Building a Simple 3-Tier Architecture with Docker Compose: A Hands-On Project Journey

Kesara Karannagoda

.

Follow

15 min read

.

Jul 9, 2023

--

Listen

Share

It’s been a long time since I posted my last article about Docker. If you haven’t had the chance to read it yet, I would like to extend an invitation for you to explore it (From OS Hassles to Team Harmony: How Docker Saved Our Group Project). This article offers valuable insights into Docker and how it played a pivotal role in the successful containerization of our university group project.

> Initially, my intention was to provide a tutorial outlining the process of containerizing our project. However, I have taken this opportunity to incorporate additional details that were not available to me at the time of doing my project. I hope you find this enhanced version enjoyable and informative as you navigate through it.

Prerequisites:

Before proceeding with this tutorial, please ensure that you have met the following prerequisites:

1. **An active Internet connection:** You will need a reliable Internet connection to access the necessary resources and documentation throughout the tutorial.
2. **Docker installation:** Make sure that Docker is installed on your machine. Docker enables the creation and management of containers, which is crucial for following along with the tutorial. **You can follow the below instructions to install it.**

Docker Installation

Please run the following commands in your terminal (for Linux and MacOS) or PowerShell (for windows) in order to check the presence of Docker installation on your computer.

```
$ docker version
$ docker compose version
```

Docker is a platform that allows you to create and manage containers, while **Docker Compose** is a tool specifically designed to simplify the management of multi-container applications.

> ***Docker Compose comes with the docker installation. You don’t have to install it separately.***

In case Docker is not installed on your computer, kindly refer to the instructions provided in the official Docker documentation for proper installation guidance.
Docker Installation (Official Docker Documentation)

A Simple 3-Tier Architecture

Today, our objective is to build a basic 3-tier architecture using Docker containers. This architecture has three components.

A simple 3-tier architecture using docker containers.

1. A frontend container
2. A backend container (With an API layer)
3. A MySQL database container

Frontend App Container

frontend container

To start our project, Let's create a new folder to maintain it as our main project folder (You can name it as you want. Ex: ***`docker-tutorial`***). Within the main folder, we'll create another folder called **`frontend`** to handle our frontend application. After that we can add a simple **`index.html`** file inside the frontend folder. This will help us to check whether the frontend app is working or not once it is deployed.

index.html

```
<html lang="en">
 <head>
   <meta charset="UTF-8" />
   <meta name="viewport" content="width=device-width, initial-scale=1.0" />
   <title>HTML Web Page</title>
 </head>
 <body>
   <h1>Hello, World!</h1>
 </body>
</html>
```

Next, we will create a new file named **`docker-compose.yml`** in our main project directory as our configuration file for deploying the frontend application within a Docker container.

docker-compose.yml

```
version: "3"
services:
 frontend:
   image: httpd:latest
   volumes:
     - "./frontend:/usr/local/apache2/htdocs"
   ports:
     - 3000:80
```

Let's discuss a little bit about our **`docker-compose.yml`** configuration.

- **`version: "3"`** : This tells us the version of the `Docker Compose` file format.
- **`services`** : This section contains the services to be deployed. (In our case, frontend container)
- **`frontend`** : This is the name of the service. You can choose any name you prefer. Ex: `frontend-service`
- **`image: httpd:latest`** : This line tells the Docker image to be used for the frontend service. In our case, we use the `httpd` image with the `latest` tag. `httpd` image is the official image for the Apache HTTP Server. You can find more images in Docker Hub.
- **`volumes:`** : This section is used to mount the local filesystem directory (File directory in your computer) to the container.
- **`- "./frontend:/usr/local/apache2/htdocs"`** : This line specifies the volume mapping. It maps the `./frontend` directory from the local filesystem to the `/usr/local/apache2/htdocs` directory inside the container. This allows the container to access the files in the frontend directory.
- **`ports:`** : This section defines the port mapping between the host (Your computer) and the container.
- **`- 3000:80`** : This line maps port 3000 of the host machine (Your computer) to port 80 of the container. It means that when the container is running, you can access the frontend application on http://localhost:3000 in your web browser.

