Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU44000 Internet Applications

Week 8 Containers

Conor Sheedy

Containers .. Lightweight Packaging

Virtual Machines are a good form of packaging but…

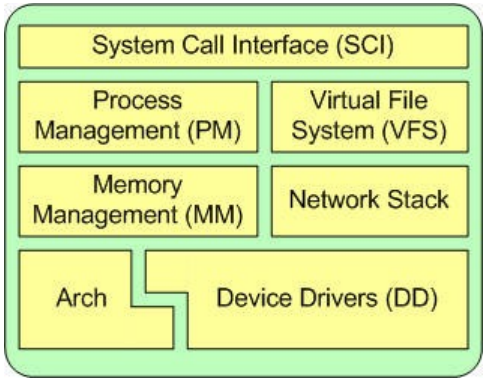
- Takes a hit on raw machine performance at the instruction level
- Requires each machine to be booted (several minutes) and have its own native filesystem
- Consumes memory for OS in each machine

Linux Containers solved many of these problems

- an operating-system-level virtualization method
- enables running multiple isolated Linux systems (containers)
- on a control host using a single Linux kernel

Structure of Linux

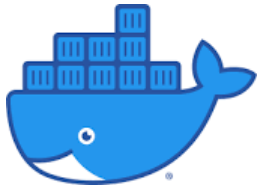
- **At the core of the Linux Operating System is the Kernel**
- **Various groups, companies have constructed components that go on top of this:**
  - Package Managers (Apt, Yum)
  - Windowing systems etc
- **The mix of kernel and components are called Distributions**
  - Popular ones include: Ubuntu, RedHat, Debian
  - 600 distributions exist
  - 500 distributions in active development
- **Taxonomy:** https://upload.wikimedia.org/wikipedia/commons/1/1b/Linux_Distribution_Timeline.svg



Linux Containers

- **In 2007, two engineers at Google began to work on Kernel cgroups**
  - which: “limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.”
- **merged into the Linux kernel mainline in kernel version 2.6.24**
  - released in January 2008
- **A control group is a collection of processes bound by the same criteria**
  - e.g. memory limit, cpu usage limit
- **In parallel, work was ongoing in Namespace Isolation**
  - each control group has separate namespaces for processes, networking, file system and could not see the resources in other control groups
- **LXC – 1st stable release, 2014**
  - runs multiple isolated Linux systems (containers)
  - on a control host using a single Linux kernel

Docker



- **A company called Docker Inc was launched in 2011**
  - from Y-Combinator
  - initially working on PaaS (Platform as a Service)
- **Released as Open Source in 2013**
  - used LXCs as a core component
- **In 2014 released 0.9 based on their own container code which replaced LXC and is written in Go**
- **In order to distinguish Docker the product from the open-source code**
  - Docker created project Moby in 2007

Timeline: Development of Containers

2007:Work on Cgroups, Linux Containers

Internally using LXCs and Borg

2014 Release Kubernetes

2006: Launch AWS EC2

Late 2014 Elastic Container Service

Late 2017 Elastic Kubernetes Svc (EKS)

