Docker and Kubernetes

Session 5

Making a real project with Docker and NodeJs

Agenda

- Assignment Review & Guides
- Keep container alive during daemon downtime
- Run multiple service in a container
- Container logging
- Docker compose introduction
- Project outline

Keep containers alive during daemon downtime

By default, when the Docker daemon terminates, it **shuts down** running containers. You can configure the daemon so that containers remain running if the daemon becomes unavailable.

This functionality is called **live restore**. The live restore option helps reduce container downtime due to daemon crashes, planned outages, or upgrades

Enable live restore

There are two ways to enable the **live restore** setting to keep containers alive when the daemon becomes unavailable

- Add the configuration to the daemon configuration file. On Linux, this defaults to **/etc/docker/daemon.json**

```
{
    "live-restore":true
}
```

- Start the dockerd process manually with the **--live-restore** flag

Run multiple service in a container

A container's main running process is the **ENTRYPOINT** and/or **CMD** at the end of the Dockerfile. It is generally recommended that you separate areas of concern by using one service per container

If you need to run more than one service within a container, you can accomplish this in a few different ways.

- Put all of your commands in a wrapper script, then run the wrapper script as your CMD
- Use bash's job control
- Use a process manager like supervisord

Using wrapper script

The Dockerfile and my_wrapper_script:

```
FROM ubuntu:latest
COPY my_first_process my_first_process
COPY my_second_process my_second_process
COPY my_wrapper_script.sh my_wrapper_script.sh
CMD ./my_wrapper_script.sh
```

```
#!/bin/bash
# Start the first process
./my_first_process -D
status=$?
if [ $status -ne 0 ]; then
    echo "Failed to start my_first_process: $status"
    exit $status
fi
# Start the second process
./my_second_process -D
status=$?
if [ $status -ne 0 ]; then
     echo "Failed to start my_second_process: $status"
     exit $status
fi

while sleep 60; do
   ps aux |grep my_first_process |grep -q -v grep

   PROCESS_1_STATUS=$?
   ps aux |grep my_second_process |grep -q -v grep

   PROCESS_2_STATUS=$?
   if [ $PROCESS_1_STATUS  -ne 0 -o $PROCESS_2_STATUS  -ne 0 ]; then
       echo "One of the processes has already exited."
       exit 1
   fi
done
```

Use bash's job control

The Dockerfile and my_wrapper_script:

```
FROM ubuntu:latest
COPY my_main_process my_main_process
COPY my_helper_process my_helper_process
COPY my_wrapper_script.sh my_wrapper_script.sh
CMD ./my_wrapper_script.sh
```

```
#!/bin/bash

# turn on bash's job control
 set -m

# Start the primary process and put it in the background
./my_main_process &

# Start the helper process
./my_helper_process

# the my_helper_process might need to know how to wait on the
# primary process to start before it does its work and returns
# now we bring the primary process back into the foreground
# and leave it there
 fg %1
```

Use a process manager

This is a moderately heavy-weight approach that requires you to package supervisord and its configuration in your image (or base your image on one that includes supervisord), along with the different applications it manages

```
FROM ubuntu:latest

RUN apt-get update && apt-get install -y supervisor

RUN mkdir -p /var/log/supervisor

COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf

COPY my_first_process my_first_process

COPY my_second_process my_second_process

CMD ["/usr/bin/supervisord"]
```

Runtime metrics

You can use the **docker stats** command to live stream a container's runtime metrics. The command supports CPU, memory usage, memory limit, and network IO metrics.

The following is a sample output from the **docker stats** command

```
$docker stats redis1 redis2
```

```
CONTAINER           CPU %               MEM USAGE / LIMIT     MEM %               NET I/O             BLOCK I/O
redis1              0.07%               796 KB / 64 MB        1.21%               788 B / 648 B       3.568 MB / 512 KB
redis2              0.07%               2.746 MB / 64 MB      4.29%               1.266 KB / 648 B    12.4 MB / 0 B
```

