Setting up a local development environment using Docker Compose

A step-by-step guide to creating a containerized environment for your application development needs

Mihir Patel

.

Follow

Published in

Simform Engineering

.

9 min read

.

May 4, 2023

--

2

Listen

Share

Setting up a local development environment using Docker Compose

Docker Compose is a handy tool that helps you create and run applications that use multiple containers. It makes the process of setting up and deploying complex applications easier by letting you define everything you need in one file. With Docker Compose, you can manage all your containers together and easily control how they interact.

In this blog, we will explore the benefits of using Docker Compose and how it can help you streamline your Docker-based application development and deployment. We will cover the basics of Docker Compose, including how to define services, networks, and volumes in a Compose file, and how to use Docker Compose to start, stop, and manage your application stack.

What is docker-compose?

> Docker Compose is a tool that helps spin up multiple containers with a single command. With Docker Compose, we can create a YAML file that defines how to create multiple services (containers) and spin up or tear it all down using a single command.

Installation:

1. How to install docker-compose on **Linux**:

- **Linux**: https://docs.docker.com/compose/install/linux/

2. How to install docker-compose on **Windows/Mac**:

> **Note:** For **Windows /Mac**, Docker Compose is pre-installed with Docker Desktop, so you need to install Docker Desktop for **Windows/Mac** to use Docker Compose.

- **Windows**: https://docs.docker.com/desktop/install/windows-install/
- **Mac**: https://docs.docker.com/desktop/install/mac-install/

Sample docker-compose file:

```
version: "3.8"
services:
 service-1:
   build:
     context: pathOfCodeFolder
     dockerfile: dockerFile name
   image: dockerImageName
   ports:
     - "hostPort:ContainerPort"
   entrypoint: ["/bin/sh","entrypoint.sh"]
   command: npm run dev
   restart: always
   container name: contianer-front-end
   environment:
     key: value
   env_file:
     - pathOfEnvFile
   networks:
     - networkName
   volumes: # this is called as volume binding
       - volumeName:ContainerPath
   volumes: # this is called as host binding
     - hostPath:containerPath
   deploy:
     resources:
       limits:
         cpus: '2'
         memory: 512M
       reservations:
         cpus: '0.25'
         memory: 64M
   depends_on:
     - serviceName


 service-2:
   ...
networks:
 network_name:
   driver: <driverType>
voulmes:
 volumeName
```

Overview of Docker Compose File Parameters:

- **version**: This specifies the version of the docker-compose file.
- **services**: With the help of services in Docker Compose, we can create multiple services such as front-end, back-end, and database services, among others.
- **build**: With the **`build`** keyword in Docker Compose, we can create an image for the service. The **`context`** keyword is used to specify the directory where the Dockerfile is located. Additionally, with the **`dockerfile`** keyword, we can specify a custom name for the Dockerfile to use.
- **image**: If we want to use pre-built images from the Docker repository, such as DockerHub, we only need to specify the image name in the Docker Compose file.

  Note: When we use both **`build`** and **`image`** keywords in a Docker Compose file, the image will be built using the context specified in the **`build`** keyword, and the **`image`** keyword will assign a name to the new image. This new image will then be listed under **`docker images`** command.

- **port**: With the help of port exposing, we can access a site running in a container from the local host. On the left, we define the host port, and on the right, we define the container port. For example, if a Node container is running on port 4000, and we want to access it from the host machine on port 80, we can define the port in the compose file like this: **`80:4000`** (hostPort:containerPort).
- **Entrypoint and command:**

  Note: In Docker Compose, there is a difference between **`entrypoint`** and **`command`**. When an **`entrypoint`** is defined for a service, it cannot be overridden, and we can run executable commands such as migrations and seeding using the **`entrypoint`**. On the other hand, the **`command`** can be overridden, and we can specify multiple commands to run in a single service. Here is an example code snippet

```
command: >
   bash -c "python manage.py migrate
   && python manage.py runserver 0.0.0.0:8000"
```

- **environment**: In Docker Compose, we can specify environment variables at runtime using the **`environment`** keyword, as shown in the example code snippet above.
- **environment_file**: In cases where we need to specify a whole file containing environment variables, we can use the **`environment_file`** keyword, as shown in the example code snippet above.
- **container_name**: The `container_name` keyword in Docker Compose allows us to specify a custom name for a container.