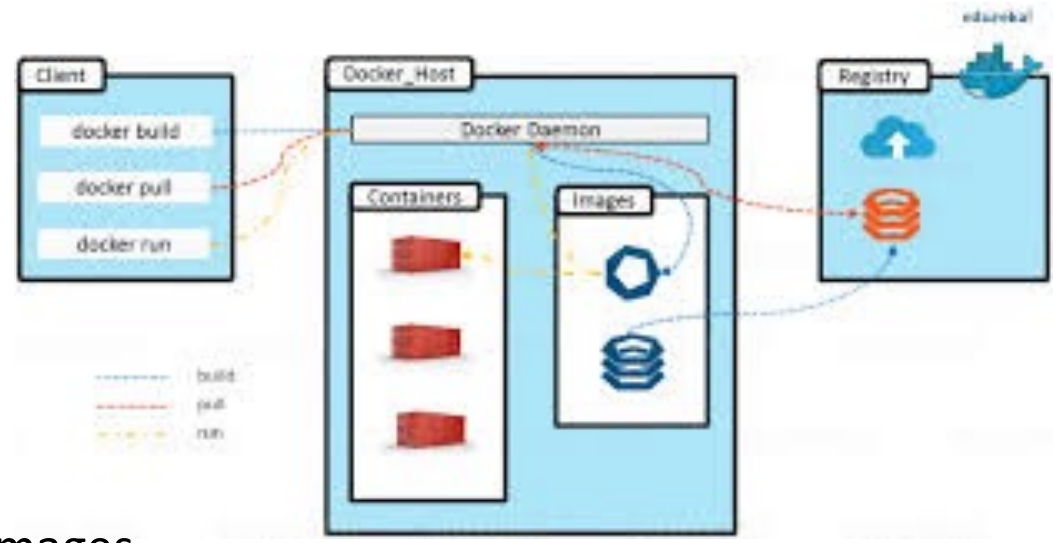
2013 Docker launched

Using Docker Containers

- **A container is built-up using layers**
  - Uses a Copy-on-Write system
  - so similar containers do not cause duplicate layers
- **Base Layer contains the Linux Kernel**
- **Next layer on is likely an OS distro**
  - e.g. Ubuntu
- **Then key operating software**
  - e.g. Node.js
- **Finally end user applications**

- **This forms a Container specification**
  - Can easily ship complex configuration with a container image

Structure of Docker



- **The target machine runs a docker 'host' – This contains:**
  - a single(usually) Linux Kernel
  - A set (initially empty) of container Images
  - A set (initially empty) of running container
- **It is remotely controlled by a**
  - Docker Client
  - using a REST API
- **If it is asked to use a container image that is not in its set of images, it can retrieve it from a container registry**

Simple Example: Hello World

Install and run Docker then:

docker run hello-world

- **In this example the docker client asks the docker host to "run hello-world" image**
- **The host does not have it locally, so it pulls it from the public registry**
  - hub.docker.com
  - 2.7m images
- **It starts a container with this image which runs to completion**

```
(base) conorsheedy@Conors-MacBook-Air Documents % docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
70f5ac315c5a: Pull complete
Digest: sha256:88ec0acaa3ec199d3b7eaf73588f4518c25f9d34f58ce9a0df68429c5af48e8d
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (arm64v8)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Dockerfile – creating an image

- **Docker containers are described using a Dockerfile**
- **Starts with a Node.js (v 20) image**
  - ‘FROM node:10’
- **Set the working directory**
  - ‘WORKDIR /usr/src/app
- **Copies some files into image**
  - ‘COPY package *.json ./’
- **Run commands on image**
  - ‘RUN npm install’
- **Exposes port 8080 for traffic**
  - ‘EXPOSE 8080’
- **Specifies a startup command** ’CMD [“node”, “server.js”]’

```
FROM node:20

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
# A wildcard is used to ensure both package.json
# AND package-lock.json are copied
# where available (npm@5+)
COPY package*.json ./

RUN npm install
# If you are building your code for production
# RUN npm ci --only=production

