Container Technology

Dr Phill Edwards

phill.edwards@dr-phill-edwards.eu

April 27, 2025

DevOps

Micro–services

- Microservices are a variant of Service Oriented Architecture
- The architecture based on loosely coupled services
- Each service is fine grained
- Services communicate using lightweight protocols

Microservice Architecture

- Microservice architectures are by definition modular
- They support continuous delivery as only affected components need to be updated and redeployed
- Use fine grained interfaces
- Preferably they use RESTful Web services
- Lend themselves to container deployment and cloud applications

Microservice Characteristics

- Services typically communicate over a network, through they can use interprocess communication
- Each service can be independently deployed or replaced
- Services can be written in different languages
- Services can use different applications such as databases

UNIX

- The UNIX operating system was developed in the 1960s
- Linux and other UNIX variants are now running on everything from servers through mobile devices
- How has UNIX stood the test of time?
- It was built on the philosophy of many small components which
  - *Do one thing and do it well*

UNIX Representation Model

- The UNIX operating system standardized on two representation models:
  - Binary stream
  - Text stream (with Line End delimiters)
- Tooling supports these models:
  - `grep`
  - `awk`
  - `sed`
  - ...

Microservice Philosophy

- Similar to the UNIX philosophy *Do one thing and do it well*
- Services are small and perform a single function
- Services lend themselves to automated testing and deployment
- Each service should be flexible, resilient, minimal and complete

DevOps

Versioning

- It is often necessary to have different versions of operating systems
- Products often need to run on different environments
- It is costly to have separate machines for each version
- Multiple boot machines can help

Virtualization

- Software systems such as operating systems expect to have complete control of the hardware they are installed on
- Virtualization is when hardware is partitioned into independent parts
- Multiple systems can be installed into separate partitions and are isolated from each other
- The software which creates and manages the partitions is called a hypervisor
- Early mainframes used virtualization to run multiple processes

Type 1 Hypervisor

- A type 1 or bare metal hypervisor is installed directly on the hardware
- The hypervisor partitions the hardware and multiple operating system are run as processes
- High end servers use type 1 hypervisors
- Products include Oracle VM Server and VMware ESX/ESXi

Type 2 Hypervisor

- A type 2 or hosted hypervisor is installed as an application on a host operating system
- It often has to emulate hardware
- It may have to map guest operating system calls onto host operating system calls
- Used on desktop and laptop computers
- Products include VMware Player and Oracle VirtualBox

Hardware Virtualization

- The Intel 80368 CPU in protected mode could run multiple virtual 8068 environments
- Intel added Virtual Machine Extension (VMX) CPU instructions
- It allows Virtual Machine Monitor (VMM) software to control virtual machines
- Memory Management Unit (MMU) support allows virtual machines to access hardware directly
- Extended Page Table (EPT) support simplified memory paging
- CPUs with VMX, MMU and EPT allows type 2 hypervisors to become effectively type 1
- Extensions are enabled on Apple machines
- Extensions are disabled on PCs in the BIOS because of security issues
- PC extensions can be enabled by changing the BIOS security settings

Virtualization Issues

- Virtual machines contain full operating systems
- These have to be installed from ISO images
- Virtual machine files can be 10s of gigabytes
- It can be difficult to install virtual machines over networks using FTP or HTTP
- A solution is to use 7Zip to compress the files and break them into manageable chunks

Virtualization and Performance

- Virtual machines can be very slow
- Host machines need more memory to run them
- CPUs without virtualization extensions run virtual machines very slowly
- It is not possible for a 32 bit host to run a 64 bit guest without virtualization extensions
- The host operating system can greatly restrict the memory available to guests

DevOps

Containers

- Containers are lightweight computing contexts
  - Supported by system level virtualization technologies
  - Provide *operating system* virtualization
  - Usually contain a single application and its dependencies
  - Spin up or shut down in *milliseconds*
- Operating system virtualization:
  - Host computer has a single Linux kernel and processes shared by containers
  - Host computer control groups and name spaces used to limit resource use and isolate containers
- Containers are self contained file systems based on minimal Linux distributions

