Docker Compose

jim stafford

Fall 2022 v2021-08-27: Built: 2022-12-07 06:18 EST

Table of Contents

Chapter 1. Introduction

In a few previous lectures we have used the raw Docker API command line calls to perform the desired goals. At some early point there will become unwieldy and we will be searching for a way to wrap these commands. Years ago, I resorted to Ant and the exec command to wrap and chain my high level goals. In this lecture we will learn about something far more native and capable to managing Docker containers — docker-compose.

1.1. Goals

You will learn:

- how to implement a network of services for development and testing using Docker Compose
- how to operate a Docker Compose network lifecycle and how to interact with the running instances

1.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. identify the purpose of Docker Compose for implementing a network of virtualized services
2. create a Docker Compose file that defines a network of services and their dependencies
3. custom configure a Docker Compose network for different uses
4. perform Docker Compose lifecycle commands to build, start, and stop a network of services
5. execute ad-hoc commands inside running images
6. instantiate back-end services for use with the follow-on database lectures

Chapter 2. Development and Integration Testing with Real Resources

To date, we have primarily worked with a single Web application. In the follow-on lectures we will soon need to add back-end database resources.

We can test with mocks and in-memory versions of some resources. However, there will come a day when we are going to need a running copy of the real thing or possibly a specific version.

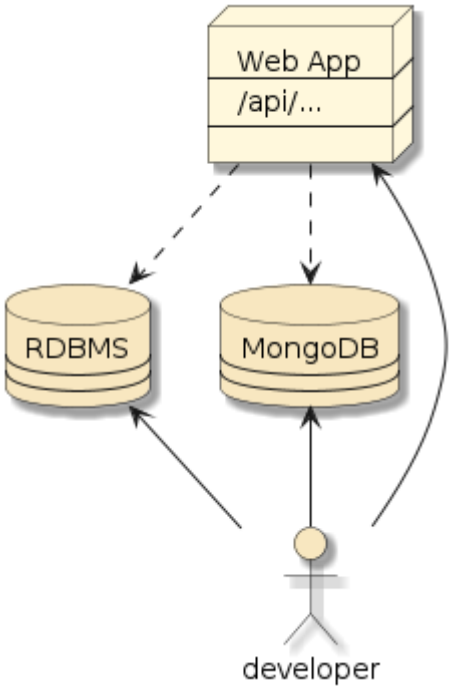

Figure 1. Need to Integrate with Specific Real Services

We have already gone through the work to package our API service in a Docker image and the Docker community has built a plethora of offerings for ready and easy download. Among them are Docker images for the resources we plan to eventually use:

- MongoDB
- Postgres

It would seem that we have a path forward.

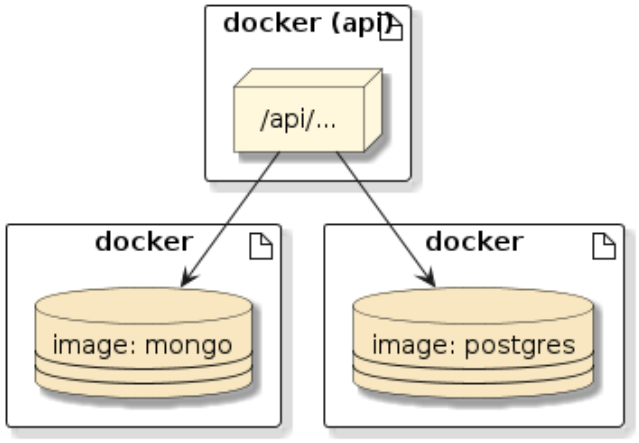

Figure 2. Virtualize Services with Docker

2.1. Managing Images

You know from our initial Docker lectures that we can easily download the images and run them individually (given some instructions) with the `docker run` command. Knowing that — we could try doing the following and almost get it to work.

Manually Starting Images

```
$ docker run --rm -p 27017:27017 \
-e MONGO_INITDB_ROOT_USERNAME=admin \
-e MONGO_INITDB_ROOT_PASSWORD=secret mongo:4.4.0-bionic ①

$ docker run --rm -p 5432:5432 \
-e POSTGRES_PASSWORD=secret postgres:12.3-alpine ②

$ docker run --rm -p 9080:8080 \
-e MONGODB_URI=... \ ③
-e DATABASE_URL=... \
docker-hello-example:6.0.1-SNAPSHOT
```

① using the mongo container from Dockerhub

② using the postgres container from Dockerhub

③ using our example Spring Boot Web application; it does not yet use the databases

However, this begins to get complicated when:

- we start integrating the API image with the individual resources through networking
- we want to make the test easily repeatable
- we want multiple instances of the test running concurrently on the same machine without interference with one another

Lets not mess with manual Docker commands for too long! There are better ways to do this with Docker Compose.