# Bundle app source
COPY . .

EXPOSE 8080
CMD [ "node", "server.js" ]
```

Dockerfile – creating an image

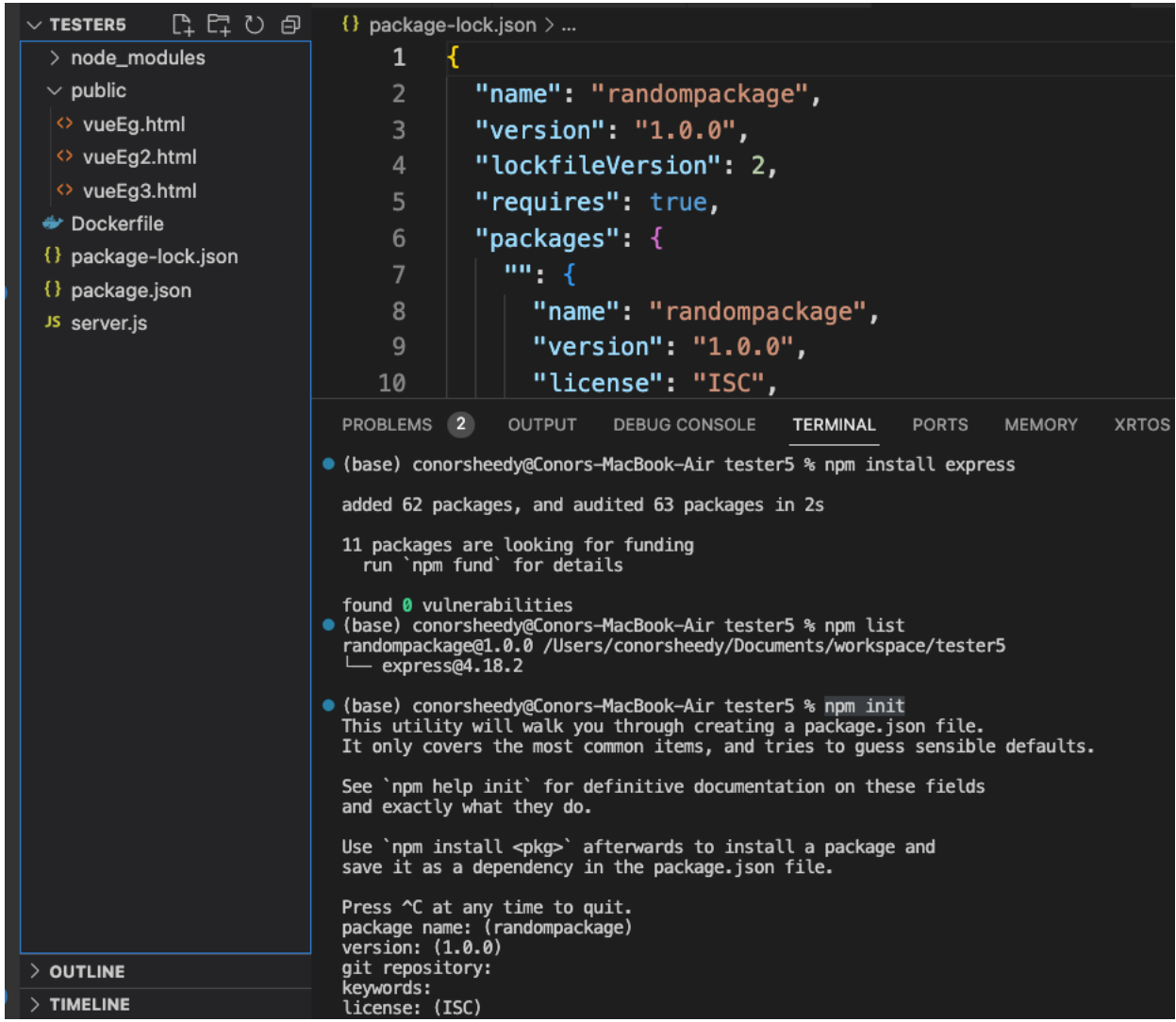
- **Install Docker**
  - sign up for a ‘personal account’
  - Install Docker Desktop if you are using Windows or Mac
    - Linux Virtual Machine with GUI
    - The Docker “host”

- **Create a file called “Dockerfile”**
  - in the same directory as your server
  - Paste from the previous file

- **Create a node package file that includes express**
  - “package

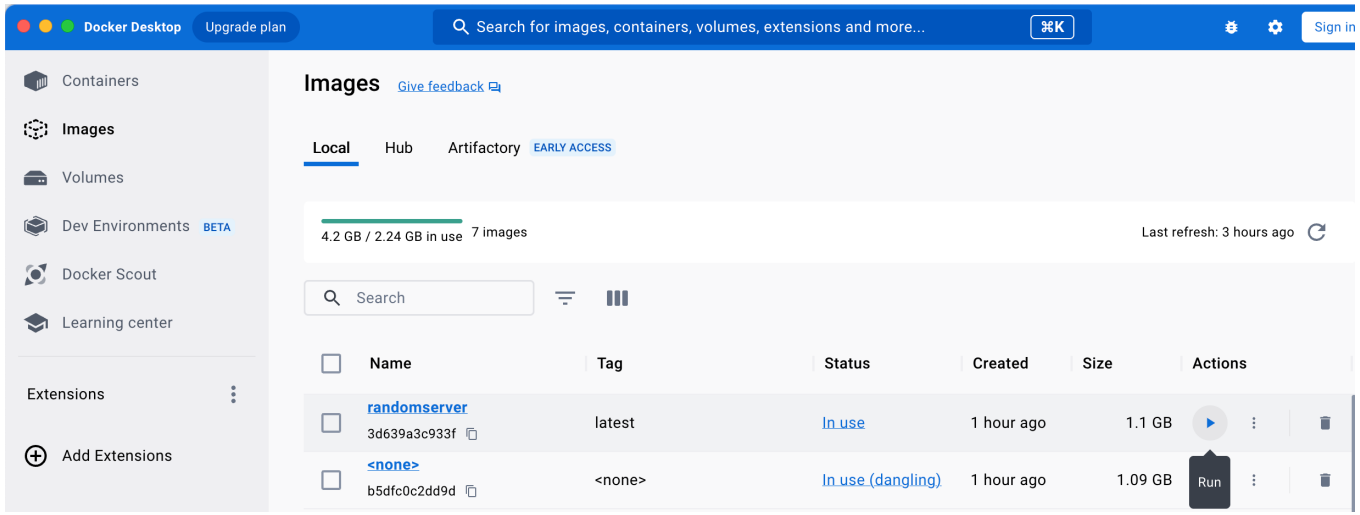
Dockerfile – creating a node package file

- **Create a node package file that includes express**
  - npm install express
    - To install express
  - npm list
    - To check that it has been installed
  - npm init
    - Answer the questions
    - package.json and package-lock.json are created

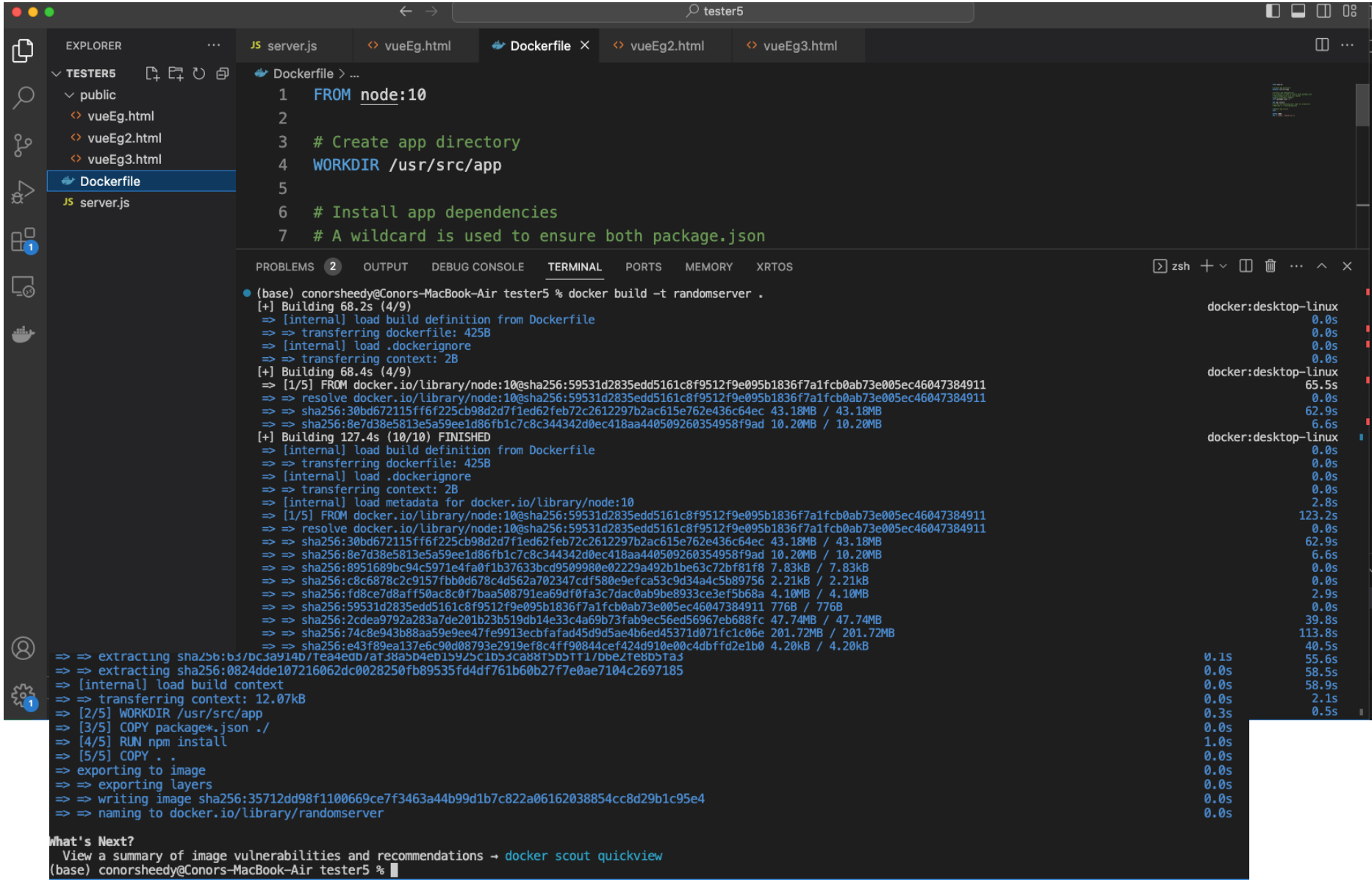


Exercise – Building a container for ExpressRandom.js