Container Implementations

- Containers can only run on Linux
- Container implementations create control groups and name spaces and run container images
- Linux Containers (LXC) is a set of low level tools which some other implementation are built upon
- LXD is a container hypervisor built on LXC with a REST API
- Open Container Initiative (OCI) sets container standards
  - Runtime Specification (runtime-spec)
  - Image Specification (image-spec)
  - Distribution Specification (distribution-spec)
- Containerd is a widely used OCI container runtime

Container Implementations

- *Docker* is a suite of tools for creating, managing and running containers
- *Podman* from RedHat is a more secure replacement for Docker
- Implementations use Containerd to run containers
- Implementations can run container images built by other implementations
- MacOS and Windows versions use a Virtual Machine running a Linux kernel

DevOps

Docker Architecture

- Docker runs containers and manages images on compute nodes
  - A node is either a computer or a virtual machine
- Docker uses a client–server architecture to manage containers
  - `docker` client communicates with Docker daemon
- The default communication mechanism is a UNIX socket
- TCP/IP communication can be enabled, except on OS X, by adding a startup option
- Docker also has tools for remote Docker node and container management

Docker Daemon

- Docker daemon is often referred to as just Docker
- Docker manages all image and container operations
- Docker runs on every node hosting containers
- Docker responds to request sent to a REST API via
  - UNIX socket `unix:///var/run/docker.sock`
  - TCP port 2375 plain text
  - TCP port 2376 TLS
- Docker creates virtual networks for container communication on a node

Docker Client

- Docker client is the command line tool `docker`
- It communicates with the Docker daemon via the REST API
- `docker` can communicate with remote daemons by setting environment variables
- `docker` has sub-commands for managing
  - images
  - containers
  - volumes
  - networks

Docker Machine

- Docker machine is the command line tool `docker-machine`
- It manages remote Docker daemons
- It can install, update and delete Docker daemon on remote machines
- It is installed on a single node and it manages other nodes

Docker Compose

- Docker compose is the command line tool `docker-compose`
- It manages groups of services which are containers
- It can build and update images and upgrade running containers
- It can start and stop multiple containers with a single command
- A YAML configuration file defines services and their configuration

Docker Swarm

- Docker Swarm creates and manages Docker clusters
- The swarm turns several nodes into a single virtual node
- A node becomes swarm master manually or by using Docker Machine
- Nodes can be added to or removed from the swarm
- Containers are run from the master using Docker client
- The swarm uses services which are containers
- Services can be scaled across swarm nodes

DevOps

Kubernetes

- Kubernetes is a container orchestration and management system
- It performs automatic deployment, scaling and operations on application containers
- It works on clusters of hosts
- It provides a container centric infrastructure
- It was started by Google in 2014, it is now at https://kubernetes.io
- Based on Google's Borg container management system
- Written in the Go programming language
- Open source project hosted on Github

Kubernetes

- Kubernetes schedules and runs containers on clusters of computers
  - Physical computers
  - Virtual computers
- It combines the simplicity of PaaS with the flexibility of IaaS
- It facilitates portability across infrastructure providers
- For developers:
  - Work in container-centric not host-centric infrastructure
  - Containers are consistent and hide potential host differences
- For operations:
  - Deployment and management of containers is based on configurations that are external to individual containers
  - Operations and management map Kubernetes abstractions to actual implementations
  - Or — rely on platform offerings such a Google Container Engine

Kubernetes Functions

- Co-locating helper processes
- Mounting storage systems
- Distributing secrets
- Application health checking
- Replicating application instances
- Horizontal auto scaling
- Naming and discovery

Kubernetes Functions

- Load balancing
- Rolling updates
- Resource monitoring
- Log access and ingestion
- Support for introspection and debugging
- Identification and authorization

