DevOps Training

Week 4:
Docker Compose

DICE
ANALYTICS

Muhammad Sohaib Ali

in /msohaibali

Week 3 Recap

- Introduction to Docker file
- Docker Images
- Containers

Before Moving Forward

- Any **Questions/Queries** from Previous Week?

Week 4 Coverage

- Docker Network
- Docker Compose
- Docker Compose (LABS including different Projects)

Docker-Compose

Prerequisites

- Complete understand of docker files, images, containers
- This is an **advance level topic** in docker
- If no understanding of Docker, Go Back and revise

Docker Compose Overview

- Compose is a tool for defining and running multi-container Docker applications.
- With Compose, you use a YAML file to configure your application’s services.
- Then, with a single command, you create and start all the services from your configuration.
- Compose works in all environments: production, staging, development, testing, as well as CI workflows. It also 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

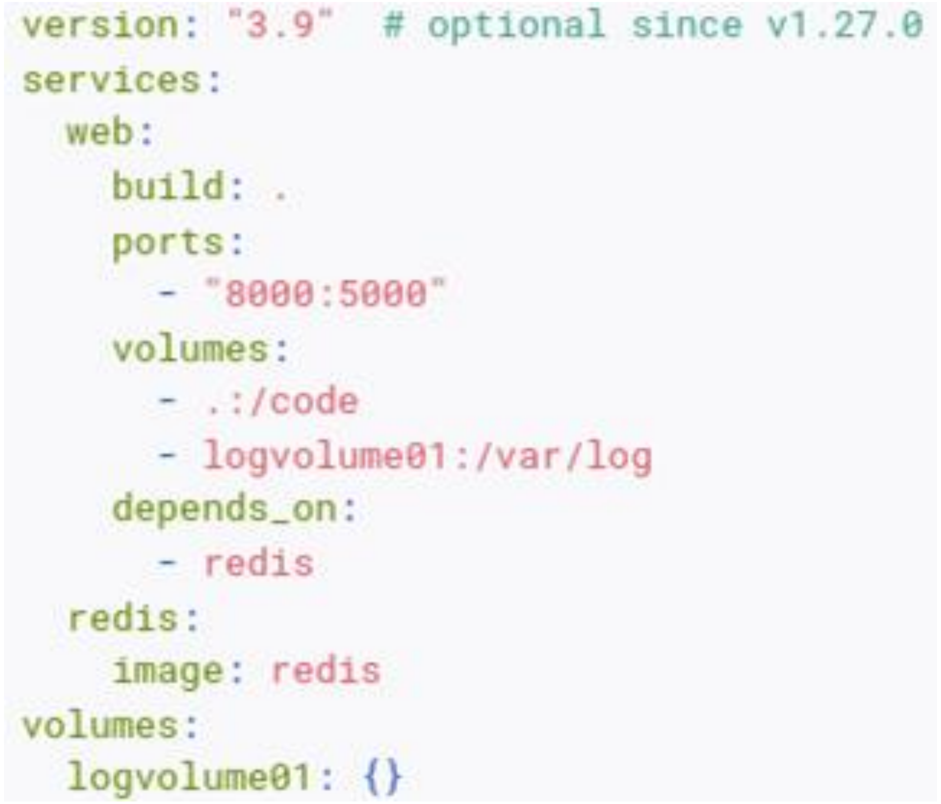
Docker Compose Overview (CONTD)

Using Compose is essentially a three-step process:

1. Define your app's environment with a Docker file so it can be reproduced anywhere.
2. Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
3. Run docker compose up and the Docker compose command starts and runs your entire app. You can alternatively run docker-compose up using Compose standalone(docker-compose binary).

Docker Compose YAML

- A docker-compose YAML looks like:



```yaml
version: "3.9"  # optional since v1.27.0
services:
  web:
    build: .
    ports:
      - "8000:5000"
    volumes:
      - .:/code
      - logvolume01:/var/log
    depends_on:
      - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

Docker-Compose

Features

Key features of Docker Compose

The Key Features of docker compose that makes is effective are

- Have multiple isolated environments on a single host
- Preserves volume data when containers are created
- Only recreate containers that have changed
- Supports variables and moving a composition between environments

Feature 1: Multiple Isolated Environments

Compose uses a project name to isolate environments from each other. You can make use of this project name in several different contexts:

- On a dev host, to create multiple copies of a single environment, such as when you want to run a stable copy for each feature branch of a project.
- On a CI server, to keep builds from interfering with each other, you can set the project name to a unique build number.
- On a shared host or dev host, to prevent different projects, which may use the same service names, from interfering with each other.

The default project name is the basename of the project directory. You can set a custom project name by using -p command line option or COMPOSE_PROJECT_NAME environment variable.

Feature 2: Preserves Volume Data

- Compose preserves all volumes used by your services.
- When docker compose up runs, if it finds any containers from previous runs, it copies the volumes from the old container to the new container.
- This process ensures that any data you’ve created in volumes isn’t lost.

Feature 3: Recreate Containers that Changed

- Compose caches the configuration used to create a container. When you restart a service that has not changed, Compose re-uses the existing containers.
- Re-using containers means that you can make changes to your environment very quickly.

Feature 4: Supports Variables - Compositions

- Compose Supports Variables in the Compose file. You can use these variables to customize your composition for different environments, or different users.

Docker-Compose

Common Use Cases

Common Use-Cases of Docker Compose

- Development Environments
- Automated Testing Environments
- Single Host Deployments

Docker-Compose

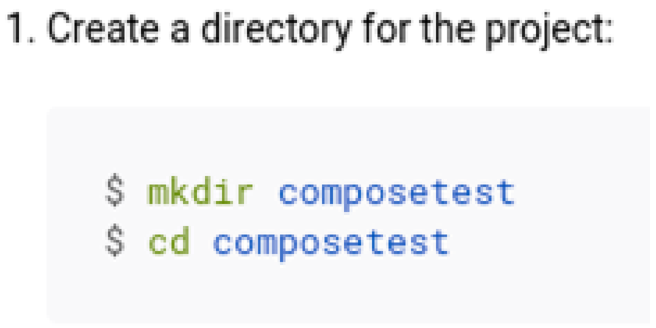
Hands On - LABS

Installing Docker-Compose

- https://docs.docker.com/compose/install/

LAB: Docker Compose

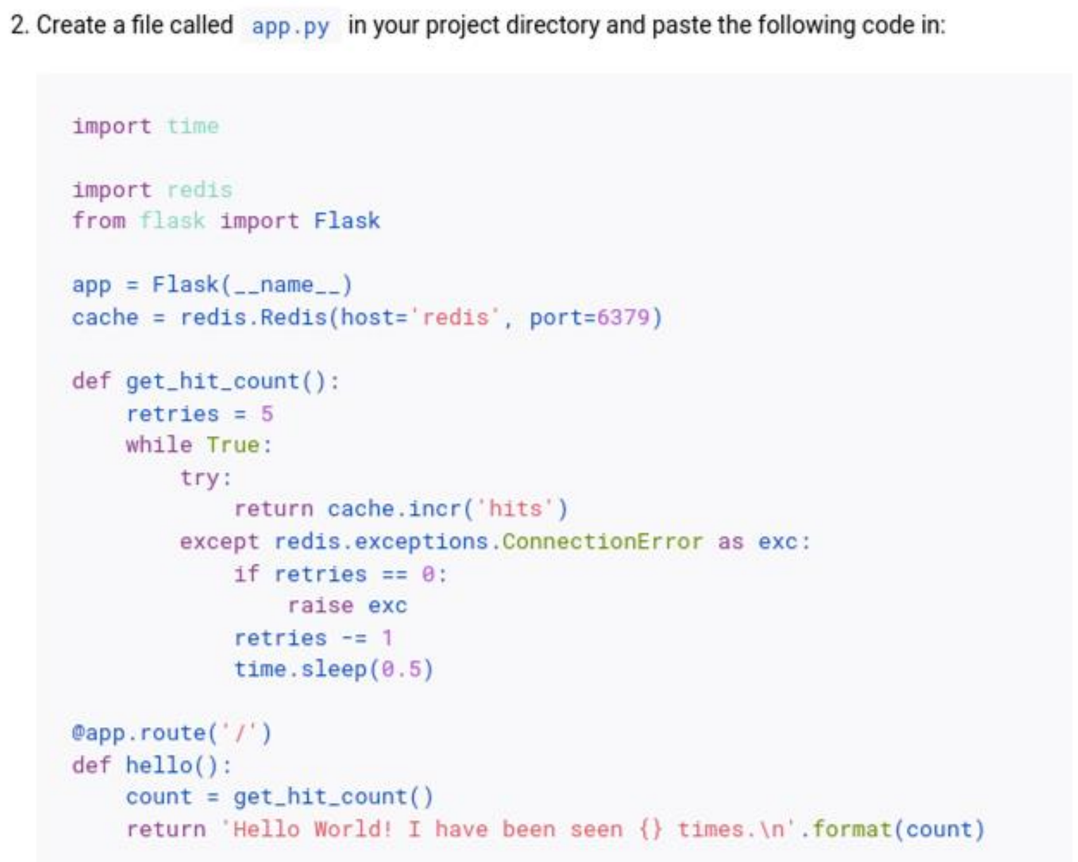
Step 1: Define the application dependencies



1. Create a directory for the project:

```
$ mkdir composetest
$ cd composetest
```

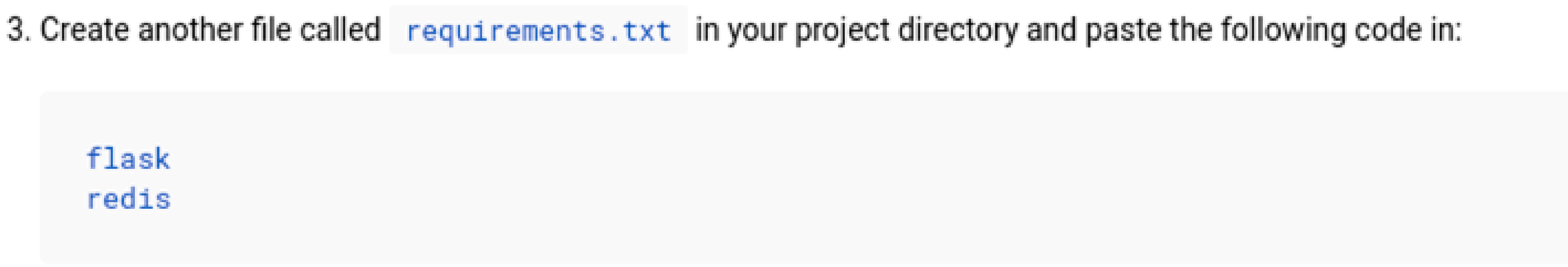
Step 1: Define the application dependencies



2. Create a file called `app.py` in your project directory and paste the following code in:

```python
import time

import redis
from flask import Flask

app = Flask(__name__)
cache = redis.Redis(host='redis', port=6379)

def get_hit_count():
    retries = 5
    while True:
        try:
            return cache.incr('hits')
        except redis.exceptions.ConnectionError as exc:
            if retries == 0:
                raise exc
            retries -= 1
            time.sleep(0.5)

@app.route('/')
def hello():
    count = get_hit_count()
    return 'Hello World! I have been seen {} times.\n'.format(count)
```

Step 1: Define the application dependencies



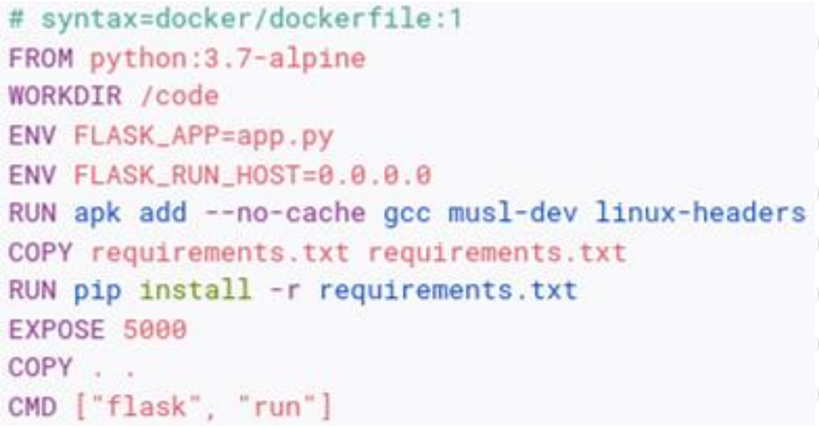
3. Create another file called `requirements.txt` in your project directory and paste the following code in:

```
flask
redis
```

Step 2: Create a Dockerfile

The Dockerfile is used to build a Docker image. The image contains all the dependencies the Python application requires, including Python itself.

In your project directory, create a file named `Dockerfile` and paste the following code in:

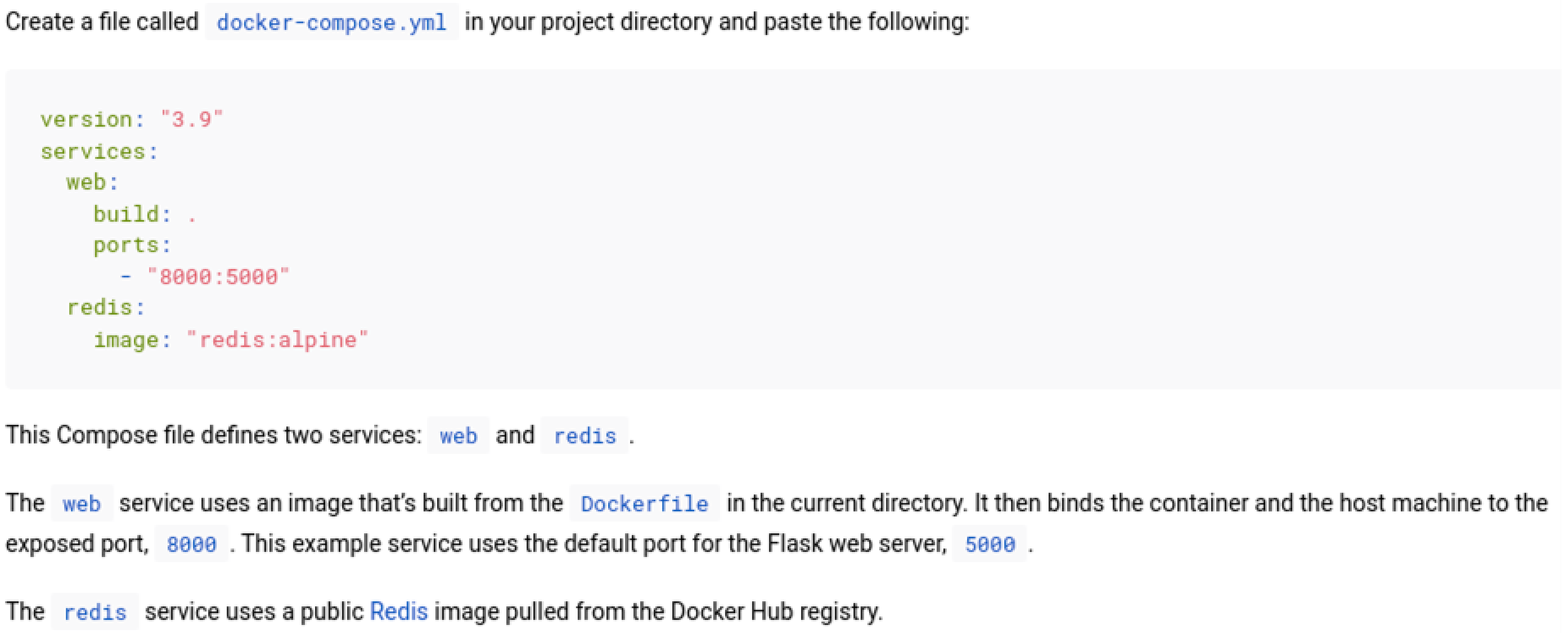


```
# syntax=docker/dockerfile:1
FROM python:3.7-alpine
WORKDIR /code
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
RUN apk add --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
EXPOSE 5000
COPY . .
CMD ["flask", "run"]
```

This tells Docker to:

- Build an image starting with the Python 3.7 image.
- Set the working directory to `/code` .
- Set environment variables used by the `flask` command.
- Install gcc and other dependencies
- Copy `requirements.txt` and install the Python dependencies.
- Add metadata to the image to describe that the container is listening on port 5000
- Copy the current directory `.` in the project to the workdir `.` in the image.
- Set the default command for the container to `flask run` .

Step 3: Define services in a Compose file



Create a file called `docker-compose.yml` in your project directory and paste the following:

```yaml
version: "3.9"
services:
  web:
    build: .
    ports:
      - "8000:5000"
  redis:
    image: "redis:alpine"
```

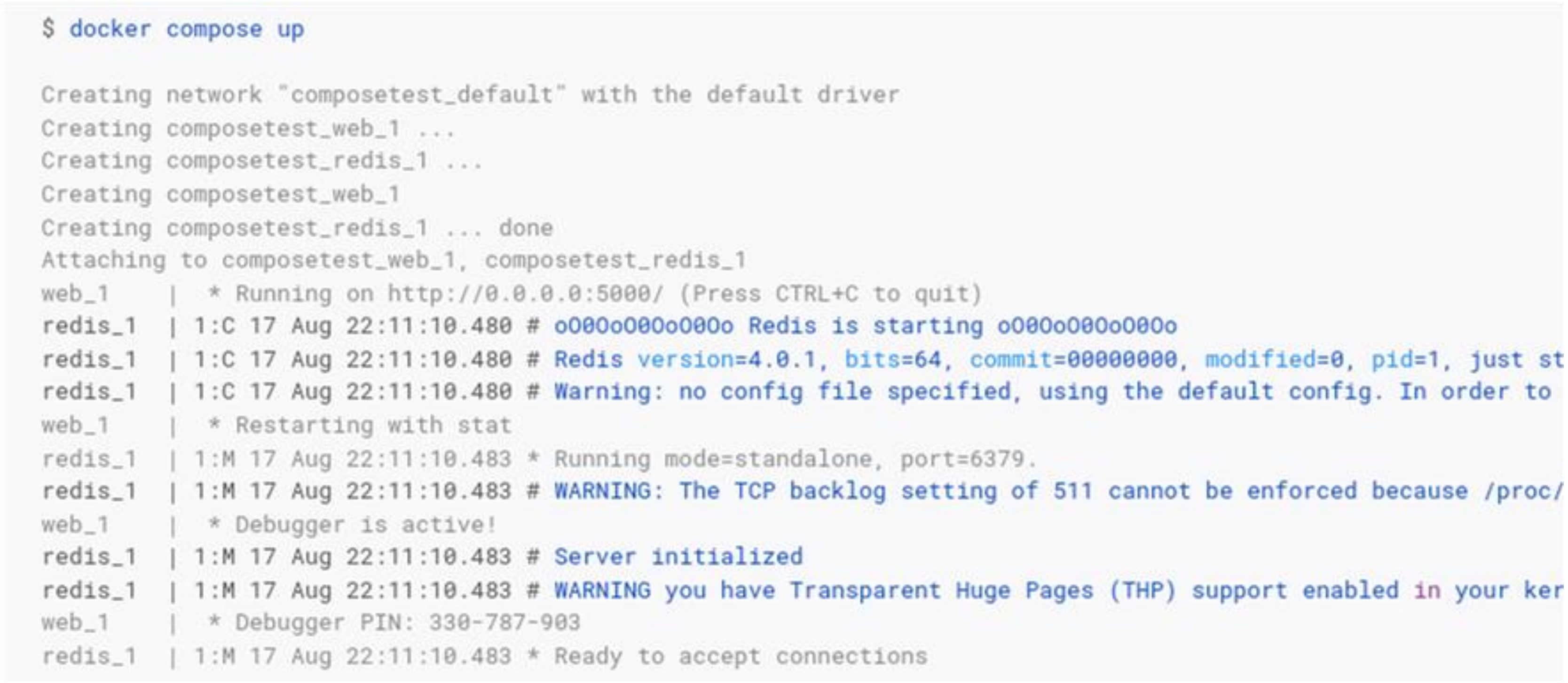
This Compose file defines two services: `web` and `redis` .

The `web` service uses an image that's built from the `Dockerfile` in the current directory. It then binds the container and the host machine to the exposed port, `8000` . This example service uses the default port for the Flask web server, `5000` .

The `redis` service uses a public Redis image pulled from the Docker Hub registry.

Step 4: Build and run your app with Compose

1. From your project directory, start up your application by running `docker compose up`.



```
$ docker compose up

Creating network "composetest_default" with the default driver
Creating composetest_web_1 ...
Creating composetest_redis_1 ...
Creating composetest_web_1
Creating composetest_redis_1 ... done
Attaching to composetest_web_1, composetest_redis_1
web_1    |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
redis_1  | 1:C 17 Aug 22:11:10.480 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
redis_1  | 1:C 17 Aug 22:11:10.480 # Redis version=4.0.1, bits=64, commit=00000000, modified=0, pid=1, just st
redis_1  | 1:C 17 Aug 22:11:10.480 # Warning: no config file specified, using the default config. In order to
web_1    |  * Restarting with stat
redis_1  | 1:M 17 Aug 22:11:10.483 * Running mode=standalone, port=6379.
redis_1  | 1:M 17 Aug 22:11:10.483 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/
web_1    |  * Debugger is active!
redis_1  | 1:M 17 Aug 22:11:10.483 # Server initialized
redis_1  | 1:M 17 Aug 22:11:10.483 # WARNING you have Transparent Huge Pages (THP) support enabled in your ker
web_1    |  * Debugger PIN: 330-787-903
redis_1  | 1:M 17 Aug 22:11:10.483 * Ready to accept connections
```

Compose pulls a Redis image, builds an image for your code, and starts the services you defined. In this case, the code is statically copied into the image at build time.

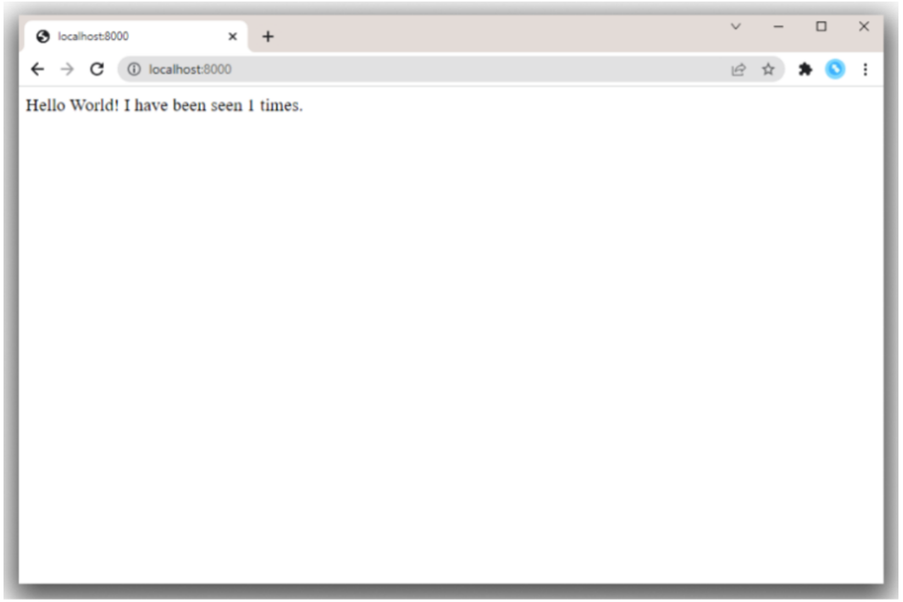
Step 4: Build and run your app with Compose

2. Enter http://localhost:8000/ in a browser to see the application running.

   If this doesn't resolve, you can also try http://127.0.0.1:8000.

   You should see a message in your browser saying:

```
Hello World! I have been seen 1 times.
```



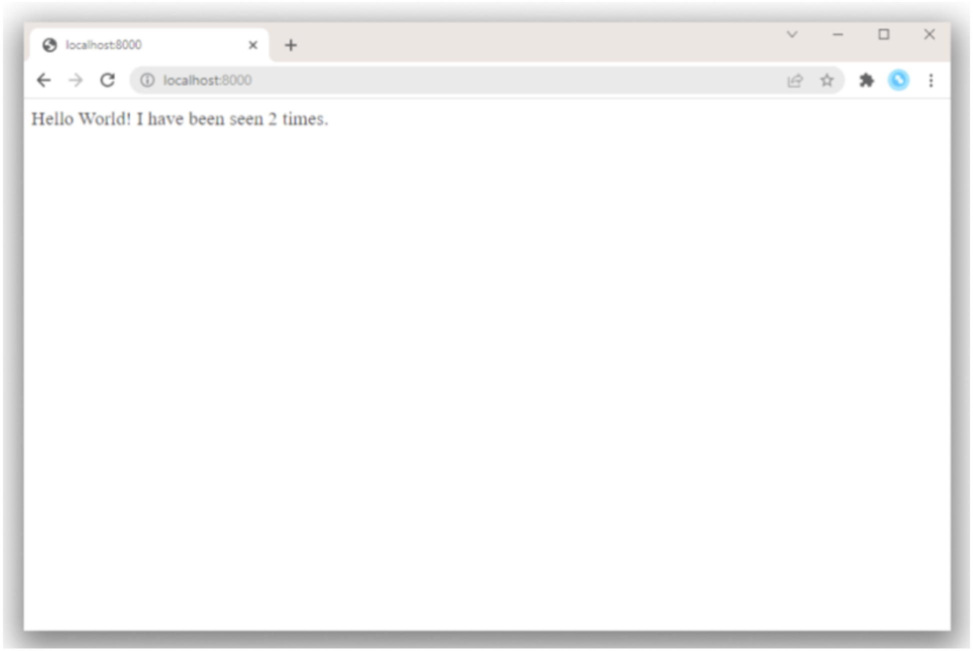
Hello World! I have been seen 1 times.

Step 4: Build and run your app with Compose

3. Refresh the page.

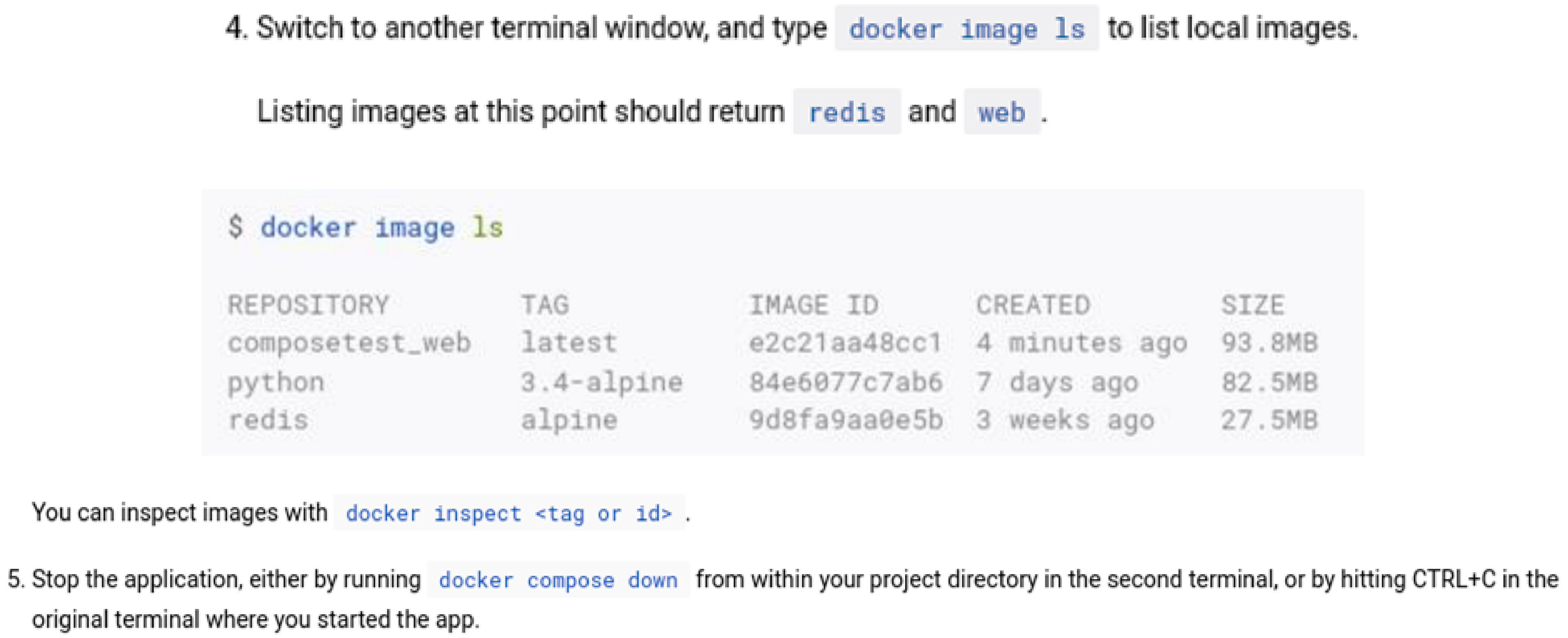
   The number should increment.

```
Hello World! I have been seen 2 times.
```



Hello World! I have been seen 2 times.

Step 4: Build and run your app with Compose



4. Switch to another terminal window, and type `docker image ls` to list local images.

   Listing images at this point should return `redis` and `web`.

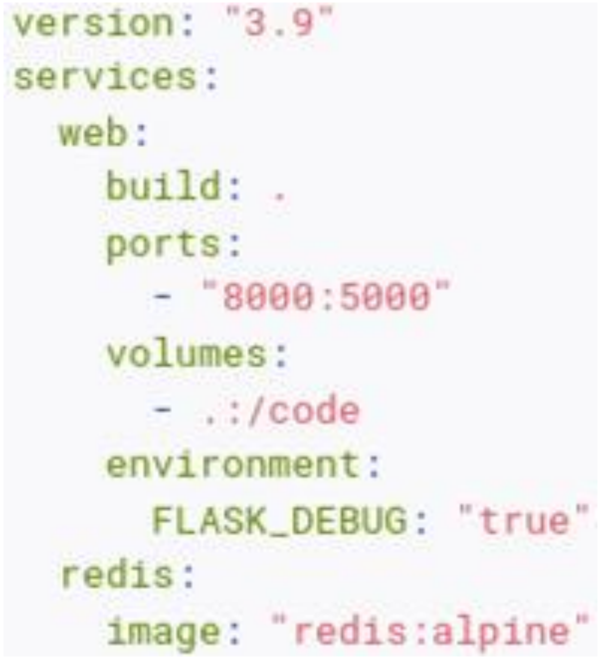
```
$ docker image ls

