Welcome to the CoGrammar

Version Control and Project Management

The session will start shortly...

Questions? Drop them in the chat. We'll have dedicated moderators answering questions.

CoGrammar

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

Docker and Containerization

October 2024

Session Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly. **(Fundamental British Values: Mutual Respect and Tolerance)**
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Academic Sessions. You can submit these questions here: **Questions**

Session Housekeeping cont.

- For all **non-academic questions**, please submit a query: **www.hyperiondev.com/support**
- Report a **safeguarding** incident: **www.hyperiondev.com/safeguardreporting**
- We would love your **feedback** on lectures: **Feedback on Workshops**

Agenda

- Overview
  - Understand how docker containers work
  - Understand how docker images are created
  - Understand the benefits of using containerization

Running an image

- Download the image from docker hub or another registry
- Run the image

```
docker pull nginx
```

```
docker run nginx
```

Understanding a container

CoGrammar

Containers

- Purpose
  - Running services locally without having to install them on the main OS
  - Allows us to isolate services that would otherwise have conflicts if they were run on the same host system
- How do they work
  - A container starts with a lightweight VM
  - The VM will have the service that needs to be run
  - The service only exists within that VM and to access it, we need to change the networking rules.
  - When we perform the `docker run` command, the VM and it's services are spun up

Interactive Terminal

- Every docker container runs some underlying OS
- We can navigate through the OS using the interactive terminal
- Within the OS, we're able to perform normal operations that we would in a regular Linux OS system.
- The `-it` flag is used to enter the interactive terminal, but it doesn't work on all images
  - After specifying the image name, we need to specify the terminal the OS uses, alpine uses ash

```
docker run -it alpine ash
```

CoGrammar

Infrastructure Overview

- Computer systems follow this architecture
- The sections in blue are not the users concern, everything above is
- Docker handles a form of virtualization allowing us to run different operating systems on a single machine
- To create a service using docker, we need to setup the OS and all of the other tools that sit on top of the OS

Example overview

- We ran the Alpine OS
- Within Alpine, we installed the Python runtime
- Once the runtime was in the system, we were able to run the Python application

Building a Docker Image

CoGrammar

Core of a Docker image

- How does a docker container work
  - A Docker container is essentially a VM that runs a specific service or a specific set of services
- Previous experience
  - We saw that we're able to run a docker OS container
  - We can install tools to that container and do what we need to do
- Problem from our experience
  - There are a lot of steps that need to be performed to run a service in a container
- How do docker images solve the problem
  - Using a Dockerfile, we can write the commands required to setup our services
  - We can generate an image from the Dockerfile which will act like a .exe file with our predefined configuration present
  - When we run this image, our service will go live and have everything we need

Core of a Docker image

- The Dockerfile specifies the services that we're using and some configurations that need to be performed
- ***From*** - Specifies the base image
  - Every Docker image requires a base image, everything we do will build on the base image.
- ***Run*** - Specifies the terminal command that needs to be run
  - Usually used for installing services in the Linux environment
  - Can be used to install other dependencies depending on the tools you're using

```
1 FROM alpine
2 RUN apk add vim
3 RUN apk add python
```

Core of a Docker image

```
docker build -t alpine-vim-python .
```

- To create an image, you need to run the ***build*** command
- ***build*** -
  - Tells the docker engine the build command is being used
  - Allows us to create an image from a Dockerfile
- ***-t*** -
  - Used to give the image a name
  - This is important if you want to reference the image locally
- ***alpine-vim-python*** -
  - This is the name provided for the image
  - Goes with the -t flag
  - Can be any name, if we build the image again using the same name, the old one will be overwritten
- **.** -
  - Specifies the directory where the Dockerfile is located
  - We are running the command in the current directory

Core of a Docker image

- When running the custom image in the interactive terminal, we can see that Python and Vim are already installed
- We're able to perform the operations that we need to perform without having to do any extra operations

CoGrammar

Q & A SECTION

Please use this time to ask any questions relating to the topic, should you have any.

Building a Docker Images for applications

CoGrammar

Docker for Applications

- Docker is usually used to run applications
- For an application to run, we need a runtime
- There are many images that we can use that come with the runtime preconfigured that we can use as our base images.
- We can also build on top of our own images

```
1 FROM alpine-python-vim
2 WORKDIR /app
3 COPY . .
```

Docker for Applications

- **WORKDIR -**
  - Creates a folder inside the docker environment and runs all of the commands that follow within that folder
  - The last WORKDIR that is set will be the default entry point when using the ***-it***
- **COPY**
  - Takes the files from the local machine and moves them to the workdir
  - First . references the local machine and the second the workdir

```
1 FROM alpine-python-vim
2 WORKDIR /app
3 COPY . .
```

CoGrammar

Docker for Applications

- **Interactive terminal**
  - All of the files from the project are moved to the VM
  - We can run the application because the Python runtime is installed
- **Possible improvements**
  - We still need to go into the terminal to run the application
  - It would be nice if we could just run **docker run** without having to go into the container.
- **Running an application**
  - Applications are usually run based on a given command

```
FROM alpine-python
WORKDIR /app/
COPY ./main.py /app/
RUN python3 main.py
```

Docker for Applications

- **Results**
  - When we do a **docker run** that does not change the outcome
- **Reason**
  - The **RUN** command is called when the image is being built, it's a command that's meant to create an image and does not exist within the container
- **CMD**
  - This command will be run when the image is run
  - Only one CMD script will be run in a dockerfile

```
FROM alpine-python
WORKDIR /app/
COPY ./main.py /app/
CMD python3 main.py
```