Kubernetes is not

Kubernetes –

- Is not an all inclusive PaaS; PaaS can be build on top of Kubernetes
- Does not limit the types of supported applications
- Does not provide middleware such as messaging systems
- Does not deploy source code or build applications
- Does not force a choice of logging, monitoring, and alerting systems
- Does not provide computer configuration, maintenance, or management

DevOps

Serverless

- Kubernetes Clusters are not easy to set up and maintain
- Cloud providers provide Kubernetes management tools
- Kubernetes Clusters use virtual machines that incur billing costs
- Cloud provide serverless container execution
  - AWS Lambda, Azure Container Applications, Google Cloud Run
  - The number of containers scale by usage
  - They can scale to zero when not used at no billing cost

DevOps

Union File System

- A union file system is built up from layers
  - Foundation is a base layer of files and directories
  - One or more layers of files and directories are added on top
- The resultant file system is the union of the layers
- If two files have the same path a priority is used to determine which is visible

Docker Image

- A Docker image is a read–only union file system
- Foundation layer is a Base Image:
  - Carefully crafted minimal Linux file systems
  - Available for most common Linux distributions
- Layers can be added to the base image to build applications
- Prebuilt images are available which contain common applications on a range of base images

Docker Images

- Each Docker image is identified by a 64 digit hexadecimal ID number
  - Docker ID (pre-1.10 release) used a random UUID
  - Docker ID (1.10+ releases) SHA256 hash of image contents
    - ★ Abbreviated ID number usually sufficiently unique
- Image may also have one or more human readable tags
  - Describes content (e.g. alpine, debian, nginx)
  - May include version numbers (e.g. 2, 2.1, latest)
  - May also specify destination Docker Registry
- Image also has a size and a creation date

Docker image listing

```
REPOSITORY   TAG      IMAGE ID       CREATED          SIZE
alpine       3.6      a41a7446062d   40 hours ago     3.97MB
alpine       latest   a41a7446062d   40 hours ago     3.97MB
alpine       3.5      75b63e430bd1   40 hours ago     3.99MB
alpine       3.4      6008ce38ddc1   40 hours ago     4.81MB
```

Image Location

- Docker creates containers from a *local* copy of an image
- Local copies are:
  - Created locally, or
  - Pulled from a repository
- By default Docker pulls images from Docker Hub
- A `docker pull` downloads all layers and assembles the image

Pull an image by repository and optional label

```
docker pull centos
docker pull centos:latest
```

Managing Images

- Images are managed with the `docker images` command
- All Docker commands have a `--help` option
- Images can be removed using `docker rmi` by registry name or id
- Images with a name containing a tag other than latest can only be removed if the tag is specified

List and delete local images

```
docker images --help
docker images
docker rmi centos:latest
docker rmi 9910dc9f2ac0
```

Images and Containers

- A container comprises:
  - An image realized as a read–only union file system (UFS) of layers
  - A read–write layer is added on top of the UFS
  - An executing program
- Any file changes are captured in the read–write layer
- The modified UFS can be used to create a new image

DevOps

Running Containers

- Containers are run using the `docker run` command
- If the image does not exist locally a pull attempt will be made
- Interactive containers are created using the `-it` option
  - `-it`, `-ti` and `-i -t` are equivalent
- A command to run within the container is also specified

Run an interactive container

```
docker run -it centos /bin/bash
```

Running Interactive Containers

- Each container has an id and a name:
  - Container name set with the `--name` option
  - Docker will make up a name if not specified
- Containers are stopped when the command exits
- The `--rm` option deletes the container on exit
- Best practice is to use either
  - `--rm`, or
  - `--name`

Run a container with a name or delete on exit

```
docker run -it --rm centos /bin/bash
docker run -it --name centos centos /bin/bash
```

Running Daemon Containers

- Daemon containers are created with the `-d` option
- Best practice is to use the `--name` option