REPOSITORY        TAG           IMAGE ID      CREATED          SIZE
composetest_web   latest        e2c21aa48cc1  4 minutes ago    93.8MB
python            3.4-alpine    84e6077c7ab6  7 days ago       82.5MB
redis             alpine        9d8fa9aa0e5b  3 weeks ago      27.5MB
```

You can inspect images with `docker inspect <tag or id>`.

5. Stop the application, either by running `docker compose down` from within your project directory in the second terminal, or by hitting CTRL+C in the original terminal where you started the app.

Step 5: Edit the Compose file to add a bind mount

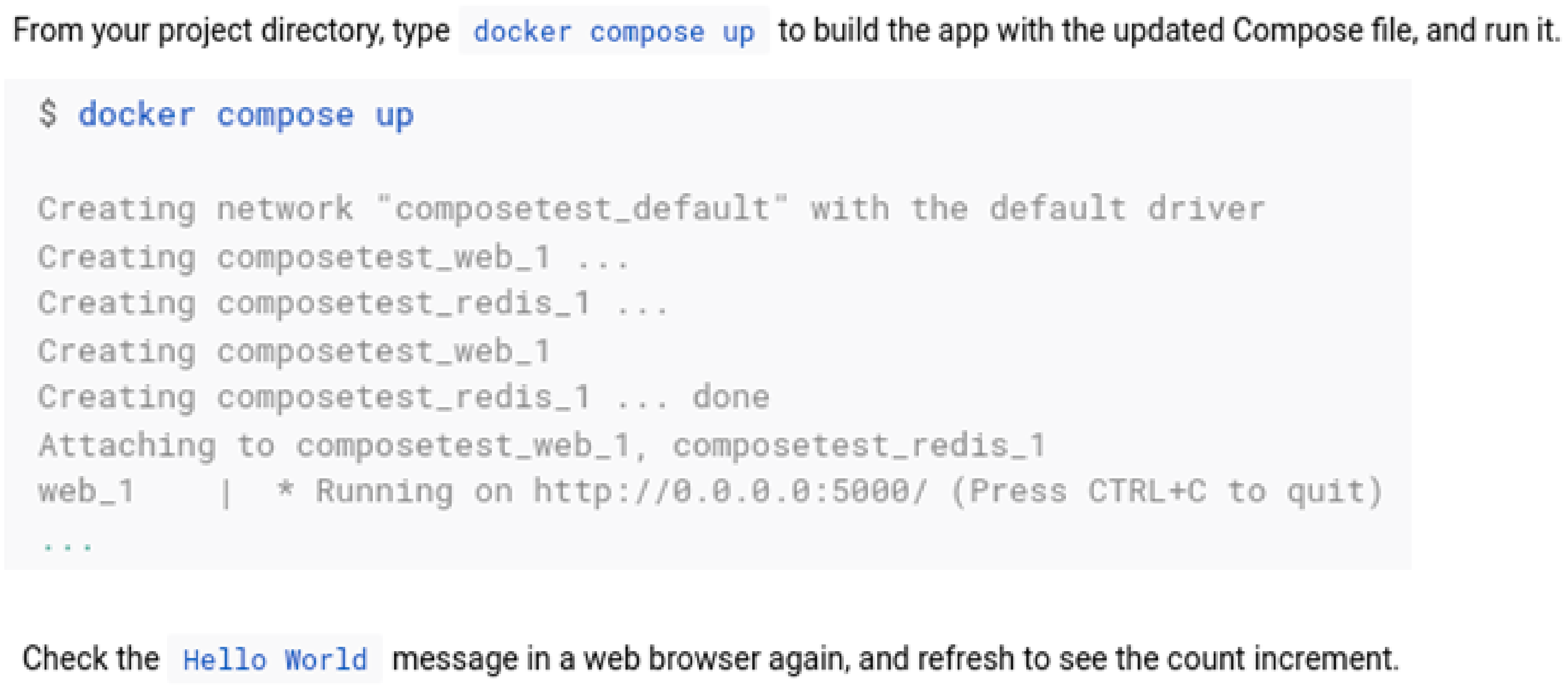
Edit `docker-compose.yml` in your project directory to add a bind mount for the `web` service:



```
version: "3.9"
services:
  web:
    build: .
    ports:
      - "8000:5000"
    volumes:
      - .:/code
    environment:
      FLASK_DEBUG: "true"
  redis:
    image: "redis:alpine"
```

The new `volumes` key mounts the project directory (current directory) on the host to `/code` inside the container, allowing you to modify the code on the fly, without having to rebuild the image. The `environment` key sets the `FLASK_DEBUG` environment variable, which tells `flask run` to run in development mode and reload the code on change. This mode should only be used in development.

Step 6: Re-build and run the app with Compose



From your project directory, type `docker compose up` to build the app with the updated Compose file, and run it.

```
$ docker compose up

Creating network "composetest_default" with the default driver
Creating composetest_web_1 ...
Creating composetest_redis_1 ...
Creating composetest_web_1
Creating composetest_redis_1 ... done
Attaching to composetest_web_1, composetest_redis_1
web_1    |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
...
```

Check the `Hello World` message in a web browser again, and refresh to see the count increment.

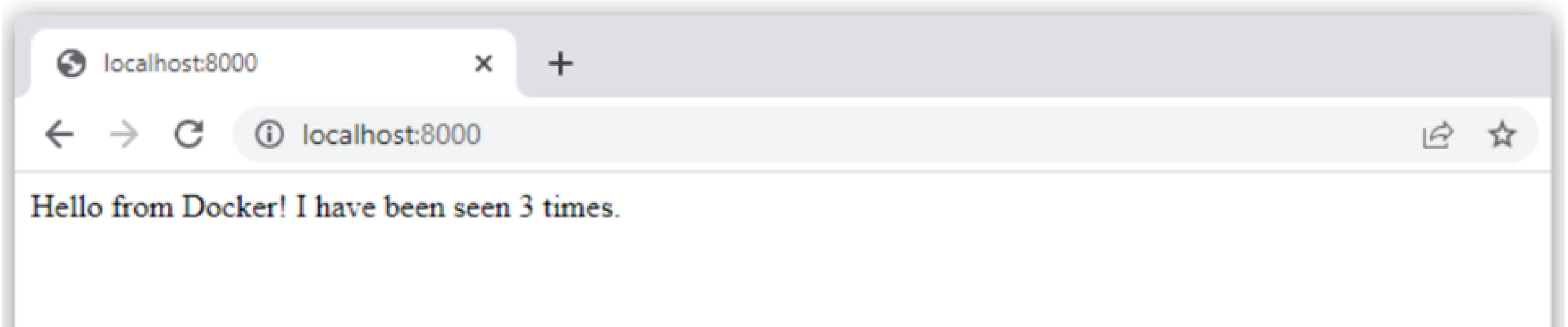
Step 7: Update the application

Because the application code is now mounted into the container using a volume, you can make changes to its code and see the changes instantly, without having to rebuild the image.

Change the greeting in `app.py` and save it. For example, change the `Hello World!` message to `Hello from Docker!` :

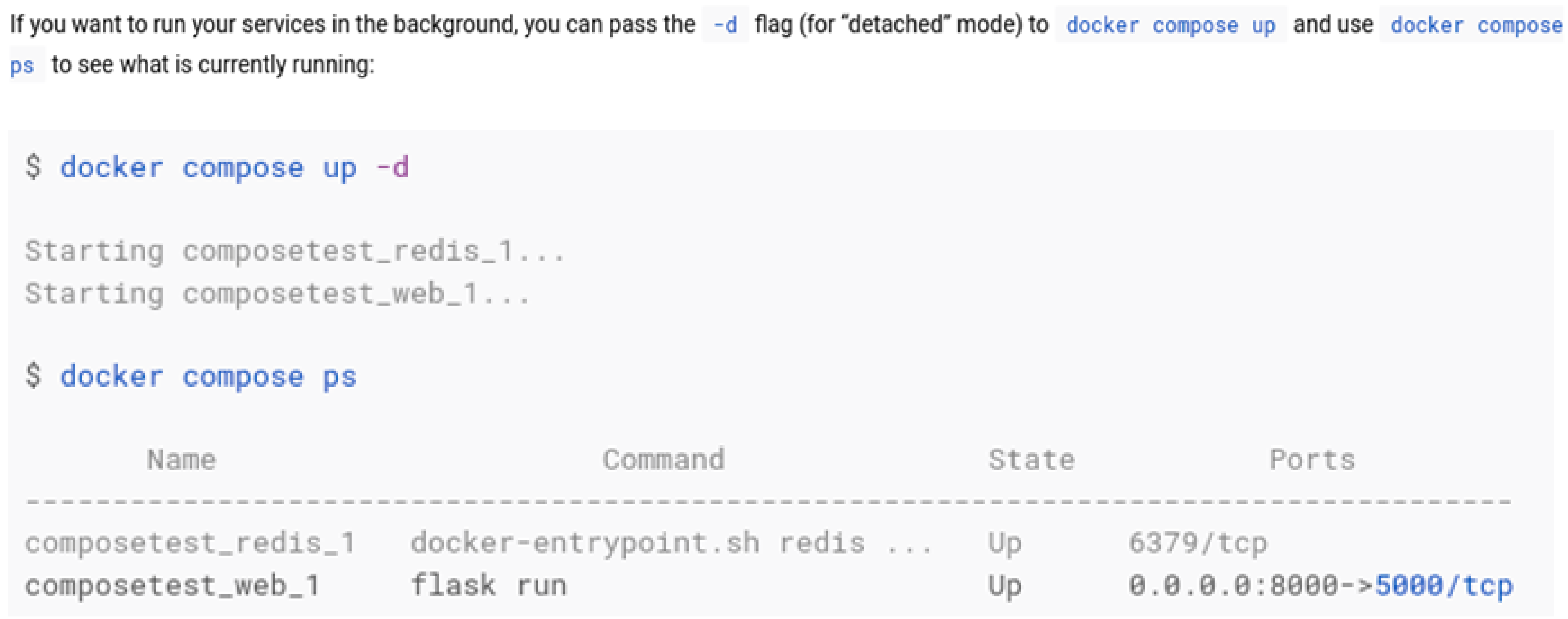
```
return 'Hello from Docker! I have been seen {} times.\n'.format(count)
```

Refresh the app in your browser. The greeting should be updated, and the counter should still be incrementing.



Hello from Docker! I have been seen 3 times.

Step 8: Experiment with some other commands



If you want to run your services in the background, you can pass the `-d` flag (for "detached" mode) to `docker compose up` and use `docker compose ps` to see what is currently running:

```
$ docker compose up -d

Starting composetest_redis_1...
Starting composetest_web_1...