- **restart policy**: In Docker Compose, it is important to specify a restart policy for each service. For example, if the restart policy is set to “always”, the Docker container will automatically restart if the host machine is rebooted. If not specified, then the container will not restart at the time of host restart or reboot. More information about restart policies can be found in the Docker documentation.
- **depends_on**: When a service depends on another service, the **depends_on** keyword can be used. For example, suppose the backend container needs the database container to be running before it can start. In that case, we can use **depends_on** in the backend service so that the database container is created before the backend container starts.
- **networks**: As shown in the code snippet above, we can create networks and attach them to specific services in Docker Compose. There are several types of networks available, including bridge, host, and overlay networks. For more information about Docker networks, please visit the official Docker documentation.
- **volumes**: We need to use volume mounting when we require persistent storage for data, which should survive container stop or removal and start again.

1. **host mounting**: In Docker Compose, we can use host-based mounting to achieve immediate code changes in the website without restarting the container. For instance, if we are running a front-end and a back-end service, we can make code changes on the host machine, and they will be immediately reflected in the container with the help of host-based mounting.

```
volumes:
     - 'hostPathOfCode:containerDirName'
```

2. **volume mounting**: To persist the volume of a database container, we can create a volume and attach it to the database service, as shown in the following code snippet.

```
volumes:
     - 'backend-db:containerDirName'

volumes:
 backend-db:
```

Resource limit on docker:

If we do not set resource limits on a container in Docker Compose, it will use the whole RAM and CPU limit of the host PC. To check the RAM and CPU usage of a container, you can use the docker stats command.

The below image shows a container without RAM and CPU limits set:

Containers without set RAM and CPU limits

The below image shows a container after RAM and CPU limits are set:

Containers after set RAM and CPU limits

```
deploy:
     resources:
       limits:
         cpus: '2'
         memory: 512M
       reservations:
         cpus: '0.25'
         memory: 64M
```

As shown above, I have set a limit of 512MB on each container. This means that each container has a maximum RAM usage of 512MB.

By limiting RAM and CPU for each container, we can protect host RAM and CPU from overutilization.

Let’s set up the local development environment using docker-compose

Example: How to create a Docker Compose file to run front-end, back-end, and database containers.

```
version: '3.8'

services:
 postgres:
   image: postgres:latest
   container_name: postgres
   ports:
     - '5432:5432'
   environment:
     POSTGRES_PASSWORD: helloworld
     POSTGRES_USER: test
     POSTGRES_DB: testdb
   volumes:
     - 'backend-db:/var/lib/postgresql/data'
   deploy:
     resources:
       limits:
         cpus: '2'
         memory: 512M
       reservations:
         cpus: '0.25'
         memory: 64M
   networks:
     - application

 frontend:
   depends_on:
     - backend
   build:
     context: ./boilerplate_next/
     dockerfile: Dockerfile
   image: compose_next
   container_name: frontend
   ports:
     - '3000:3000'
   volumes:
     - './boilerplate_next:/app'
     - '/app/node_modules'
     - '/app/.next'
   deploy:
     resources:
       limits:
         cpus: '2'
         memory: 512M
       reservations:
         cpus: '0.25'
         memory: 64M
   networks:
     - application

 backend:
   depends_on:
     - postgres
   build:
     context: ./server-js/
     dockerfile: Dockerfile
   image: compose_node
   container_name: backend
   volumes:
     - './server-js/:/app'
   ports:
     - '8000:8000'
   deploy:
     resources:
       limits:
         cpus: '2'
         memory: 512M
       reservations:
         cpus: '0.25'
         memory: 64M
   networks:
     - application

networks:
 application:
   driver: bridge
volumes:
 backend-db:
```

Note: We need to mount the node_modules and .next folder of frontend because the volume is not mounted during the build.

