Agenda

Section 1:
What is Docker
What is Not Docker
Basic Docker Commands
Dockerfiles

Section 2:
Anatomy of a Docker image
Docker volumes

Section 3:
Networking

Section 4:
Docker compose / stacks
Demo
Section 1: What is Docker
Basic Docker Commands
Dockerfiles
What Is Docker?

• Lightweight, open, secure platform
  Simplify building, shipping, running apps

• Runs natively on Linux or Windows Server

• Runs on Windows or Mac Development machines (with a virtual machine)

• Relies on "images" and "containers"
What is a container?

- Standardized packaging for software and dependencies
- Isolate apps from each other
- Share the same OS kernel
- Works for all major Linux distributions
- Containers native to Windows Server 2016
The Role of Images and Containers

Docker Image

Example: Ubuntu with Node.js and Application Code

Docker Container

Created by using an image. Runs your application.
Docker containers are NOT VMs

- Easily misconceptualised
- Fundamentally different architectures
Docker Containers Versus Virtual Machines

**Virtual Machines**

- Hypervisor
- Guest OS
- Bins/Libs
- App 1
- App 2

**Docker Containers**

- Hypervisor
- Docker Engine
- Guest OS
- Bins/Libs
- App 1
- App 2
- Bins/Libs

**Host Operating System**
Using Docker: Build, Ship, Run Workflow

**Developers**

**BUILD**
Development Environments

**SHIP**
Create & Store Images

**RUN**
Deploy, Manage, Scale

**IT Operations**
Some Docker vocabulary

**Docker Image**
The basis of a Docker container. Represents a full application

**Docker Container**
The standard unit in which the application service resides and executes

**Docker Engine**
Creates, ships and runs Docker containers deployable on a physical or virtual, host locally, in a datacenter or cloud service provider

**Registry Service (Docker Hub (Public) or Docker Trusted Registry (Private))**
Cloud or server based storage and distribution service for your images
Basic Docker Commands

$ docker image pull node:latest

$ docker image ls

$ docker container run -d -p 5000:5000 --name node node:latest

$ docker container ps

$ docker container stop node(or <container id>)

$ docker container rm node (or <container id>)

$ docker image rmi (or <image id>)

$ docker build -t node:2.0 .

$ docker image push node:2.0

$ docker --help
Docker Build Cache Gotcha

- Sometimes you will change your Dockerfile and do a build and yet your container image will not change.
- This is because of the docker **cache** – when you do a docker build it tries to intelligently cache the layers such that it only rebuilds the minimum number of layers.
- You can override this behavior by doing:
  - `docker build -t <image-name> --no-cache`
- You can also avoid this by deleting the container image and then rebuilding it, but it is likely more convenient for you to use the no-cache option in docker build shown above.

- Sometimes you may also need to delete the image and completely regenerate.
  - You can remove all unused images with `docker image prune -a`
Docker Build Args Gotcha

- You can pass build
Dockerfile – Linux Example

- Instructions on how to build a Docker image
- Looks very similar to “native” commands
- Important to optimize your Dockerfile

Dockerizing a Node.js web app
Section 2:
Anatomy of a Docker
Let’s Go Back to Our Dockerfile

```dockerfile
# Create image based on the official Node 6 image from dockerhub
FROM node:latest

# Create a directory where our app will be placed
RUN mkdir -p /usr/src/app

# Change directory so that our commands run inside this new directory
WORKDIR /usr/src/app

# Copy dependency definitions
COPY package.json /usr/src/app

# Install dependencies
RUN npm install

# Get all the code needed to run the app
COPY . /usr/src/app

# Expose the port the app runs in
EXPOSE 4200

# Serve the app
CMD ["npm", "start"]
```
Each Dockerfile Command Creates a Layer

... 
EXPOSE
COPY
WORKDIR
RUN
FROM
Kernel
Docker Image Pull: Pulls Layers

```
Alexander@DESKTOP-90ATKET MINGW64 ~/Docker/Demo
$ docker pull nginx:latest
latest: Pulling from library/nginx
bc95e04b23c0: Pull complete
f3186e650f4e: Pull complete
9ac7d6621708: Pull complete
Digest: sha256:b81f317384d7388708a498555c28a7c6e778a8f291d90021208b3e6ba3fe74887
Status: Downloaded newer image for nginx:latest
```
Section 3: Networking
What is Docker Bridge Networking

docker network create -d bridge --name bridgenet1
Docker Bridge Networking and Port Mapping

$ docker container run -p 8080:80 ...
Section 4: Docker Compose
Docker Compose: Multi Container Applications

- Build and run one container at a time
- Manually connect containers together
- Must be careful with dependencies and start up order

- Define multi container app in compose.yml file
- Single command to deploy entire app
- Handles container dependencies
- Works with Docker Swarm, Networking, Volumes, Universal Control Plane
Docker Compose: Multi Container Applications

```yaml
version: '2' # specify docker-compose version

# Define the services/containers to be run
services:
  angular: # name of the first service
    build: client # specify the directory of the Dockerfile
    ports:
      - "4200:4200" # specify port forwarding

  express: # name of the second service
    build: api # specify the directory of the Dockerfile
    ports:
      - "3977:3977" # specify ports forwarding

  database: # name of the third service
    image: mongo # specify image to build container from
    ports:
      - "27017:27017" # specify port forwarding
```

 compose.yml
 images
 ports
 volumes
 links
Docker Compose Networking

• By default, docker compose will put all of the **services** specified in your compose.yml file will be put on a docker network together.
• This allows you to access the other containers in the network via their name in the compose.yml file.
• If you have one service named **server** and another service named **database**
  • Suppose database exposes port 5001 to access the database
  • In the server container you can use **database:5001** to access it across the network
• Helpful Tip: The server container may take some time to
Docker Compose Networking

• By default, docker compose will put all of the **services** specified in your compose.yml file will be put on a docker network together.
• This allows you to access the other containers in the network via their name in the compose.yml file.
• If you have one service named **server** and another service named **database**
  • Suppose database exposes port 5001 to access the database
  • In the server container you can use **database:5001** to access it across the network
Docker Compose: Scale Container Applications
Python Server

To run Python server-side code, you'll need to use a Python web framework. Flask is a good lightweight web framework.

To run this you'll need to install Python/PIP, then install Flask using pip3 install flask. (This should be done using the Requirements.txt and docker file)

At this point you should be able to run the Python Flask examples using for example python3 python-example.py, then navigating to localhost:5000 in your browser.
To run Python server-side code, you'll need to use a Python web framework. Flask is a good lightweight web framework.

To run this you'll need to install Python/PIP, then install Flask using pip3 install flask. (This should be done using the Requirements.txt and docker file)

At this point you should be able to run the Python Flask examples using for example python3 python-example.py, then navigating to localhost:5000 in your browser.
Python Client

To run Python Client-side code, you'll need to use requests framework. This is included by importing the urllib.req.

Then you need to listen the port you have exposed from the server.

Read the content from the port, print the values and close the connection.