Run a daemon container with a name

```
docker run -d --name nginx nginx
fe2b9cc6a2de27389dbdbbecc9eb9232d46d8ee69fee1f15fe2cdad28b5bf245

docker ps
CONTAINER ID        ...STATUS              PORTS                NAMES
fe2b9cc6a2de           Up 3 seconds        8080/tcp             nginx
```

Executing Commands in Daemon Containers

- Commands can be run in daemon containers with the `docker exec` command
- When the command exits the container is still running

Execute a command in a running container

```
docker run -d --name nginx nginx
docker exec -it nginx /bin/bash
```

Running Daemon Containers

- Stopped containers listed with `docker ps -a` command

Run a daemon container with a name

```
docker stop nginx
nginx

docker ps
CONTAINER ID        ...STATUS              PORTS               NAMES
<empty>

docker ps -a
CONTAINER ID        ...STATUS                        PORTS               NAMES
fe2b9cc6a2de           Exited (137) 9 minutes ago                        nginx
```

Running Daemon Containers

- Stopped containers can be restarted with `docker start` command

Run a daemon container with a name

```
docker start nginx
nginx

docker ps
CONTAINER ID        ...STATUS              PORTS               NAMES
fe2b9cc6a2de           Up 2 seconds        8080/tcp            nginx
```

Managing Containers

- Stopped containers can be deleted using `docker rm`
- Running containers can be force deleted using `docker rm -f`
- Deleting a container destroys an changes in the read-write layer

Delete a container

```
docker run -d --name mysql mysql
docker rm -f mysql
```

Port Mapping

- Docker creates a private internal virtual network
- The virtual network is not accessible from the host
- Containers are connected to the internal network
- A container port is mapped to a host port using the `-p` option
- Connections to host's mapped port are redirected to the container port

Map host port 2022 to container port 22

```
docker run -d --name sshd -p 2022:22 sshd
```

Host Naming

- Docker allows containers to be given host names
- Other containers can be linked to a container with a host name
- The name or id of the container is specified to link
- The host name gets added to the linked containers' `/etc/hosts`

Specify and link a host name

```
docker run -d --name rabbit --host rabbit rabbitmq
docker run -it --rm --link rabbit centos /bin/bash
```

Volumes

- Host directories or files are mounted into a container using the `-v` option
- Host file or directory must be specified with absolute path
- Container mount point will be created if it does not already exist

Map a host directory into a container

```
docker run -it --rm -v $PWD/software:/opt/software centos /bin/bash
```

DevOps

Docker Nginx Exercise

- Pull an nginx web server image
- Run a container exposing port 80 as port 8080
- Load the web site using a browser or `curl`
- Execute a shell in the container
- Edit the web page `/usr/share/nginx/html/index.html`
- Load the web site using a browser or `curl`
- Save the container to the image nginx:v1
- Remove the container
- Start an nginx:latest container and see that the changes were lost
- Start an nginx:v1 container and see that the changes have been saved
- Full instructions are in `KPMG/README.md`

DevOps

Registries

- A Registry Service stores Docker images
- Public registries are available from various providers
  - Docker Hub https://hub.docker.com is the default public registry
    - ★ Hosts official images of Linux distributions and applications
    - ★ Any registered user can add images
- Private registries are also available
  - Docker Hub has optional (for fee) private registry
  - Google Cloud Platform (GCP) allows Docker registries to be created in Artifact Registry
  - Amazon Elastic Container Registry (Amazon ECR) - AWS
  - Azure Container Registry

Repositories

- Each registry contains a number of repositories
- A repository contains different versions of the same image
- Repository names often contain the base Linux image and the application `centos-ssh`
- Each image in a repository has a different tag - usually a version number or `latest`
- If no tag is specified `latest` is the default
- The latest CentOS with ssh image could be `centos-ssh:latest`

