Instruction

- The CMD instruction is often used alongside ENTRYPOINT
  - ENTRYPOINT specifies the entrypoint to run, plus stable args
  - CMD specifies additional args, which you might want to override when you run a container instance

- Example - see the EntrypointCmdExample folder

```
FROM alpine:latest
ENTRYPOINT ["ping", "127.0.0.1"]
CMD ["-c", "3"]
```
Dockerfile

- Build the image as normal

```
docker build -t entrypoint_cmd .
```

# Running a Container with Default Arguments

- If you run the container as follows, it uses the default command-line args as specified by CMD

```
docker run -t entrypoint_cmd
```

```
$ docker run -t entrypoint_cmd
PING 127.0.0.1 (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: seq=0 ttl=64 time=0.751 ms
64 bytes from 127.0.0.1: seq=1 ttl=64 time=0.077 ms
64 bytes from 127.0.0.1: seq=2 ttl=64 time=0.106 ms

--- 127.0.0.1 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.077/0.311/0.751 ms
```

# Running a Container and Overriding Arguments

- If you run the container as follows, it overrides the command-line args specified by CMD

```
docker run -t entrypoint_cmd -q -w 10
```

```
$ docker run -t entrypoint_cmd -q -w 10
PING 127.0.0.1 (127.0.0.1): 56 data bytes

--- 127.0.0.1 ping statistics ---
11 packets transmitted, 10 packets received, 9% packet loss
round-trip min/avg/max = 0.069/0.138/0.634 ms
```

# Any Questions?

# Annex: The Layered Filesystem of Docker Images

- Overview
- Example of layering
- Image layers and container layers
- Advantages of the layered approach

# Overview

- Recall the Dockerfile from the previous section, which specifies a simple image for a Python app

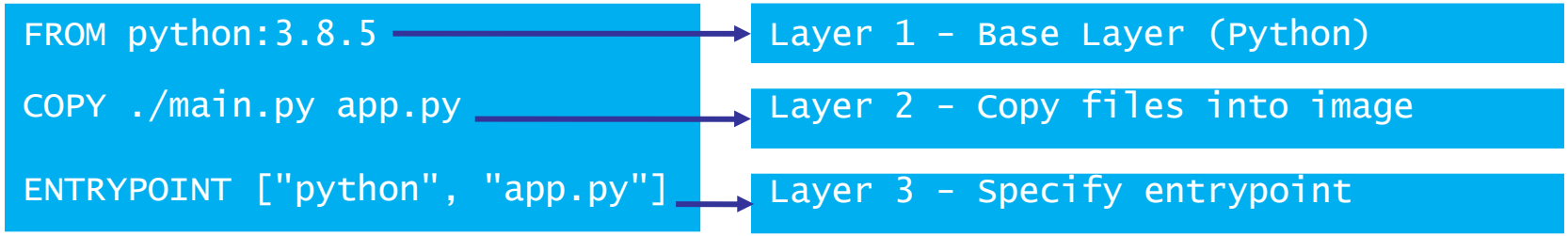
```
FROM python:3.8.5

COPY ./main.py app.py

ENTRYPOINT ["python", "app.py"]
```

- A Docker image is a layered filesystem
  - Each layer contains only the changes to the filesystem, relative to the underlying layers

- Docker uses a "union filesystem"
  - It creates a virtual filesystem for an image, out of a set of layers

# Example of Layering

- Here are the layers for our simple Python image

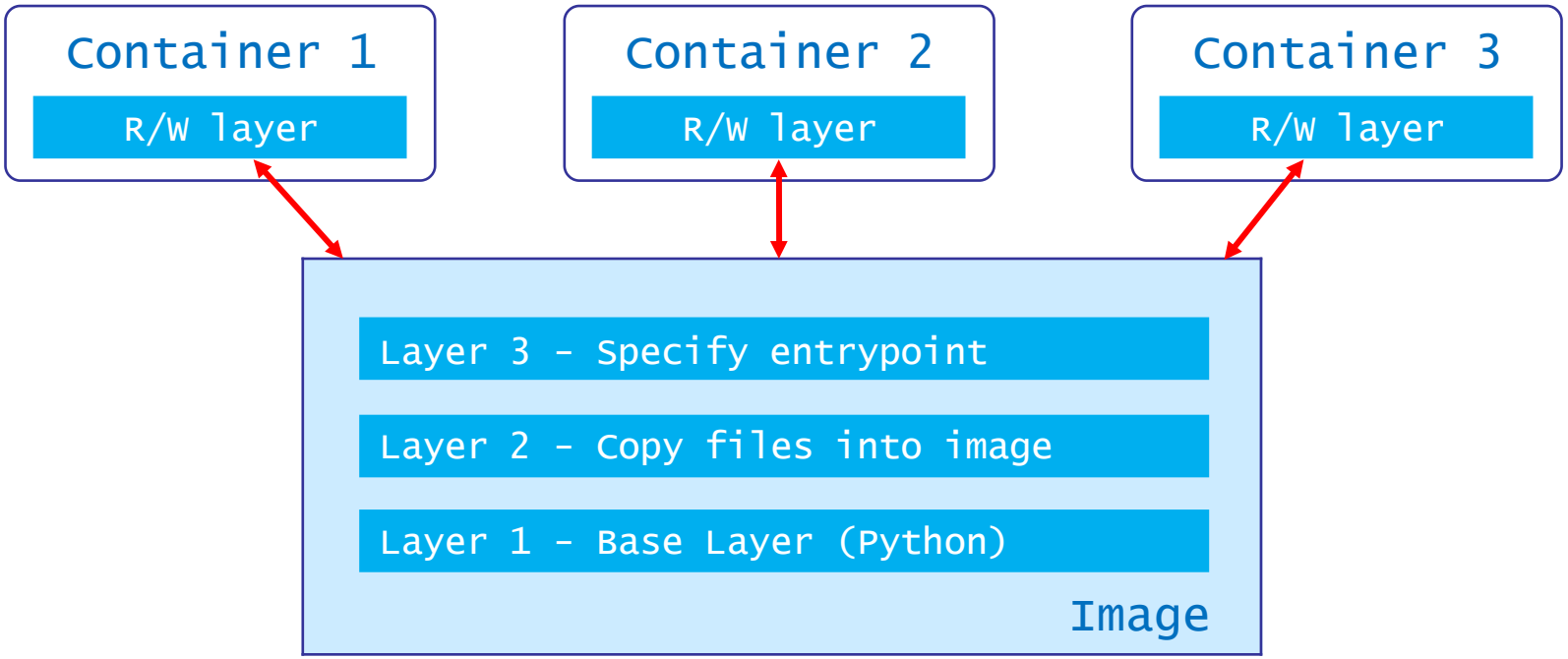



- Each statement in the Dockerfile creates another layer
  - Layer 1 is the base layer (the Python distribution here)
  - Layer 2 copies files into the image
  - Layer 3 specifies the entrypoint

- The content of each layer is mapped to a special folder on the host system

# Image Layers and Container Layers

- Each layer in an image is read-only
  - Layers are shared among containers created from the same image
- When Docker runs a container from an image…
  - It adds a thin R/W container layer on top, just for that container



# Advantages of the Layered Approach

- Reduction of resource consumption
  - The read-only image layers are only loaded into memory once, when the first container is run
- Fast load-time for containers
  - When we run a container, Docker only needs to create the thin R/W layer for that container
  - All the underlying layers are already loaded (except for the first time we run a container)