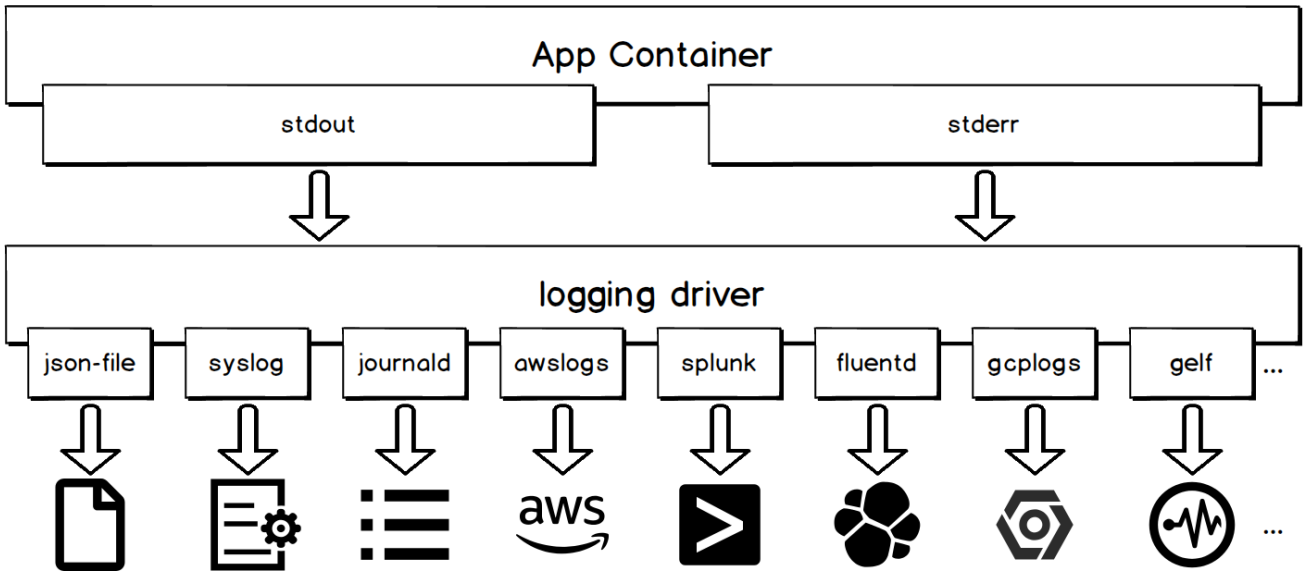
Control groups

Linux Containers rely on **control groups** which not only track groups of processes, but also expose metrics about CPU, memory, and block I/O usage. You can access those metrics and obtain network usage metrics as wellControl groups are exposed through a pseudo-filesystem.

