Docker and Kubernetes

Kaavya Antony

2020

1. Dive into Docker!

Why use Docker?

· goal of Docker - make it easy to install software on any computer
· example command: **docker run -it <software>**

What is Docker?

· an ecosystem around creating or running containers
· image - single file with all dependencies and configs required to run a program
· container - instance of an image, runs a program

Docker for Windows/MAC

· Docker Client (CLI) - tool that we issue commands to
· Docker Server (Docker Daemon) - tool responsible for creating images, running containers, etc.

Using the Docker Client

· example command: **docker run -it hello-world**
· the server checks the image cache first to see if an image already exists on your local machine, if not, the server checks Docker Hub to download the image to your local cache
· the server then loads the image and creates a container out of the image to then run the program
· if we re-run **docker run -it hello-world**, the image can now be found in our image cache

But Really... What's a Container?

· container - a process or a set of processes that have a grouping of resources specifically assigned to it, a "slice" of a computer
· the flow - processes running on your computer → running program issues request to kernel to interact with a piece of hardware → kernel requests piece of hardware for processes
· namespacing - segmenting a hardware resource per process or group of processes
· control groups (cgroups) - limit the amount of resources used per processes

How's Docker Running on your Computer?

· namespacing and control groups belong to Linux
· Docker runs on MAC and Windows by using a Linux virtual machine

2. Manipulating Containers with the Docker Client

Docker Commands

· running a docker image overriding default command - **docker run <imagename> <command>**
  – Example: **docker run busybox echo hi there**
· docker run = docker create + docker start
  – docker create - setting up the file system snapshot (returns the ID of the container created)
  – docker start - execute the startup command
  – **docker start -a <container id>** - the 'a' flag tells Docker to watch for output and print it to the terminal
· restart a Docker container - **docker start -a <container id>** but this cannot replace the default command
· clear out all stopped Docker containers - **docker system prune**
· show all Docker containers - **docker ps –all**
· get logs from a container - **docker logs <container id>** this does not re-start or re-run the container, just outputs the logs
· how to stop a container:
  – **docker stop <container id>**
    – a hardware signal is sent to the primary process inside that container
    – SIGTERM - gives the process time to shut itself down and do some clean up
  – **docker kill <container id>**
    – a kill signal is sent to the primary process inside that container
    – SIGKILL - shut down immediately, no clean up
  – stop takes a little more time because it performs clean up, but this is recommended as a first try

Executing Commands in Running Containers

– docker exec - execute an additional command inside a container
– -it - allows us to provide input to the container
· **docker exec -it <container id> < command >**
  – Example: **docker exec -it 093asdlakjf345243 redis-cli**

The Purpose of the IT Flag

· -it = -i -t
  – -i - attaches the terminal to STDIN, so that the user can type in input into the

terminal
  – -t - responsible for terminal output formatting, so we get readable output

Getting a Command Prompt in a Container

· **docker exec -t <container id> sh** - gets shell/terminal access to your container, now you can run all shell commands inside the context of a container
· sh - command processor, allows commands typed in terminal to be executed

Starting with a Shell

· **docker run -it <container id> sh** - not recommended, better to use docker exec

Container Isolation

· two containers do not automatically share their file system
· for example, if you run **docker run -it busybox sh** and create a file named *script.py* in there, a second run of **docker run -it busybox sh** will not show *script.py*

3. Building Custom Images Through Docker Server

Creating Docker Images

· Dockerfile - a configuration to define how our container should behave
· Dockerfile → Docker Client → Docker Server → Usable Image

Creating a Dockerfile

· specify a base image → run some commands to install additional programs → specify a command to run on container startup

Dockerfile Teardown

Instruction telling Docker Server What to Do	Argument to the Instruction
FROM	alpine
RUN	apk add –update redis
CMD	[“redis-server”]

What’s a Base Image?

· a base image is an initial starting point/set of programs that we can use to further customize our image

The Build Process in Detail

· the build command creates an image from the Dockerfile
· FROM - downloads the initial starting image from local cache of Docker Hub to use as a base image
· CMD - tells you what the default startup command is

Rebuilds with Cache

· when instructions are repeated, the image from the previous Docker build gets used
  * but only if the order of operations in the Dockerfile remains the same

Tagging an Image

· **docker build -t <tag name> .**
· tag name = your Docker ID / repo name: version
  Ex.) kantony/redis:latest
· now we don't have to use the container ID when referencing the container
  Ex.) **docker run kantony/redis**
    * don't need the version tag

4. Making Real Projects with Docker

Example Project Outline

1. Create web app
2. Create a Dockerfile
3. Build image from Dockerfile
4. Run image as container
5. Connect to web app from a browswer

Docker Run with Port Mapping

· **docker run -p <port a>:<port b> <image id>**
· incoming requests to port a on local hosts will be forwarded to port b inside the container
  * port a and port b do not have to be the same

5. Docker Compose with Multiple Local Containers

Introducing Docker Compose

· Docker Compose
  · separate CLI that gets installed along with Docker
  · used to start up multiple containers at the same time
  · automates some of the long-winded arguments we were passing into **docker run**

- docker-compose.yml
  - contains all the options we'd normally pass to docker-cli

Docker Compose Commands

- docker run myimage → docker-compose up
- docker build . + docker run myimage → docker-compose up –build

Stopping Docker Compose Commands

- launch containers in the background - **docker-compose up -d**
- stop containers - **docker-compose down**

Container Maintenance with Compose

- status codes:
  - 0 = exited without issue
  - 1, 2, 3, etc = exited with some issue
- Restart policies:
  - "no" - default, never attempt to restart this if container stops or crashes
  - always - restart for any reason
  - on-failure - only restart if the container stops with an error code (not 0)
  - unless-stopped - always restart unless forcibly stopped