- **Create Docker Image** docker build –t myfirstdockerimage .
  - Creates the container image (~ 1Gb)
- **Run an instance of the container** docker run <switches> myfirstdockerimage
  - Or use GUI with Docker Desktop



Dockerfile – creating an image

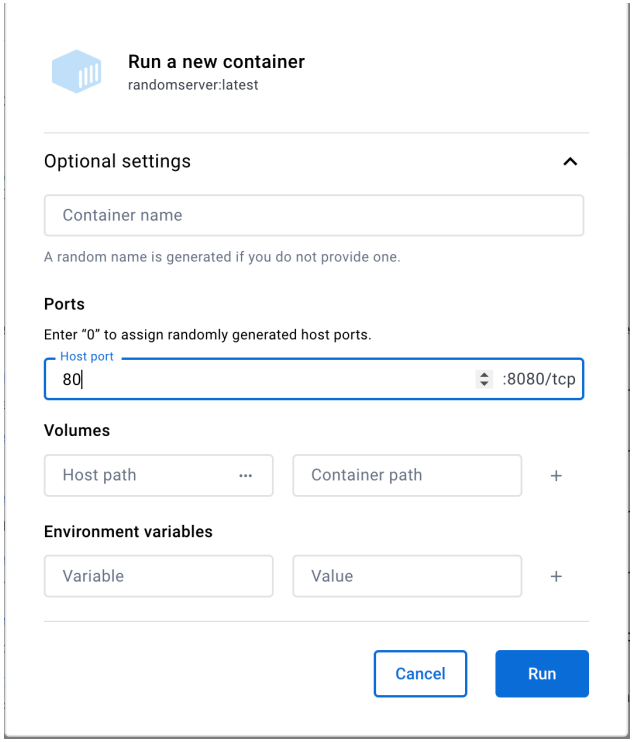


Managing Container Run-environment - Ports

- **The Docker-host has complete control over the networking environment of the running container**
  - can map ports on it to alternative ports on the running machine

docker run –p 80:8080 XXXXXXXXX

- **Above maps port 80 on the local machine to port 8080 on running container**
  - You can also do this in the “Optional settings” in Docker Desktop
  - Now use your browser to connect to localhost:80/



Managing Container Run-Environment - Volumes

- **The disk storage in containers is ephemeral**
  - It is re-initialized each time the container is created
- **Using the 'Volume' support, file system areas can be mounted into the running instance**
  - This will expose files to the container
  - E.g. docker run –v /users/home/name:/users/homedir busybox
    - this maps the 'name' directory to the 'busybox' container image
    - Mounts /users/home/name in the docker host onto /users/homedir in the container
    - all changes will persist

  docker run –volumes-from container1 –name container2 busybox

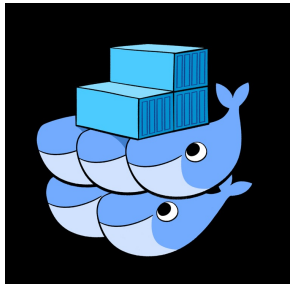
- Starts a 2nd container with the same mounted volumes as container1

Managing Container Run-time….

- **Resources can be allocated to containers**
  - To limit how much they can consume
- **The CPU usage of a container can be throttled**
  - Scheduler divides available CPU into 1024 time shares
  - These can be allocated to individual containers
  - only matters when CPU is busy
- **Memory**
  - swap space can also be constrained
    - E.g. a ‘stress test’ image
    - Limited to 512 units of cpu time
    - 128 Megabytes of memory