Chapter 3. Docker Compose

DockerCompose is a tool for defining and running multi-container Docker applications. With Docker Compose, we can:

- define our network of applications in a single YAML file
- start/stop applications according to defined dependencies
- run commands inside of running images
- treat the running applications as normal, running Docker images

3.1. Docker Compose is Local to One Machine

Docker Compose runs everything local. It is a modest but necessary step above Docker but far simpler than any of the distributed environments that logically come after it (e.g., Docker Swam, Kubernetes). If you are familiar with Kubernetes and MiniKube, then you can think of Docker Compose is a very simple/poor man's Helm Chart. "Poor" in that it only runs on a single machine. "Simple" because you only need to define details of each service and not have to worry about distributed aspects or load balancing that might come in a more distributed solution.

With Docker Compose, there

- are one or more configuration files
- is the opportunity to apply environment variables and extensions
- are commands to build and control lifecycle actions of the network

Let's start with the Docker Compose configuration file.

Chapter 4. Docker Compose Configuration File

The Docker Compose (configuration) file is based on YAML — which uses a concise way to express information based on indentation and firm symbol rules. Assuming we have a simple network of three (3) services, we can limit our definition to a file `version` and individual `services`.

docker-compose.yml Shell

```
version: '3.8'
services:
  mongo:
    ...
  postgres:
    ...
  api:
    ...
```

- **version** - informs the docker-compose binary what features could be present within the file. I have shown a recent version of `3.8` but our use of the file will be very basic and could likely be set to `3` or as low as `2`.
- **services** - lists the individual nodes and their details. Each node is represented by a Docker image and we will look at a few examples next.

Refer to the Compose File Reference for more details.

4.1. mongo Service Definition

The `mongo` service defines our instance of MongoDB.

mongo Service Definition

```
  mongo:
    image: mongo:4.4.0-bionic
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: secret
#    ports: ①
#      - "27017" ②
#      - "27017:27017" ③
#      - "37001:27017" ④
#      - "127.0.0.1:37001:27017" ⑤
```

① not assigning port# here

② `27017` internal, random external

③ `27017` both internal and external

④ `37001` external and 27017` internal

⑤ `37001` exposed only on `127.0.0.1` external and 27017` internal

- **image** - identifies the name and tag of the Docker image. This will be automatically downloaded if not already available locally
- **environment** - defines specific environment variables to be made available when running the image.
  - `VAR: X` passes in variable `VAR` with value `X`.
  - `VAR` by itself passes in variable `VAR` with whatever the value of `VAR` has been assigned to be in the environment (i.e., environment variable or from environment file).
- **ports** - maps a container port to a host port with the syntax `"host interface:host port#:container port#"`
  - `host port#:container port#` by itself will map to add host interfaces
  - `"container port#"` by itself will be mapped to a random host port#
  - no ports defined means the container port# that do exist are only accessible within the network of services defined within the file

4.2. postgres Service Definition

The `postgres` service defines our instance of Postgres.

postgres Service Definition

```
  postgres:
    image: postgres:12.3-alpine
#    ports: ①
#      - "5432:5432"
    environment:
      POSTGRES_PASSWORD: secret
```

- the default username and database name is `postgres`
- assigning a custom password of `secret`

Mapping Port to Specific Host Port Restricts Concurrency to one Instance

Mapping a container port# to a fixed host port# makes the service easily accessible from the host via a well-known port# but restricts the number of instances that can be run concurrently to one. This is typically what you might do with development resources. We will cover how to do both easily — shortly.

4.3. api Service Definition

The `api` service defines our API server with the Votes and Elections Services. This service will

become a client of the other three services.

api Service Definition

```
  api:
    build:
      context: .
      dockerfile: Dockerfile.layered
    image: docker-hello-example:layered
    ports:
      - "${API_PORT:-8080}:8080"
    depends_on:
      - mongo
      - postgres
    environment:
      - spring.profiles.active=integration
      - MONGODB_URI=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin
      - DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres
```