$ docker compose ps

       Name                      Command               State           Ports
-------------------------------------------------------------------------------------
composetest_redis_1   docker-entrypoint.sh redis ...   Up      6379/tcp
composetest_web_1     flask run                        Up      0.0.0.0:8000->5000/tcp
```

Step 8: Experiment with some other commands

The `docker compose run` command allows you to run one-off commands for your services. For example, to see what environment variables are available to the `web` service:

```
$ docker compose run web env
```

See `docker compose --help` to see other available commands.

If you started Compose with `docker compose up -d`, stop your services once you've finished with them:

```
$ docker compose stop
```

You can bring everything down, removing the containers entirely, with the `down` command. Pass `--volumes` to also remove the data volume used by the Redis container:

```
$ docker compose down --volumes
```

Ways to Set Environment Variables in Compose

Environment variables

Environment variables can help you define various configuration values. They also keep your app flexible and organized.

We will cover:

- The various ways you can set environment variables in Compose.
- How environment variable precedence works.
- The correct syntax for an environment file.
- Changing pre-defined environment variables.

Substitute with an .env file

The `.env` file is useful if you have multiple environment variables you need to store.

Below is a simple example:

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

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

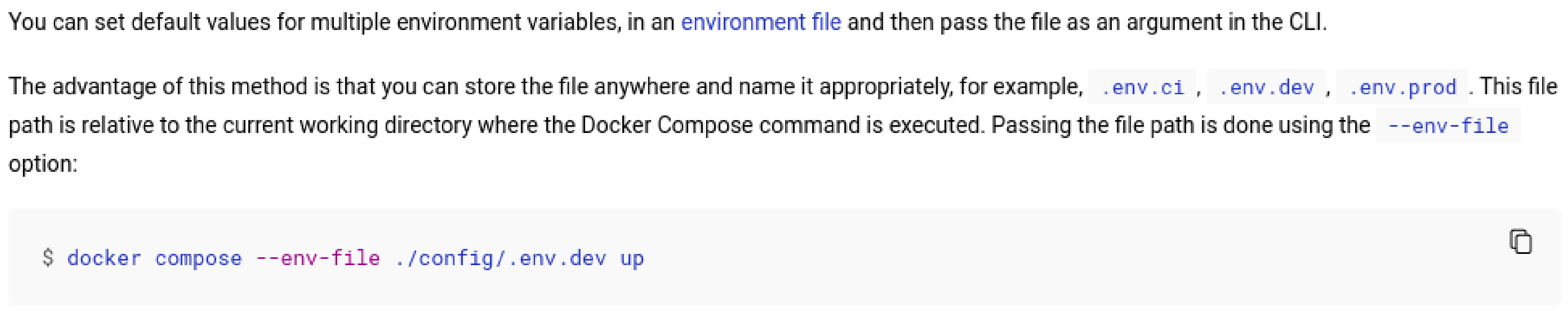
When you run `docker compose up`, the `web` service defined in the Compose file substitues in the image `webapp:v1.5` which was set in the `.env` file. You can verify this with the convert command, which prints your resolved application config to the terminal:

```
$ docker compose convert

services:
  web:
    image: 'webapp:v1.5'
```

The `.env` file should be placed at the root of the project directory next to your `docker-compose.yml` file. You can use an alternative path with one of the following methods:

- The `--file` option in the CLI
- The `--env-file` option in the CLI

Substitute with --env-file



You can set default values for multiple environment variables, in an environment file and then pass the file as an argument in the CLI.

The advantage of this method is that you can store the file anywhere and name it appropriately, for example, `.env.ci`, `.env.dev`, `.env.prod`. This file path is relative to the current working directory where the Docker Compose command is executed. Passing the file path is done using the `--env-file` option:

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

Use the environment attribute

You can set environment variables in a service's containers with the `environment` attribute in your Compose file. It works in the same way as `docker run -e VARIABLE=VALUE ...`

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

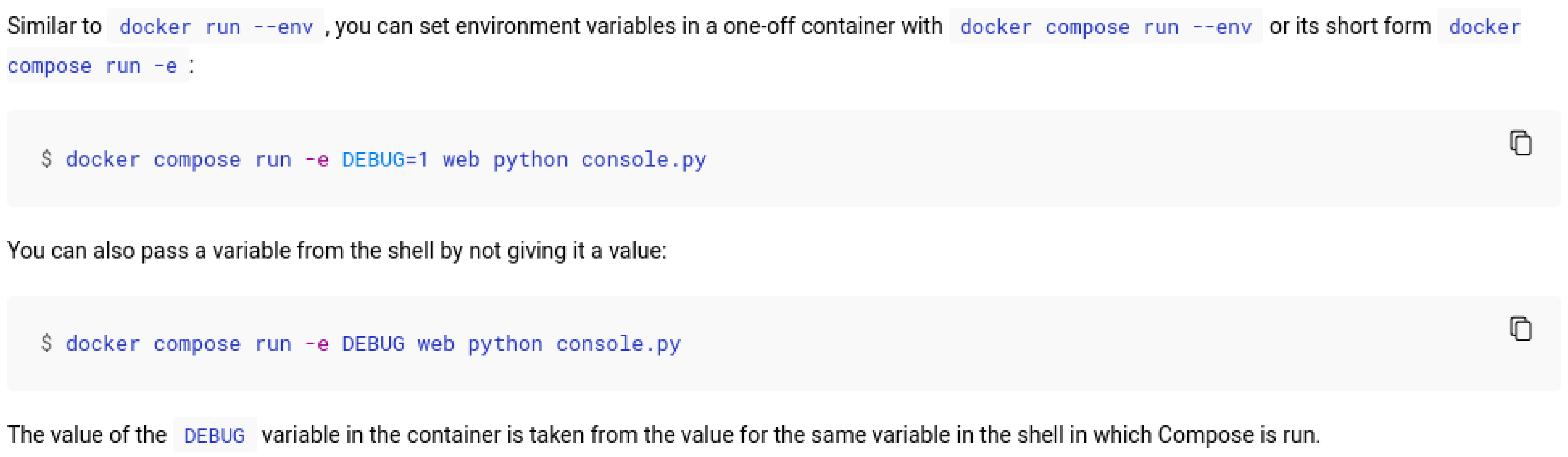
Use the env_file attribute

You can pass multiple environment variables from an external file through to a service's containers with the `env_file` option. This works in the same way as `docker run --env-file=FILE ...`:

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

If multiple files are specified, they are evaluated in order and can override values set in previous files.

Set environment variables with docker compose run --env



Similar to `docker run --env`, you can set environment variables in a one-off container with `docker compose run --env` or its short form `docker compose run -e`:

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

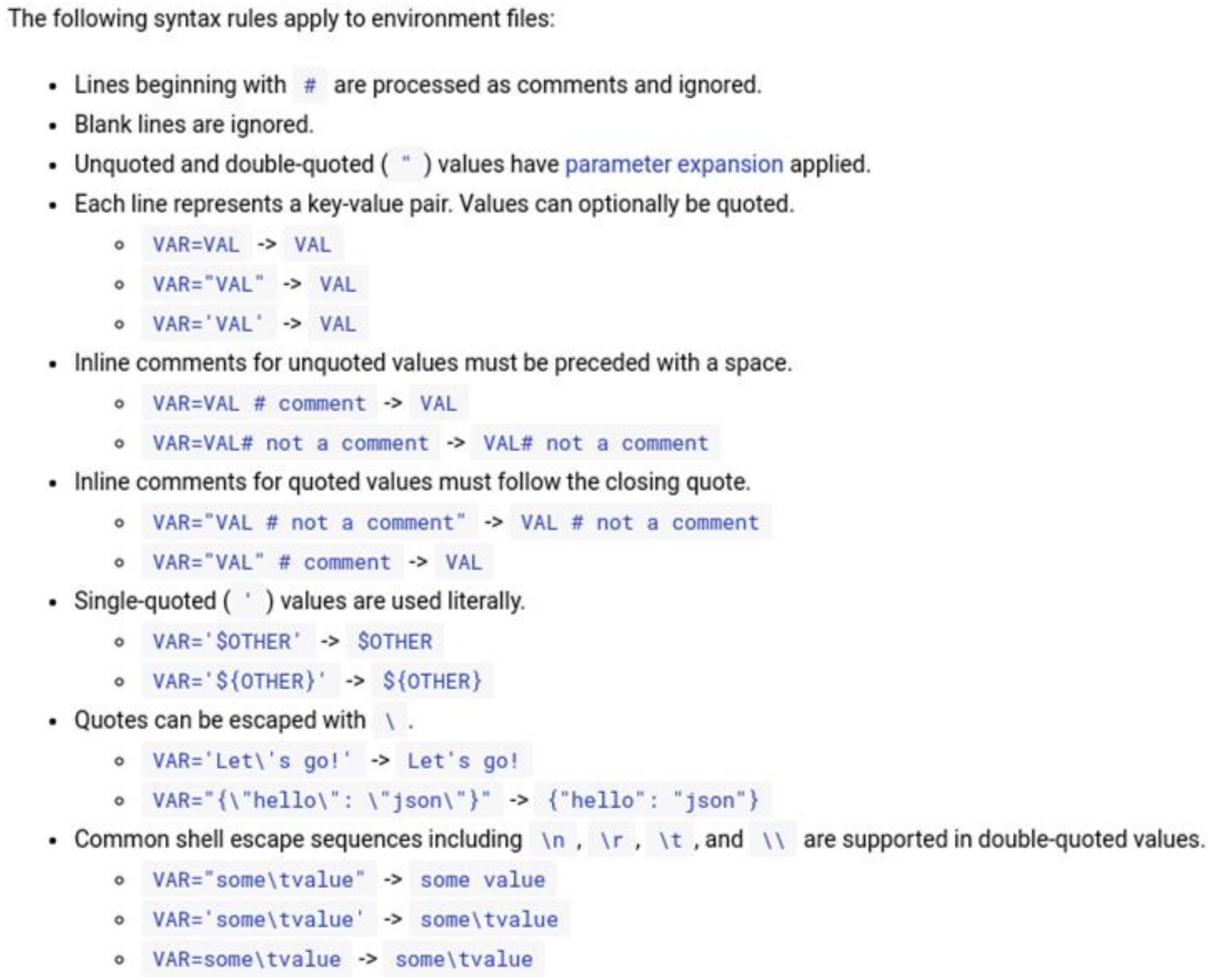
You can also pass a variable from the shell by not giving it a value:

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

The value of the `DEBUG` variable in the container is taken from the value for the same variable in the shell in which Compose is run.

Use an Environment File

Syntax



The following syntax rules apply to environment files:

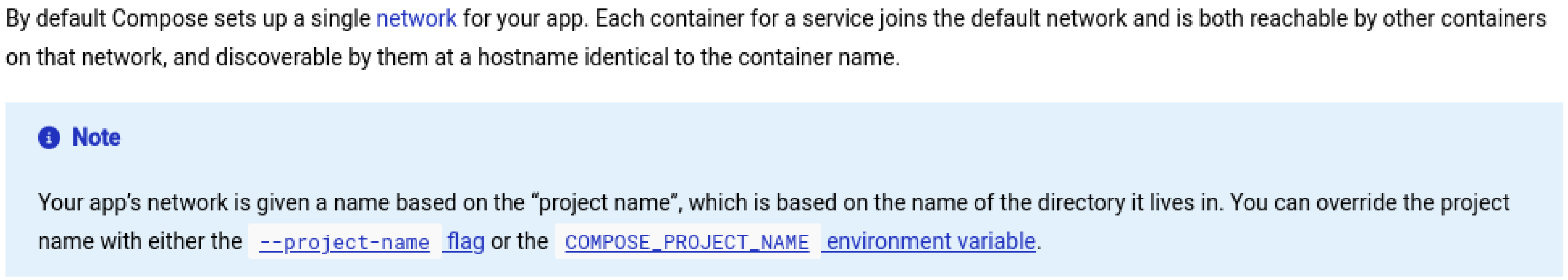
- Lines beginning with `#` are processed as comments and ignored.
- Blank lines are ignored.
- Unquoted and double-quoted (`"`) values have parameter expansion applied.
- Each line represents a key-value pair. Values can optionally be quoted.
  - `VAR=VAL` -> `VAL`
  - `VAR="VAL"` -> `VAL`
  - `VAR='VAL'` -> `VAL`
- Inline comments for unquoted values must be preceded with a space.
  - `VAR=VAL # comment` -> `VAL`
  - `VAR=VAL# not a comment` -> `VAL# not a comment`
- Inline comments for quoted values must follow the closing quote.
  - `VAR="VAL # not a comment"` -> `VAL # not a comment`
  - `VAR="VAL" # comment` -> `VAL`
- Single-quoted (`'`) values are used literally.
  - `VAR='$OTHER'` -> `$OTHER`
  - `VAR='${OTHER}'` -> `${OTHER}`
- Quotes can be escaped with `\` .
  - `VAR='Let\'s go!'` -> `Let's go!`
  - `VAR="{\"hello\": \"json\"}"` -> `{"hello": "json"}`
- Common shell escape sequences including `\n` , `\r` , `\t` , and `\\` are supported in double-quoted values.
  - `VAR="some\tvalue"` -> `some value`
  - `VAR='some\tvalue'` -> `some\tvalue`
  - `VAR=some\tvalue` -> `some\tvalue`

Enabling GPU access with Compose

- Compose services can define GPU device reservations if the Docker host contains such devices and the Docker Daemon is set accordingly.
- For MLOps related stuff

Networking in Compose

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.

Note

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

For example, suppose your app is in a directory called `myapp`, and your `docker-compose.yml` looks like this:

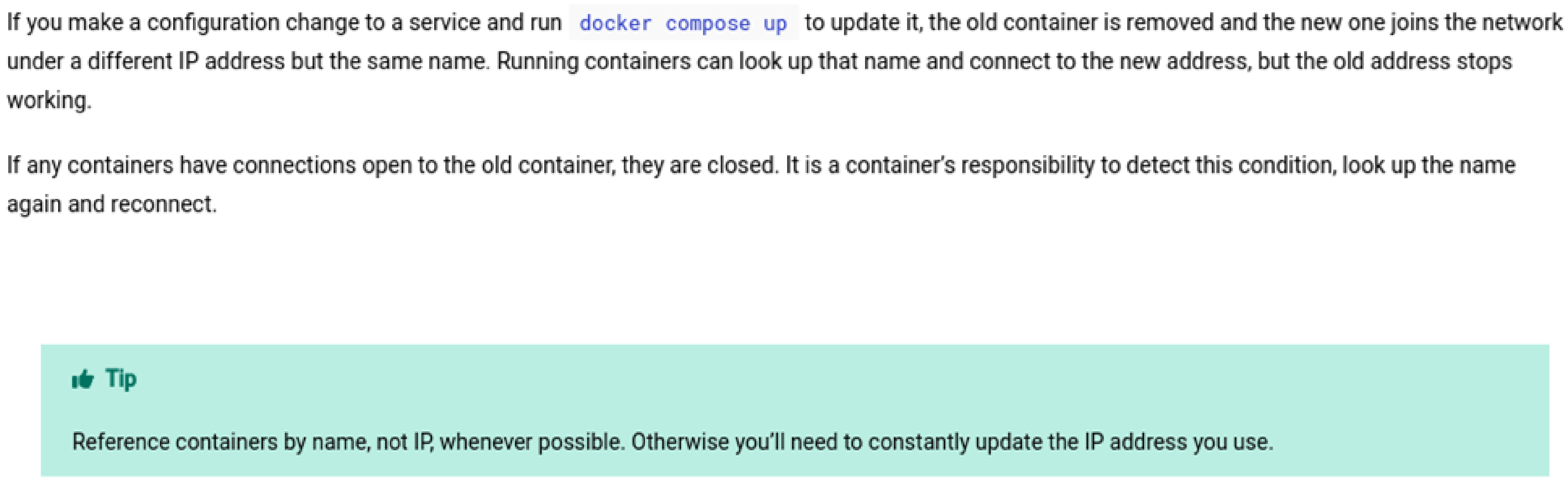
```yaml
services:
  web:
    build: .
    ports:
      - "8000:8000"
  db:
    image: postgres
    ports:
      - "8001:5432"
```

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

1. A network called `myapp_default` is created.
2. A container is created using `web`'s configuration. It joins the network `myapp_default` under the name `web`.
3. A container is created using `db`'s configuration. It joins the network `myapp_default` under the name `db`.

Each container can now look up the hostname `web` or `db` and get back the appropriate container's IP address. For example, `web`'s application code could connect to the URL `postgres://db:5432` and start using the Postgres database.

Update Containers on the Network



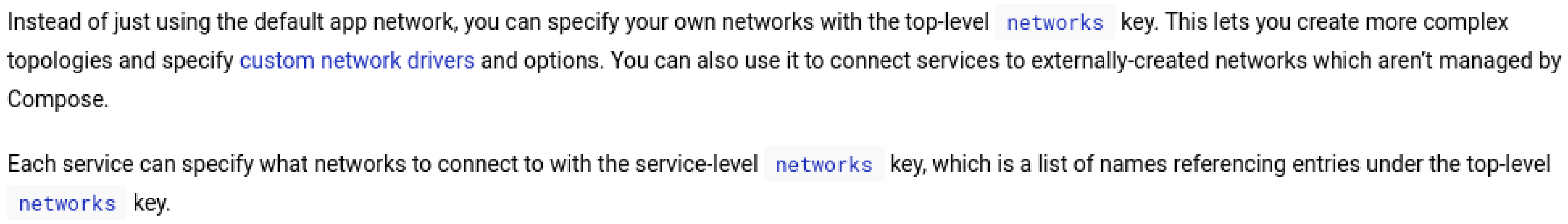
If you make a configuration change to a service and run `docker compose up` to update it, the old container is removed and the new one joins the network under a different IP address but the same name. Running containers can look up that name and connect to the new address, but the old address stops working.

If any containers have connections open to the old container, they are closed. It is a container's responsibility to detect this condition, look up the name again and reconnect.

Tip

Reference containers by name, not IP, whenever possible. Otherwise you'll need to constantly update the IP address you use.

Specify Custom Networks



Instead of just using the default app network, you can specify your own networks with the top-level `networks` key. This lets you create more complex topologies and specify custom network drivers and options. You can also use it to connect services to externally-created networks which aren't managed by Compose.

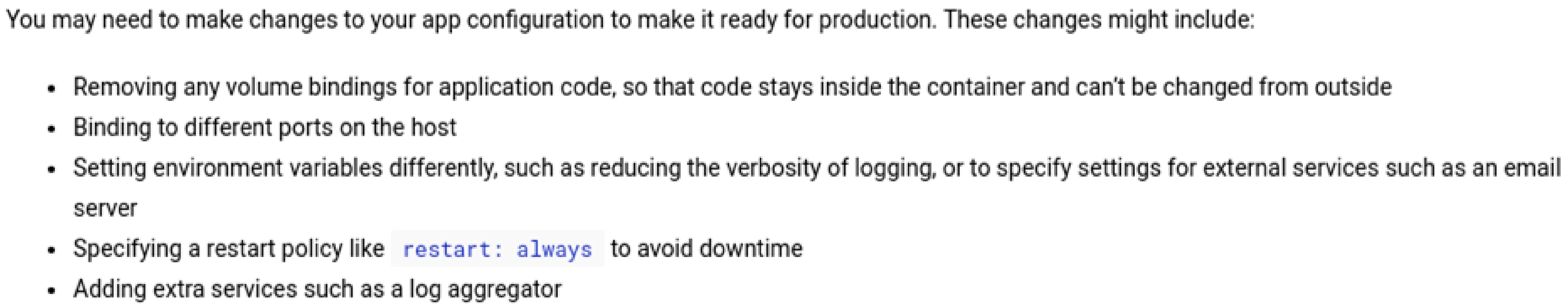
Each service can specify what networks to connect to with the service-level `networks` key, which is a list of names referencing entries under the top-level `networks` key.

The following example shows a Compose file which defines two custom networks. The `proxy` service is isolated from the `db` service, because they do not share a network in common. Only `app` can talk to both.