```
docker run –rm –ti progrium/stress –c 512 –vm-bytes 128M
```

Running on a Cluster – Docker Swarm



Run swarm manager on 1 node (in a container)

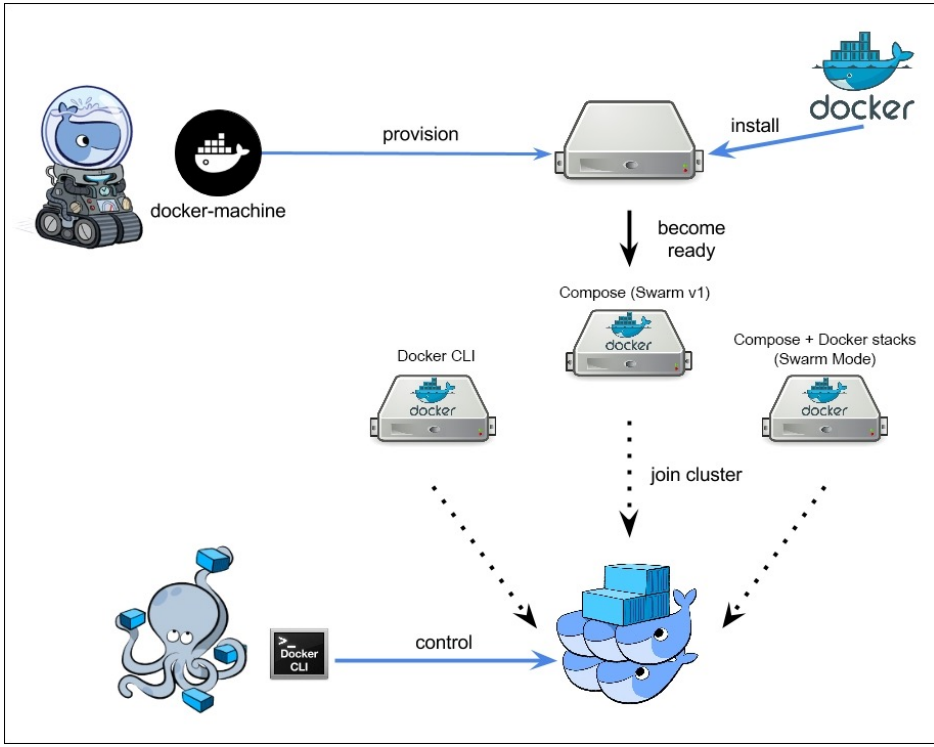
Run docker daemon on new nodes and then 'join' the swarm