Labels

- Every image is uniquely identified by a label which looks like a URI
- Labels have the registry URI which defaults to https://hub.docker.com
- They have a name space _ is Docker Hub, `r` is user
- Podman prefixes Docker Hub images with the namespace `docker.io/`
- They have the repository name space
- They have a tag which can be a version or a label

Example labels

```
https://hub.docker.com/_/centos:latest
https://hub.docker.com/r/myrepo/centos-ssh:latest
https://gcr.io/project_id/centos-ssh:1.0.0
```

Commiting Containers

- Images can be created from containers
- Start an interactive container from a base image
- Manually install software
- Commit the image

Commit a container

```
docker run -it --name centos centos /bin/bash
yum install -y which
yum install -y net-tools
exit
docker commit centos centos-net:latest
docker rm centos
```

Registry Images

- Images must have the same name as the repository they are to be stored in
- If an image has the wrong name can create a container and commit it
- Can use `docker create` instead of `docker run` to create a stopped container
- Can also use `docker tag` to create an image from another

Example labels

```
docker create --name centos centos /bin/bash
docker commit centos repo/centos:latest
docker rm centos
docker tag centos repo/centos:latest
```

Checking Images

- `docker inspect` command provides image and container details
- Create an interactive container from built images to check contents

Inspect image and run interactively

```
docker inspect centos-java
docker run -it --rm centos-java /bin/bash
```

Automated Docker Builds

- Automated Docker builds with `docker build` command
- Create a directory to serve as build context
- Add a `Dockerfile` containing sequence of build steps
- Copy any files destined for image into build context root directory
- Do not put any unnecessary files in the context as they all get uploaded to the daemon

Dockerfile

- Several Dockerfile commands take arguments
- Some arguments can be specified in either command or JSON form
- The two form of arguments can often have different effects
- Command form often allows shell wildcards * ? []
- JSON arguments take the form `["cmd", "-a", "file"]`
- Docker documentation prefers the JSON form

Dockerfile FROM

- First line must contain the `FROM` command specifying the base image
- If no tags is specified the `:latest` will be used
- Images often have multiple tags for the same image
- Subsequent commands are executed in a container which is then committed as a layer

Dockerfile

```
FROM centos:7
```

Dockerfile LABEL

- Optional `LABEL` commands can be specfied
- Labels are key value pairs
- Multiple labels can be specified
- Labels can be seen using `docker inspect`
- The `maintainer` label replaces the deprecated `MAINTAINER` command

Dockerfile

```
LABEL maintainer=user@email.address decription="A messaging system"
```

Dockerfile RUN

- The `RUN` command executes a command
- Commands must not requre user input
- Multiple commands should be executed as one using `&&`
- Remember each command creates a new image layer

Dockerfile

```
RUN yum install -y which && \
    yum install -y net-tools
```

Dockerfile ADD and COPY

- `ADD` copies files from the build context and remote URLs into the container
- `COPY` copies files from the build context into the container
- Source file specifier may include shell wildcards ? * []
- Commands specify a destination directory in the container:
  - The destination directory must end with a /
  - Destination directory is created if it does not exist
  - TAR files and compressed tar files are unpacked automatically

Dockerfile

```
COPY sshd.config /etc/sshd/
ADD apache-maven*.tar.gz /opt/
```

Dockerfile ENV

- `ENV` sets global environment variable for the container
- It has two forms for setting name-value pairs
- The form using = allows multiple variables to be set
- Environment variables can be added or changed using `docker run -e key=value`

Dockerfile

```
ENV JAVA_HOME /usr/java/latest
ENV EDITOR=vi TZ=Europe/Paris
```

Dockerfile CMD and ENTRYPOINT

- `CMD` and `ENTRYPOINT` specify an initial program to run when container starts
- Only one of each can be specified - the latest overrides
  - The container stops when the initial program terminates
  - The initial program must be a foreground process so it does not exit
  - Use a shell script to start multiple programs within container