Docker for Applications

- **Web application**
  - We can use the node image to run a node application using docker
  - We're only taking the files we need with the **COPY** command
  - We're using the **RUN** command to install the packages
  - When the container is run, the **run start** command should run the web application

```
FROM node:22-alpine3.19
WORKDIR /app
COPY package.json .
COPY server.js .
RUN npm i
CMD run start
```

Docker for Applications

- **Results**
  - **docker run** runs the application, but we can't access the web application
  - **docker run -p PORT:PORT** allows us to access the application

```
docker run -p 9000:9000 tipp_site
```

Docker for Applications

- A docker container is an isolated environment
- Anything that runs in a docker container does not have access ot the outside world
- The outside world does not have access to the stuff in the docker container
- **-p**
  - The flag allows us to map the internal port to an external port allowing us to access the service from outside the container
  - local:container
    - The port on the left is the external port on the main machine
    - The port on the right is the port that is being used in the contianer

Docker for Applications

- **EXPOSE**
  - Specifies any ports that the container will use
  - It's not required, you can still access the ports without this in the dockerfile
  - More useful for a **docker compose**

```
FROM node:22-alpine3.19
WORKDIR /app
COPY . .
RUN npm i
EXPOSE 9000
CMD run start
```

Docker for Applications

- **.dockerignore**
  - Instead of copying a single file, we can use a .dockerignore to exclude specific files from being passed to our container

```
.git
node_modules
Dockerfile
.env
```

```
FROM node:22-alpine3.19
WORKDIR /app
COPY . .
RUN npm i
EXPOSE 9000
CMD run start
```

Managing Environment Variables

CoGrammar

Environment Variables

- The dockerfile is used for configuring the application
- Sometimes we need custom configurations to do certain things
- Environment variables allow us to pass values that can be passed beyond the building of the image allowing for custom configurations

Environment Variables

Passing Variables on run

- docker run **-e PORT=9000** <image-name>
  - Assigns a PORT environment variable to the container environment, it can be accessed by any service running in the container
- Prefered method for defining variables
  - You don't want to have your secrets in the dockerfile when you build the image as they can be accessed by anyone with the built image
  - This approach allows different container instances to have their own variables for the same image
- Passing more than 1 variable
  - docker run **-e PORT=9000 -e DB_NAME=postgres** <image-name>
  - docker run **–env-file .env** <image-name>

Environment Variables

Dockerfile

- Benefits
  - We can set default values for variables that are required for the application and still allow the user to set their own when they run the container
- Cons
  - Terrible place to store secrets because the docker image inspect command will show the values for the variables

```
FROM node:22-alpine3.19
ENV PORT=9000
ENV DB_HOST=localhost
WORKDIR /app
COPY . .
RUN npm i
EXPOSE 9000
CMD run start
```

Commands and Tips

CoGrammar

Command and Tips

Overview

- Docker has a standard formula for running commands
  - docker <resource-type> <command> <resource>
    - docker image inspect alpine
    - docker container inspect my-container
    - docker volume inspect my-volume
- Help is never far away, if you type a command and resource type, you'll get a list of commands an their descriptions
  - docker <resource-type>
    - docker image
    - docker container
- There are short hands for some resource type specific stuff

Command and Tips

Shared commands

- ls - (lower case L, s)
  - Lists the created/available resources
  - Docker <resource-type> ls
- rm
  - Removes a specific resource
  - docker <resource-type> rm <resource-name/id>
- inspect
  - Shows details about the resource
  - docker <resource-type> inspect <resource-name/id>
- prune
  - Removes all unused resources
  - docker <resource-type> prune

CoGrammar

Image Commands

- Run **docker image**

```
Commands:
  build       Build an image from a Dockerfile
  history     Show the history of an image
  import      Import the contents from a tarball to create a filesystem image
  inspect     Display detailed information on one or more images
  load        Load an image from a tar archive or STDIN
  ls          List images
  prune       Remove unused images
  pull        Pull an image or a repository from a registry
  push        Push an image or a repository to a registry
  rm          Remove one or more images
  save        Save one or more images to a tar archive (streamed to STDOUT by default)
  tag         Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
```

Container Commands

- Run **docker container**

```
Commands:
  attach      Attach local standard input, output, and error streams to a running container
  commit      Create a new image from a container's changes
  cp          Copy files/folders between a container and the local filesystem
  create      Create a new container
  diff        Inspect changes to files or directories on a container's filesystem
  exec        Run a command in a running container
  export      Export a container's filesystem as a tar archive
  inspect     Display detailed information on one or more containers
  kill        Kill one or more running containers
  logs        Fetch the logs of a container
  ls          List containers
  pause       Pause all processes within one or more containers
  port        List port mappings or a specific mapping for the container
  prune       Remove all stopped containers
  rename      Rename a container
  restart     Restart one or more containers
  rm          Remove one or more containers
  run         Run a command in a new container
  start       Start one or more stopped containers
  stats       Display a live stream of container(s) resource usage statistics
  stop        Stop one or more running containers
  top         Display the running processes of a container
  unpause     Unpause all processes within one or more containers
  update      Update configuration of one or more containers
  wait        Block until one or more containers stop, then print their exit codes
```

Important Resources

CoGrammar

Helpful Resources

- https://docker-curriculum.com/#introduction
- https://docs.docker.com/get-started/
- https://cognitiveclass.ai/courses/docker-essentials
- https://labs.play-with-docker.com/

CoGrammar


CoGrammar
SKILLS FOR LIFE
SKILLS BOOTCAMPS
Department for Education
We will be back soon...
Feel free to leave your questions in the questions section