The metrics can be exposed from cgroups are memory, CPU, block I/O

View logs for a container

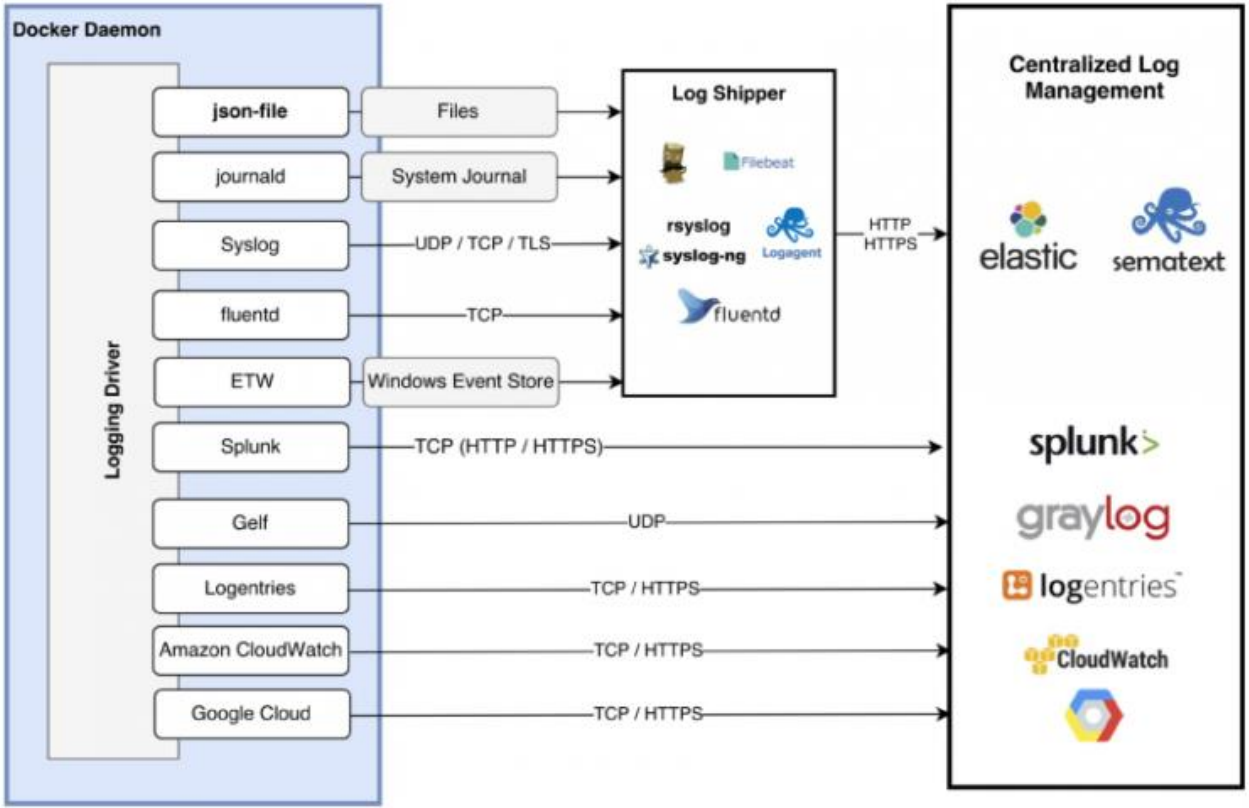
The **docker logs** command shows information logged by a running container



Docker logging driver overview

Docker includes multiple logging mechanisms to help you **get information from running containers and services**. These mechanisms are called logging drivers

As a default, Docker uses the **json-file** logging driver, which caches container logs as JSON internally



Configure the default logging drivers

To configure the Docker daemon to default to a specific logging driver, set the value of log-driver to the name of the logging driver in the daemon.json configuration file

The default logging driver is json-file. The following example sets the default logging driver to the local log driver:

```
{
    "log-driver": "local"
}
```

We can config logging option with the key log-opts

```
{
    "log-driver": "json-file",
    "log-opts": {
        "max-size": "10m",
        "max-file": "3",
        "labels": "production_status",
        "env": "os,customer"
    }
}
```

Configure the logging driver for a container

When you start a container, you can configure it to use a different logging driver than the Docker daemon's default, using the **--log-driver** flag. If the logging driver has configurable options, you can set them using one or more instances of the **--log-opt <NAME>=<VALUE>** flag

The following example starts an Alpine container with the none logging driver.

```
$ docker run -it --log-driver none alpine ash
```

To find the current logging driver for a running container, use docker inspect command

```
$ docker inspect -f '{{.HostConfig.LogConfig.Type}}' <CONTAINER>

  json-file
```

Install the logging driver plugin

To install a logging driver plugin, use **docker plugin install <org/image>**, using the information provided by the plugin developer.

You can list all installed plugins using docker plugin ls, and you can inspect a specific plugin using **docker inspect**.

Customize log driver output

The tag log option specifies how to format a tag that identifies the container's log messages. By default, the system uses the first 12 characters of the container ID. To override this behavior, specify a tag option:

```
$ docker run --log-driver=fluentd --log-opt fluentd-address=myhost.local:24224 --log-opt tag="mailer"
```

For example, specifying a --log-opt tag="{{.ImageName}}/{{.Name}}/{{.ID}}" value yields syslog log lines like:

```
Aug  7 18:33:19 HOSTNAME hello-world/foobar/5790672ab6a0[9103]: Hello from Docker.
```

Overview of Docker Compose

Compose is a tool for defining and running multi-container Docker applications.