- **build** - identifies a source Dockerfile that can build the image for this service
  - **context** - defines the path to the Dockerfile
  - **dockerfile** - defines the specific name of the Dockerfile (optional in this case)
- **image** - identifies the name and tag used for the built image
- **ports** - using a ${variable:-default} reference so that we have option to expose the container port# 8080 to a dynamically assigned host port# during testing. If `API_PORT` is not resolved to a value, the default `8080` value will be used.
- **depends_on** - establishes a dependency between the images. This triggers a start of dependencies when starting this service. It also adds a hostname to this image's environment. Therefore, the `api` server can reach the other services using hostnames `mongo` and `postgres`. You will see an example of that when you look closely at the URLs in the later examples.
- **environment** - environment variables passed to Docker image.
  - using `spring.profiles.active` to instruct API to use `integration` profile
  - API is not yet using the databases, but these URLs are consistent with what will be encountered when deployed to Heroku.
  - if only the environment variable name is supplied, it's value will not be defined here and the value from external sources will be passed at runtime

4.4. Build/Download Images

We can trigger the build or download of necessary images using the `docker-compose build` command or simply by starting `api` service the first time.

Building API Service

```
$ docker-compose build
```

```
postgres uses an image, skipping
mongo uses an image, skipping
Building api
[+] Building 0.2s (13/13) FINISHED
 => => naming to docker.io/library/docker-hello-example:layered
..
```

After the first start, a re-build is only performed using the `build` command or when the `--build` option.

4.5. Default Port Assignments

If we start the services ...

```
$ export API_PORT=1234 && docker-compose up -d  ①
Creating network "docker-hello-example_default" with the default driver
Creating docker-hello-example_mongo_1    ... done
Creating docker-hello-example_postgres_1 ... done
Creating docker-hello-example_api_1      ... done
```

① `up` starts service and `-d` runs the container in the background as a daemon

You will notice that no ports were assigned to the unassigned `mongo` and `postgres` services. However, the given shown port# in the output is available to the other hosts within that Docker network. If we don't need `mongo` or `postgres` accessible to the host's network — we are good. The `api` service was assigned a variable (value `1234`) port# — which is accessible to the host's network.

```
$ docker-compose ps
             Name                State     Ports
-------------------------------------------------------------------------------
docker-hello-example_api_1      Up        0.0.0.0:1234->8080/tcp,:::1234->8080/tcp
docker-hello-example_mongo_1    Up        27017/tcp
docker-hello-example_postgres_1 Up        5432/tcp
```

Using Variable-Assigned API Port#

```
$ curl  http://localhost:1234/api/hello?name=jim
hello, jim
```

4.6. Compose Override Files

Docker Compose files can be layered from base (shown above) to specialized. The following example shows the previous definitions being extended to include mapped host port# mappings. We might add this override in the development environment to make it easy to access the service ports on the host's local network using well-known port numbers.

Example Compose Override File

```
version: '3.8'
services:
  mongo:
    ports:
      - "27017:27017"
  postgres:
    ports:
      - "5432:5432"
```

Notice how the container port# is now mapped according to how the override file has specified.

Shutdown and Start New Container Instances

```
$ unset API_PORT
$ docker-compose down
$ docker-compose up -d
```

Port Mappings with Compose Override File Used

```
$ docker-compose ps
             Name               State                       Ports
-------------------------------------------------------------------------------------
docker-hello-example_api_1       Up     0.0.0.0:8080->8080/tcp,:::8080->8080/tcp
docker-hello-example_mongo_1     Up     0.0.0.0:27017->27017/tcp,:::27017->27017/tcp
docker-hello-example_postgres_1  Up     0.0.0.0:5432->5432/tcp,:::5432->5432/tcp
```

Override Limitations May Cause Compose File Refactoring

There is a limit to what you can override versus augment. Single values can replace single values. However, lists of values can only contribute to a larger list. That means we cannot create a base file with ports mapped and then a build system override with the port mappings taken away.

4.7. Compose Override File Naming

Docker Compose looks for a specially named file of `docker-compose.override.yml` in the local directory next to the local `docker-compose.yml` file.

Example File Override Syntax

```
$ ls docker-compose.*
docker-compose.override.yml docker-compose.yml

$ docker-compose up ①
```

① Docker Compose automatically applies overrides from `docker-compose.override.yml` in this case

4.8. Multiple Compose Files

Docker Compose will accept a series of explicit `-f file` specifications that are processed from left to right. This allows you to name your own override files.

Example File Override Syntax

```
$ docker-compose -f docker-compose.yml -f development.yml up ①
$ docker-compose -f docker-compose.yml -f integration.yml up
$ docker-compose -f docker-compose.yml -f production.yml up
```

① starting network in foreground with two configuration files, with the left-most file being specialized by the right-most file

4.9. Environment Files

Docker Compose will look for variables to be defined in the following locations in the following order:

1. as an environment variable
2. in an environment file
3. when the variable is named and set to a value in the Compose file

Docker Compose will use `.env` as its default environment file. A file like this would normally not be checked into CM since it might have real credentials, etc.