```
docker pull swarm.. # on master node
docker run swarm create …
#on slave nodes
docker run swarm join –addr=172.17.42.10:2375….
docker ps   #list all containers on swarm cluster
```

Raspberry Pi Supercomputer:

- Building a supercomputer from a swarm of Raspberry Pis

https://www.raspberrypi.com/news/supercomputing-with-raspberry-pi-hackspace-41/

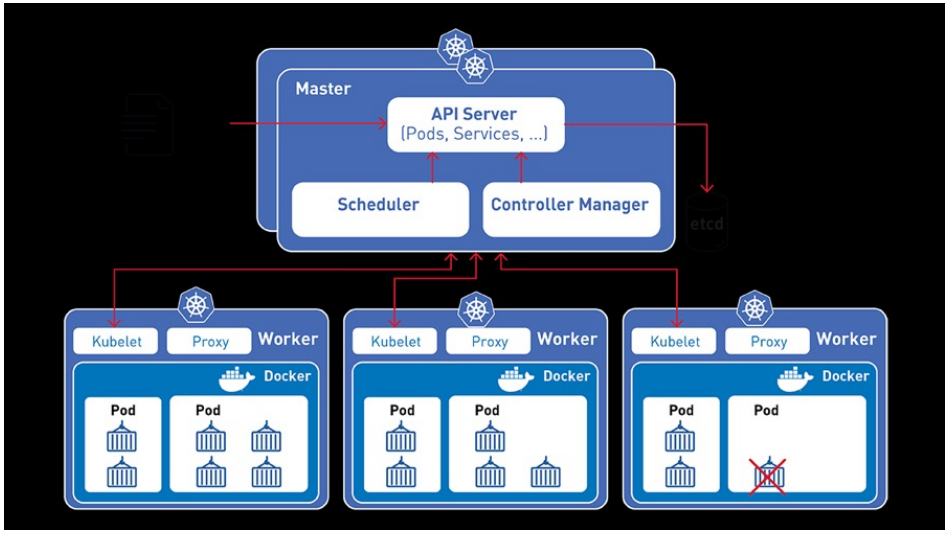


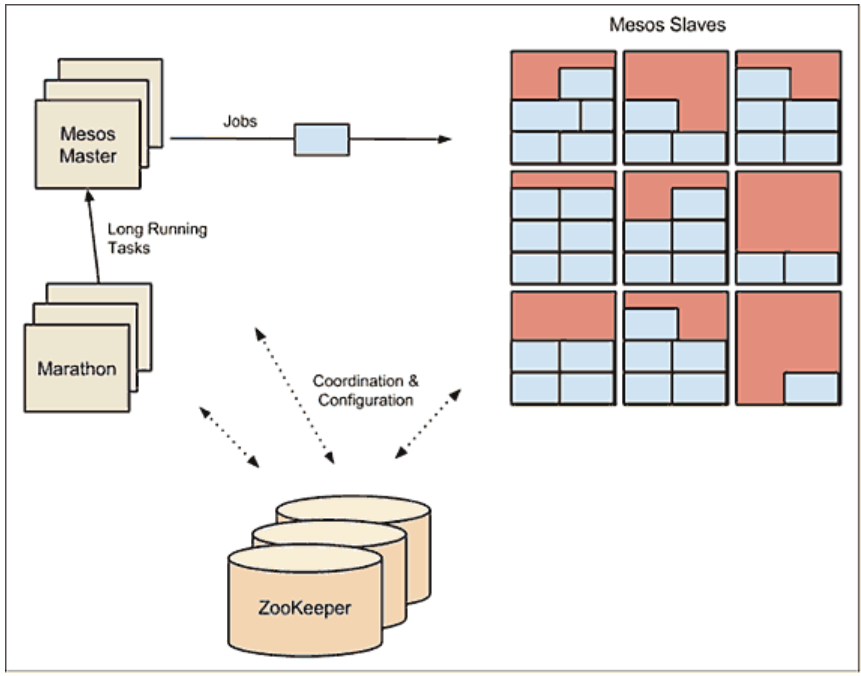
Container Orchestration Tools

- **Orchestration is the process of managing large complex configurations on running containers running on potentially very large clusters or multi-location cloud facilities**
- **Since around 2012 multiple companies and groups have been working on their own (families of) Orchestration Tools – these include**
  - Swarm from Docker
  - Fleet from CoreOS
  - Apache Mesos
  - Google Kubernetes
  - Helios from Spotify
- **Area is still very much in flux as the market tries to converge on a winner**
  - Google Kubernetes is leading

Orchestration Architectures

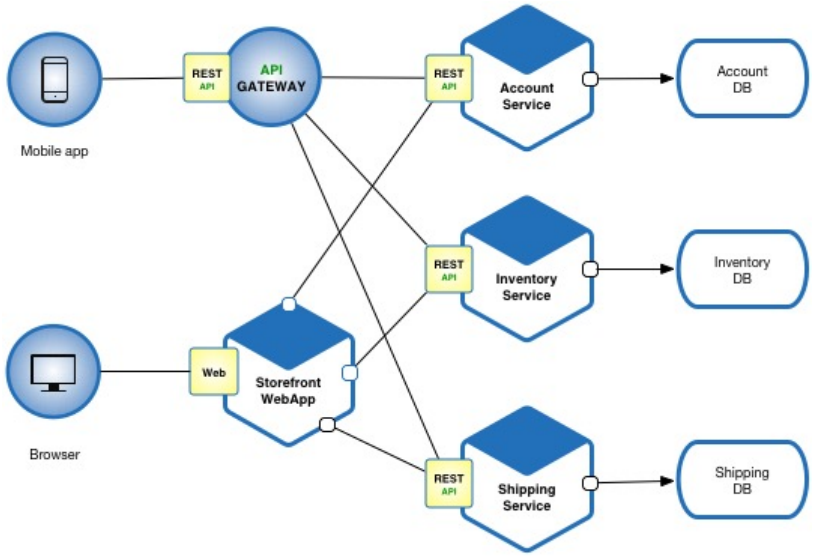
- **Vary in how they structure the hierarchies of containers**
- **Scheduling algorithms**
- **Packing schemes etc. etc.**
  - E.g.s Meso and Kubernetes





Microservices Architecture

- **Split entire application into distinct "Services" that communicate using REST APIs or other similar comms mechanisms**
- **Services can be**
  - in different languages
  - Use different technologies
  - Be owned by separate teams
  - Be independently testable
  - Easily scaled



AWS & Containers

- **Amazon AWS were a pioneer in VM-Based Cloud Computing and Support Services**