Dockerfile

```
CMD /bin/bash
```

Dockerfile CMD and ENTRYPOINT

- `CMD` can specify a program or arguments to `ENTRYPOINT`
- It can be overriden by command arguments to `docker run`
- The shell command form of `ENTRYPOINT` disables an `CMD` or `docker run` commands
- it can be overriden using `docker run -entrypoint command`

Dockerfile EXPOSE

- `EXPOSE` simply documents which ports are to be listened on
- The port still need to be mapped to host ports using `docker run -p`
- All exposed ports can be exposed on random ports on the host using `docker run -P` - rarely used

Dockerfile

```
EXPOSE 22 8080
```

Dockerfile VOLUME

- `VOLUME` specifies a mount point for a volume
- The actual volume should be specified using `docker run -v`
- If no volume is specified when running a new volume will be created and mounted every time the container is run

Dockerfile

```
VOLUME /opt
```

Dockerfile USER

- `USER` specifies the user or UID the program will run as

Dockerfile

```
USER git
```

Dockerfile WORKDIR

- `WORKDIR` specifies the working directory for commands
- It applies to subsequent RUN, CMD, ENTRYPOINT, COPY and ADD commands
- It can be used multiple times in a Dockerfile
- The last WORKDIR becomes the default container working directory it can be overriden using `-w`

Dockerfile

```
WORKDIR /user/git
```

Working Directory

```
docker run -it --rm -v $PWD:/app -w /app alpine:latest sh
```

Dockerfile ARG

- `ARG` specifies an argument which is specified on the build command line
- It can be given a default value in case none is specified
- It specifies a variable which can be dereferenced
- The value is specified using the `--build-arg` argument

Dockerfile

```
ARG user1
ARG user2=guest
USER $user1

docker build --build-arg user1=apache -t centos:apache .
```

Dockerfile ONBUILD

- `ARG` specifies commands to be run if the image is used as a base image
- The cpmmands are triggers which get executed when the image if loaded using a Dockerfile `FROM` command

Dockerfile

```
ONBUILD RUN apt-get install -y net-tools
```

Dockerfile STOPSIGNAL

- `STOPSIGNAL` specifies the signal name or number which will be sent to terminate the container

Dockerfile

```
STOPSIGNAL SIGTERM
STOPSIGNAL 9
```

Dockerfile HEALTHCHECK

- `HEALTHCHECK` specifies a command which determines the health of the container
- It can have options - the defaults are shown
  - `--interval=30s` the interval between healthcheck runs
  - `--timeout=30s` timeout for healthcheck
  - `--retries=3` number of retries
- Container is healthy if the return status is 0
- Healthcheck failes if return status is 1 or timeout
- Container is unhealth after `retries` failures
- Only the last `HEALTHCHECK` is used `HEALTHCHECK NONE` disables

Dockerfile

```
HEALTHCHECK --interval=5m --timeout=3s get_status || exit 1
```

Dockerfile SHELL

- `SHELL` overrides the default shell for subsequent command form of commands
- It must use the JSON format - the Linux and Windows defaults are shown
- There can be multiple `SHELL` commands

Dockerfile

```
SHELL ["/bin/sh", "-c"]
SHELL ["cmd", "/S", "/C"]
```

Building Steps

- Create a context directory, add a Dockerfile, add other required files
- Tag image with destination repository name
- Run `docker build` command:
  - Each command in the Dockerfile creates a container and a layer
  - Each build steps' resulting images are cached
  - If the build fails, fix Dockerfile and re–run `docker build`
  - Build restarts with image of last successful build step
    - ★ Caching dramatically speeds up build process
    - ★ Successful file downloads from failed builds are not downloaded again as the cached file is used

Docker build using the java build context directory

```
docker build -t centos-java:latest java
```

Image Size

- Small Docker images are preferred
  - Speeds `push` and `pull` operations
  - Comparisons
    - Alpine Linux image is `4.8 MB`
    - CentOS Linux image is `197 MB`
- Tips:
  - `yum` and `apt-get` can pull in unwanted dependencies
  - Install software from tar archives instead of packages
  - Reduce the number of `RUN` command (use `&&` instead)