```
services:
  proxy:
    build: ./proxy
    networks:
      - frontend
  app:
    build: ./app
    networks:
      - frontend
      - backend
  db:
    image: postgres
    networks:
      - backend

networks:
  frontend:
    # Use a custom driver
    driver: custom-driver-1
  backend:
    # Use a custom driver which takes special options
    driver: custom-driver-2
    driver_opts:
      foo: "1"
      bar: "2"
```

Networks can be configured with static IP addresses by setting the ipv4_address and/or ipv6_address for each attached network.

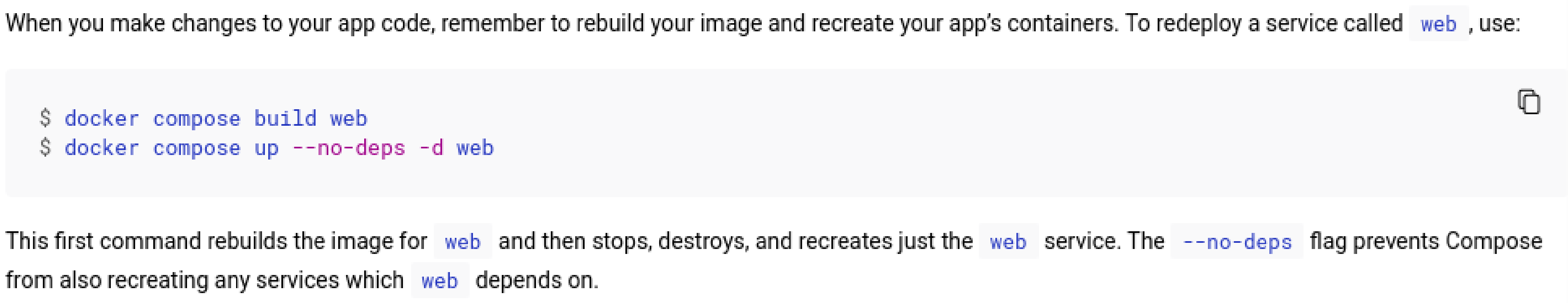
Modify your Compose file for Production



You may need to make changes to your app configuration to make it ready for production. These changes might include:

- Removing any volume bindings for application code, so that code stays inside the container and can't be changed from outside
- Binding to different ports on the host
- Setting environment variables differently, such as reducing the verbosity of logging, or to specify settings for external services such as an email server
- Specifying a restart policy like `restart: always` to avoid downtime
- Adding extra services such as a log aggregator

Deploying Changes



When you make changes to your app code, remember to rebuild your image and recreate your app's containers. To redeploy a service called `web`, use:

```
$ docker compose build web
$ docker compose up --no-deps -d web
```

This first command rebuilds the image for `web` and then stops, destroys, and recreates just the `web` service. The `--no-deps` flag prevents Compose from also recreating any services which `web` depends on.

LAB: Sample apps with Compose

LAB 1: Compose and Django

- Tutorial Link:
  - https://github.com/docker/awesome-compose/blob/master/official-documentation-samples/django/README.md

LAB 2: Compose and WordPress

- Tutorial Link:
    - https://github.com/docker/awesome-compose/blob/master/official-documentation-samples/wordpress/README.md

LAB 3: Deploying Portainer

- Tutorial Link:
  - https://github.com/docker/awesome-compose/tree/master/portainer

LAB 4: Deploying a local private Docker Registry

- Tutorial Link:
  - https://github.com/slydeveloper/docker-registry-joxit-ui-compose

LAB 5: Deploying flask application using Compose

- Tutorial Link:
  - https://github.com/docker/awesome-compose/tree/master/flask

LAB 6: NextCloud using Compose

- Tutorial Link:
  - https://github.com/docker/awesome-compose/tree/master/nextcloud-postgres

Assignment 2

- **Topic:** Docker Network & Docker Compose
- Goto Portal to Download
- Deadline Next Friday Night (**26-JAN-2024**)

The End