- **A bit late into containers, but catching up fast**
  - April 2015 launched Elastic Container Service
  - June 2018 launched Elastic Container Service for Kubernetes (EKS)
    - announced at RE:invent2017
  - 2018 launched AWS Fargate
    - containers without having to think (much) about the underlying cluster
    - Container-as-a-Service (CaaS)

AWS Elastic Container Service

- **First setup a Cluster specifying VM sizes, numbers, memory etc**
- **Describe the container configuration as a 'Task'**
- **Can use Elastic Load Balancers to share load across containers**
- **Can use a private container registry**
  - More control from a security perspective
- **Launch & View**
- **Pay for the underlying resources (VCPUs etc) used**

AWS Elastic Container Services for Kubernetes (EKS)

- **Create a cluster**
  - specifying machine types etc
- **Create a kubeconfig file specifying the container layout**
- **Deploy**
- **Charged for underlying AWS resources used**

AWS Fargate – Containers-as-a-Service

- **Define container configuration as a 'Task'**
- **Deploy on Fargate**
- **Pricing is based on vCPU and GB memory used from the "docker pull" command**

	Price
per vCPU per hour	$0.04048
per GB per hour	$0.004445

Comment from:https://www.trek10.com/blog/fargate-pricing-vs-ec2/
Break-even between Fargate & EC2 now happens in the 60-80% reservation rate, so if your cluster is only 50% utilized you might see a 10-20% cost *reduction* with Fargate! At the high end Fargate will increase your costs by 50-100% for a very tightly packed cluster with heavy EC2 Reserved Instances.