Docker Registry Automatic Builds

- Docker Registries offer and may require automated builds
- Each registry repository is associated with a Git respository.
- Git repository contains a complete build context
- Docker registry monitors Git repository
  - A commit to Git repository triggers a Docker repository image build

Image Configuration

- Images may accept run–time configuration for fine–tuning
  - A database may need schema and user configuration
  - Initial script file might accept parameters for customization
- External information may be passed to the container context via environment variables

Run container passing configuration environment variable

```
docker run -d -p 3306:3306 \
           --name mysql \
           -e MYSQL_DATABASE=mydb \
           -e MYSQL_USER=dba \
           -e MYSQL_PASSWORD=pw mysql
```

Docker Flask Exercise

- Go to the `KPMG/Flask` directory
- Look through the files for a Python Flask Web Server
- Build a Docker image called `pyserver:latest`
- Run a Docker container that maps port 8080 to port 8080
- Access the web pages
- Full instructions are in `KPMG/README.md`

DevOps

Monolithic Applications

- Monolithic Applications share a single Web Server and other resources
- They have a common base URL
- Component code makes assumptions about their environment
- Containerising components can lead to unexpected issues

Container Lifetime

- Monolithic Applications often run indefinitely
- Managed containers are started and stopped based on usage
- Container state must be in persistent storage
- Containers may need a caching system such as Redis

URLs

- Some applications use absolute URLs
- A container can't determine its base URL from its Web server
- Base URLs need to be passed through environment variables

CORS

- Web sites often use Javascript functions to fetch content
- Content may be on a different domain to the web content
- Javascript is easily modified and can be redirected to a fake site
- Cross-Origin Resource Sharing (CORS) provides a solution to these issues
- Web browsers will block Javascript requests to untrusted domains
- CORS requires request and response header to establish trust

Basic CORS

- A Web site has a domain `www.example.com`
- Its Javascript fetches data from a different domain `api.example.com`
- Only readonly GET and OPTIONS requests are allowed here
- The request must set a header `Origin: https://www.example.com`
- The response must set a header `Access-Control-Allow-Origin: https://www.example.com`
- Don't use `Access-Control-Allow-Origin: *`

More CORS

- If Javascript makes a cross domain POST, PUT, or DELETE request it will change server content
- The browser will perform a preflight check
- This is an OPTIONS request with headers Origin and `Access-Control-Request-Method: PUT`
- The response must set a header `Access-Control-Allow-Origin: https://www.example.com`
- It must also set allowable methods `Access-Control-Allow-Methods: GET, OPTIONS, POST, PUT`
- The actual request will be blocked unless the preflight check passes

Exposing Containers

- Containers can only expose a unique port number to the world
- Mapping port numbers to services is messy
- Containers share an internal network with unique hostnames
- A container can communicate with other containers using their hostname using the same port

Reverse Proxy

- A reverse proxy is a web server container
- All requests go to the proxy
- They are forwarded to other containers based on URL paths
- The proxy passes responses back to the caller

Docker Compose

- Docker Compose enables multiple containers to be run together
- They are given hostnames to communicate with each other
- They are defined in `compose.yml`
- Run containers `docker compose up -d`
- Stop containers `docker compose down`

Reverse Proxy Exercise

- Go to the `KPMG/Nginx` directory
- Look through the files for an Nginx reverse proxy
- Build a Docker image called `nginx:proxy`
- Go to the `KPMG` directory
- Look at the `compose.yml` file
- Run the containers
- Access `http://localhost:8080/flask/static/index.html`
- What happens with `http://localhost:8080/flask/`?
- Full instructions are in `KPMG/README.md`

Summary

- What happened at the end of the exercise and why?
- What are the challenges moving from monothic to containers?
- Any questions?

Thank you for attending