.env Files Normally are not Part of SCM Check-in

```
$ cat .gitignore
...
.env
```

Example .env File

```
API_PORT=9090
```

You can also explicitly name an environment file to use. The following is explicitly applying the `alt-env` environment file — thus bypassing the `.env` file.

Example Explicit Environment File

```
$ cat alt-env
API_PORT=9999

$ docker-compose --env-file alt-env up -d ①
$ docker ps
IMAGE                          PORTS
NAMES
```

```
dockercompose-votes-api:latest 0.0.0.0:9999->8080/tcp
...
```

① starting network in background with an alternate environment file mapping API port to 9999

Chapter 5. Docker Compose Commands

5.1. Build Source Images

With the `docker-compose.yml` file defined — we can use that to control the build of our source images. Notice in the example below that it is building the same image we built in the previous lecture.

Example Docker Compose build Output

```
$ docker-compose build
postgres uses an image, skipping
mongo uses an image, skipping
Building api
[+] Building 0.2s (13/13) FINISHED
 => => naming to docker.io/library/docker-hello-example:layered
```

5.2. Start Services in Foreground

We can start all the the services in the foreground using the `up` command. The command will block and continually tail the output of each container.

Example docker-compose up Command

```
$ docker-compose up
docker-hello-example_mongo_1 is up-to-date
docker-hello-example_postgres_1 is up-to-date
Recreating docker-hello-example_api_1 ... done
Attaching to docker-hello-example_mongo_1, docker-hello-example_postgres_1, docker-
hello-example_api_1
```

We can trigger a new build with the `--build` option. If there is no image present, a build will be triggered automatically but will not be automatically reissued on subsequent commands without supplying the `--build` option.

5.3. Project Name

Docker Compose names all of our running services using a project name prefix. The default project name is the parent directory name. Notice below how the parent directory name `docker-hello-example` was used in each of the running service names.

Project Name Defaults to Parent Directory Name

```
pwd
.../svc-container/docker-hello-example

$ docker-compose up
```

```
docker-hello-example_mongo_1 is up-to-date
docker-hello-example_postgres_1 is up-to-date
Recreating docker-hello-example_api_1 ... done
```

We can explicitly set the project name using the `-p` option. This can be helpful if the parent directory happens to be something generic — like `target` or `src/test/resources`.

```
$ docker-compose -p foo up ①
Creating network "foo_default" with the default driver
Creating foo_postgres_1 ... done ②
Creating foo_mongo_1    ... done
Creating foo_api_1      ... done
Attaching to foo_postgres_1, foo_mongo_1, foo_api_1
```

① manually setting project name to `foo`

② network and services all have prefix of `foo`

5.4. Start Services in Background

We can start the processes in the background by adding the `-d` option.

```
$ docker-compose up -d
Creating network "docker-hello-example_default" with the default driver
Creating docker-hello-example_postgres_1 ... done
Creating docker-hello-example_mongo_1    ... done
Creating docker-hello-example_api_1      ... done
$ ①
```

① `-d` option starts all services in the background and returns us to our shell prompt

5.5. Access Service Logs

With the services running in the background, we can access the logs using the `docker-compose logs` command.

```
$ docker-compose logs api ①
$ docker-compose logs -f api mongo ②
$ docker-compose logs --tail 10 ③
```

① returns all logs for the `api` service

② tails the current logs for the `api` and `mongo` services.

③ returns the latest 10 messages in each log

5.6. Stop Running Services

If the services were started in the foreground, we can simply stop them with the <ctl>+C command. If they were started in the background or in a separate shell, we can stop them by executing the `down` command in the `docker-compose.yml` directory.

```
$ docker-compose down
Stopping docker-hello-example_api_1      ... done
Stopping docker-hello-example_mongo_1    ... done
Stopping docker-hello-example_postgres_1 ... done
Removing docker-hello-example_api_1      ... done
Removing docker-hello-example_mongo_1    ... done
Removing docker-hello-example_postgres_1 ... done
Removing network docker-hello-example_default
```

Chapter 6. Docker Cleanup

Docker Compose will mostly cleanup after itself. The only exceptions are the older versions of the API image and the builder image that went into creating the final API images. Using my example settings, these are all end up being named and tagged as none in the images repository.