It has commands for managing the whole lifecycle of your application:

- Start, stop, and rebuild services
- View the status of running services
- Stream the log output of running services
- Run a one-off command on a service

Using Docker Compose

Using Compose is basically a three-step process:

- Define your app's environment with a Dockerfile
- Define the services that make up your app in docker-compose.yml
- Run **docker compose up** and the Docker compose command starts and runs your entire app.

A docker-compose.yml looks like this:

```yaml
version: "3.9"  # optional since v1.27.0
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

Features of Docker Compose

The features of **Compose** that make it effective are:

- Multiple isolated environments on a single host
- Preserve volume data when containers are created
- Only recreate containers that have changed
- Variables and moving a composition between environments

Install Docker Compose

You can run **Compose** on macOS, Windows, and 64-bit Linux.

Docker Compose relies on Docker Engine for any meaningful work, so make sure you have **Docker Engine** installed either locally or remote, depending on your setup.

- On desktop systems like Docker Desktop for Mac and Windows, Docker Compose is included as part of those desktop installs.
- On Linux systems, first install the Docker Engine for your OS as described on the Get Docker page, then installing Compose on Linux systems.
- You can run Compose as a non-root user

Environment variables in Compose

It's possible to use environment variables in your shell to populate values inside a Compose file:

```
web:
  image: "webapp:${TAG}"
```

You can set default values for any environment variables referenced in the Compose file, or used to configure Compose, in an environment file named .env

```
$ cat .env
TAG=v1.5

$ cat docker-compose.yml
version: '3'
services:
  web:
    image: "webapp:${TAG}"
```

Using the “--env-file” option

By passing the file as an argument, you can store it anywhere and name it appropriately, for example, .env.ci, .env.dev, .env.prod. Passing the file path is done using the --env-file option:

```
$ docker-compose --env-file ./config/.env.dev up
```

The .env file is loaded by default. Passing the --env-file argument overrides the default file path

Set environment variables in containers

You can set environment variables in a service's containers with the 'environment' key, just like with docker run -e VARIABLE=VALUE ...:

```
web:
  environment:
    - DEBUG=1
```

Pass environment variables to containers

You can pass environment variables from your shell straight through to a service's containers with the 'environment' key by not giving them a value, just like with docker run -e VARIABLE ...

```
web:
  environment:
    - DEBUG
```

The “env_file” configuration option

You can pass multiple environment variables from an external file through to a service’s containers with the ‘env_file’ option, just like with docker run --env-file=FILE …:

```
web:
  env_file:
    - web-variables.env
```

Set environment variables with 'docker-compose run'

Similar to docker run -e, you can set environment variables on a one-off container with docker-compose run -e:

```
$ docker-compose run -e DEBUG=1 web python console.py
```

You can also pass a variable from the shell by not giving it a value:

```
$ docker-compose run -e DEBUG web python console.py
```

Networking in Compose

By default **Compose** sets up a single network for your app. Each container for a service joins the default network and is both reachable by other containers on that network, and discoverable by them at a hostname identical to the container name.

Your app's network is given a name based on the "project name", which is based on the name of the directory it lives in. You can override the project name with either the --project-name flag or the COMPOSE_PROJECT_NAME environment variable.

Networking in Compose

When you run docker-compose up, the following happens:

- A network called myapp_default is created.
- A container is created using web's configuration. It joins the network myapp_default under the name web.
- A container is created using db's configuration. It joins the network myapp_default under the name db.

```
version: "3.9"
services:
  web:
    build: .
    ports:
      - "8000:8000"
  db:
    image: postgres
    ports:
      - "8001:5432"
```

Links

Links allow you to define extra aliases by which a service is reachable from another service. They are not required to enable services to communicate - by default, any service can reach any other service at that service's name

```
version: "3.9"
services:

  web:
    build: .
    links:
      - "db:database"
  db:
    image: postgres
```

docker-compose CLI

You can also see this information by running **docker-compose --help** from the command line.

Use the **-f** flag to specify the location of a Compose configuration file.

You can supply multiple **-f** configuration files. When you supply multiple files, Compose combines them into a single configuration. Compose builds the configuration in the order you supply the files. Subsequent files override and add to their predecessors.

DO A LAB: COMPOSE AND WORDPRESS

DO A PROJECT: COMPOSE WITH NODEJS AND MYSQL

Q&A