Dev Environments

Why and how you should be composing with Docker Compose

Alec Fong

August 10, 2022 · 7 read

How to manage multi-service, multi-environment dev environments/applications - with Docker Compose

---

If you're building an application with server-side logic you probably have run into the challenge of replicating your services in a local and isolated manner. The application may need a database, cache, worker, and/or other services. There are several ways to address this problem, but in this article, we will be looking at using Docker Compose to manage a multi-service, multi-environment application.

Docker compose is a tool designed to help manage your docker containers through configuration instead of through CLI commands. Docker Compose simplifies managing multiple services, networking, and logging. Due to Docker Compose's relative simplicity and lightweight, it can be a useful dev playground even when using other container orchestration tools like Kubernetes or Nomad and especially useful if using container cloud solutions like Google Cloud Run, AWS ECS, or Azure Containers.

One of Docker Compose's most powerful features is also one of its most under-leveraged: config composability. Docker Compose merges multiple config files together allowing you to easily separate concerns, and create multiple different environments all while not repeating yourself.

Managing multiple environments can be a bit tricky. Let's take a look at an example; we are developing two services:

- API server
- background job worker

These services depend on two third-party dependencies:

- mysql database
- redis cache

A standard way to write your docker-compose file might be like this.

TL;DR

Check out the **complete source code**

```
# docker-compose.yml
version: '3.9'
services:
  server:
    build:
      context: .
    ports:
      - '8000:8000'
  worker:
    build:
      context: .
  database:
    image: mysql
    ports:
      - '3306:3306'
  cache:
    image: redis
    ports:
      - '6379:6379'
```

```
$ docker-compose -f docker-compose.yml up
```

This configuration is a great start! The server and worker gets built and run with access to the helper services: database and cache. This can satisfy simple application setups, but now let's add another variable to our application development.

What if we have multiple ways to build and run our services? What if we wanted to simulate “prod”, “staging”, or “dev"? There are several ways this can be achieved depending on how the environments differ. The naive solution might be to have separate docker-compose files for each environment. This works just fine but may become tedious if there are lots of similarities and many services to manage. Changing one file may mean you need to change all— this can quickly become tedious to maintain.

Our solution leverages Docker Compose’s composability! Imagine our server and worker have different images, environment variables, and start commands for “dev” and “prod”. Let’s create a base file and then apply our environment-specific changes to the base.

```
# base.yml
version: '3.9'
services:
  server:
    build:
      context: .
    ports:
      - '8000:8000'
  worker:
    build:
      context: .
  database:
    image: mysql
    ports:
      - '3306:3306'
  cache:
    image: redis
    ports:
      - '6379:6379'
```

Our base.yml file specifies the configuration default and shared values.

```
# dev.yml
version: "3.9"
services:
  server:
    build:
            target: dev
        env_file: [dev.env]
  worker:
    build:
            target: dev
        env_file: [dev.env]
```

The dev.yml specifies dev-specific configuration.

```
# prod.yml
version: "3.9"
services:
  server:
    build:
            target: prod
        env_file: [dev.env]
  worker:
    build:
            target: prod
        env_file: [prod.env]
```

The prod.yml specifies prod-specific configuration.

To run the dev environment specify both the base and dev file:

```
$ docker-compose -f base.yml -f dev.yml up
```

To run the prod environment specify both the base and prod file:

```
$ docker-compose -f base.yml -f prod.yml up
```

To debug how the files are merged together run:

```
$ docker-compose -f base.yml -f {env}.yml config
```

Using Docker Compose with multiple files is a powerful way to maintainably and extensibly manage services. To learn more check out **Docker's official documentation**.

If working on developer experience problems like these interest you, check out **brev.dev**

Previous

← Don't let a bad abstraction cost you 2 years

Next

Harness Multi-stage builds to create optimal images →