Example Docker Image Repository State

```
$  docker images
REPOSITORY                          TAG        IMAGE ID       CREATED          SIZE
docker-hello-example                layered          9c45ff5ac1cf   17 hours ago
316MB
registry.heroku.com/ejava-docker/web latest          9c45ff5ac1cf   17 hours ago
316MB
docker-hello-example                execjar          669de355e620   46 hours ago
315MB
dockercompose-votes-api             latest           da94f637c3f4   5 days ago
340MB
<none>                              <none>           d64b4b57e27d   5 days ago
397MB
<none>                              <none>           c5aa926e7423   7 days ago
340MB
<none>                              <none>           87e7aabb6049   7 days ago
397MB
<none>                              <none>           478ea5b821b5   10 days ago
340MB
<none>                              <none>           e1a5add0b963   10 days ago
397MB
<none>                              <none>           4e68464bb63b   11 days ago
340MB
<none>                              <none>           b09b4a95a686   11 days ago
397MB
...
<none>                              <none>           ee27d8f79886   4 months ago
396MB
adoptopenjdk                        14-jre-hotspot   157bb71cd724   5 months ago
283MB
mongo                               4.4.0-bionic     409c3f937574   12 months ago
493MB
postgres                            12.3-alpine      17150f4321a3   14 months ago
157MB
<none>                              <none>           b08caee4cd1b   41 years ago
279MB
docker-hello-example                6.0.1-SNAPSHOT   a855dabfe552   41 years ago
279MB
```

Docker Images are Actually Smaller than Provided SIZE

Even though Docker displays each of these images as >300MB, they may share some base layers and — by themselves — much smaller. The value presented is the

> space taken up if all other images are removed or if this image was exported to its own TAR file.

6.1. Docker Image Prune

The following command will clear out any docker images that are not named/tagged and not part of another image.

Example Docker Image Prune Output

```
$ docker image prune
WARNING! This will remove all dangling images.
Are you sure you want to continue? [y/N] y
Deleted Images:
deleted: sha256:ebc8dcf8cec15db809f4389efce84afc1f49b33cd77cfe19066a1da35f4e1b34
...
deleted: sha256:e4af263912d468386f3a46538745bfe1d66d698136c33e5d5f773e35d7f05d48

Total reclaimed space: 664.8MB
```

6.2. Docker System Prune

The following command performs the same type of cleanup as the `image` prune command and performs an additional amount on cleanup many other Docker areas deemed to be "trash".

Example Docker System Prune Output

```
$ docker system prune
WARNING! This will remove:
  - all stopped containers
  - all networks not used by at least one container
  - all dangling images
  - all dangling build cache

Are you sure you want to continue? [y/N] y
Deleted Networks:
testcontainers-votes-spock-it_default

Deleted Images:
deleted: sha256:e035b45628fe431901b2b84e2b80ae06f5603d5f531a03ae6abd044768eec6cf
...
deleted: sha256:c7560d6b795df126ac2ea532a0cc2bad92045e73d1a151c2369345f9cd0a285f

Total reclaimed space: 443.3MB
```

6.3. Image Repository State After Pruning

After pruning the images — we have just the named/tagged image(s).

Docker Image Repository State After Pruning

```
$ docker images
REPOSITORY                              TAG             IMAGE ID       CREATED
SIZE
docker-hello-example                    layered         9c45ff5ac1cf   17 hours ago
316MB
registry.heroku.com/ejava-docker/web    latest          9c45ff5ac1cf   17 hours ago
316MB
docker-hello-example                    execjar         669de355e620   46 hours ago
315MB
mongo                                   4.4.0-bionic    409c3f937574   12 months ago
493MB
postgres                                12.3-alpine     17150f4321a3   14 months ago
157MB
docker-hello-example                    6.0.1-SNAPSHOT  a855dabfe552   41 years ago
279MB
```

Chapter 7. Summary

In this module we learned:

- the purpose of Docker Compose and how it is used to define a network of services operating within a virtualized Docker environment
- to create a Docker Compose file that defines a network of services and their dependencies
- to custom configure a Docker Compose network for different uses
- perform Docker Compose lifecycle commands
- execute ad-hoc commands inside running images

Why We Covered Docker and Docker Compose

The Docker and Docker Compose lectures have been included in this course because of the high probability of your future deployment environments for your Web applications and to provide a more capable and easy to use environment to learn, develop, and debug.

Where are You?

This lecture leaves you at a point where your Web application and database instances are alive but not yet communicating. The URLs/URIs shown in this example are consistent with what you will encounter in Heroku when deploying. However, we have much to do before then.

Where are You Going?

In the following series of lectures we will dive into the persistence tier, do some local development with the resources we have just setup, and then return to this topic once we are ready to re-deploy with a database-ready Web application.