Dockerfiles:

Front-end dockerfile:

```
FROM node:18.16.0-slim

WORKDIR /app

COPY package.json .

RUN npm install

COPY . .

EXPOSE 3000

RUN npm run build

CMD ["npm","run","start"]
```

Back-end Dockerfile:

```
FROM node:16.15.0-alpine

WORKDIR /app

COPY package.json ./

RUN npm i

COPY . ./

EXPOSE 8000

RUN ["chmod","+x","entrypoint.sh"]


Entrypoint.sh

npm install

npx sequelize db:migrate --config config/config.js

npx sequelize db:seed --seed ./backend/seeders/20220630094531-create-user.js

npm run dev
```

For **Postgres** database we have used pre-built docker image as **postgres:latest**

Explanation:

1. Here we need to run three services front-end, back-end, and Postgres, so create three services as shown above.
2. *Best practice:* In each service, we should define the port to access the service from the host, specify whether to build or use an existing image, mount volumes to persist data and mount code from the host to the container, set resource limits, apply networks, and assign a container name.
3. To check the log of the container: **docker logs <container_name of contaienr_id>**
4. To run compose file: **docker-compose up -d** (-d used for running container in the background)
5. To down compose file: **docker-compose down** (Stop and remove containers, networks it will not remove volume; to remove volume use **-v**).

Here is the output of the above code.
Run: docker-compose up

Access front-end by 3000 port define in docker-compose

Note: To connect the backend container with the database container, use the service names as the host name in the environment variables of the backend container.

Hot reloading:
Reloading the application without restarting the container

Front-end: To reload the application while changing code in the host code without restarting the container, we need to follow two steps.

1. We have to mount a volume in compose file.
2. Use development commands like `npm run dev` .

Back-end: To update the containerized application without rebuilding it when changes are made to the host code, we need to mount a volume as shown in the code above and add **nodemon** functionality to the package.json file.
For example, If you make changes to the front-end code, Docker Compose can automatically reflect those changes in the running application without requiring you to recreate the container.

Here, I made a change to one line of code.
Without restarting the container, changes are reflected in the running application

Troubleshooting commands:

- To remove all Docker images that are not attached to any container: **`docker rmi -f `docker imaages -aq``**
- To remove all stopped Docker containers: **`docker rm `docker ps -aq``**
- To go inside a container: **`docker exec -it <contianerid or name>`**
- To show the RAM and CPU usage of Docker: **`docker stats`**
- To check the logs of a container: **`docker logs <container_id_or_name>`**
- To build the images if the images **do not exist** and start the containers: **`docker-compose up`**
- To build the images if the images **do not exist** and start the containers in the background: **`docker-compose up -d`**
- To build images before starting containers: **`docker-compose up --build`**
- To stop containers and remove containers, networks, volumes, and images created by up: **`docker-compose down`**

  By default **`docker-compose down`** does not remove volume to remove use **`docker-compose down -v`**

Wrapping Up ■

With the help of Docker Compose, developers can easily spin up a full-stack application in a local environment.

By setting up the application in a local environment, we can debug issues related to the front-end, back-end, and database. This enables us to quickly identify and fix any issues that may arise during development.

Another advantage of using Docker Compose is its ability to replicate the production environment locally. By defining the production environment in the Docker Compose configuration file, developers can easily recreate it on their local machines, eliminating the need for expensive hardware and infrastructure. This results in a faster development cycle and more accurate testing, leading to a more stable and reliable application.

Follow Simform Engineering to keep up with all the latest trends and insights in the development ecosystem.