Main Project Directory

After all the settings, you should have a folder structure like this.

```
.
■■■ frontend                # Frontend files. (.html/.js/.css)
■   ■■■ index.html          # html page
■■■ docker-compose.yml
```

What is docker-compose?

Docker compose is a tool that was developed to help define and share multi-container applications. docker-compose reads configuration data from a **`YAML`** file. You can learn more about docker-compose from here.

Next, Open up the terminal(Linux, Mac), or powershell(Windows) in your project directory, and run the command.

```
$ docker compose up
```

> ***Note!***
>
> You need to have an Internet connection to perform above command for the first time.
> First time setup will take some time because it has to pull the `httpd` container image from the docker hub.

At this point, you should be able to see your frontend application at http://localhost:3000/. We've used `httpd docker container image` to serve this simple frontend app.

You can press `Ctrl + c` in the terminal to stop all the containers.

What is Docker Hub?

Docker Hub is the world's largest container image library and community for `container images`. It has over 100,000 **container images** from various software vendors. And It allows developers to store, manage, and distribute Docker container images.

What is a Docker Image?

A Docker image is a lightweight, standalone, and executable package that contains all the necessary components, including the code, runtime environment, system tools, libraries, and dependencies, required to run a specific application. **It serves as a blueprint for creating Docker containers.**

Backend API Container

backend container

Now Let's create our backend api container,
To create the backend API container, we will follow these steps:

1. Create a new folder named `api` inside the main project directory.
2. Inside the `api` folder, create a new PHP file named `index.php`.
3. COPY & PASTE the below code snippet to your `index.php` file.

Then let's add a simple todos api endpoint in our index.php file.

index.php

```
<?php
// Set the response content type to JSON
header('Content-Type: application/json');
header('Access-Control-Allow-Origin: http://localhost:3000');
header('Access-Control-Allow-Methods: GET, POST, OPTIONS');

// Create a response array
$response = [
   [
       "_id" => 1,
       "todo" => "Todo 1"
   ],
   [
       "_id" => 2,
       "todo" => "Todo 2"
   ],
   [
       "_id" => 3,
       "todo" => "Todo 3"
   ],
   [
       "_id" => 4,
       "todo" => "Todo 4"
   ]
];

// Encode the response array as JSON
$jsonResponse = json_encode($response);

// Output the JSON response
echo $jsonResponse;
```

Dockerfile

In this case, I'm not going to make the Backend API container using the approach we've used to create our frontend container. We used a pre-existing Docker image directly from the Docker Hub to create the frontend container.

Instead, we will create the Backend API container by writing the Docker Build process in a **Dockerfile** located in the main project directory.

What is a Dockerfile?

Dockerfiles are used in Docker to define the configuration and instructions necessary to build a Docker image. Dockerfiles provide a simple and declarative way to automate the creation of Docker images, ensuring consistency and reproducibility across different environments.

We'll talk more about Dockerfiles in my upcoming articles. For now, Let's add the below `Dockerfile` inside our main project directory by naming it `Dockerfile` and **you don't need to have an file extension for that file.**

> **Note!**
> If you're a beginner to **Docker**, You don't have to worry about Dockerfile implementation.
> For the purpose of this tutorial, you can simply copy and paste the provided code, and it will be sufficient to follow along.

Dockerfile

```
FROM ubuntu:20.04

LABEL maintainer="kesaralive@gmail.com"
LABEL description="Apache / PHP development environment"

ARG DEBIAN_FRONTEND=newt
RUN apt-get update && apt-get install -y lsb-release && apt-get clean all
RUN apt install ca-certificates apt-transport-https software-properties-common -y
RUN add-apt-repository ppa:ondrej/php

RUN apt-get -y update && apt-get install -y \
apache2 \
php8.0 \
libapache2-mod-php8.0 \
php8.0-bcmath \
php8.0-gd \
php8.0-sqlite \
php8.0-mysql \
php8.0-curl \
php8.0-xml \
php8.0-mbstring \
php8.0-zip \
mcrypt \
nano

RUN apt-get install locales
RUN locale-gen fr_FR.UTF-8
RUN locale-gen en_US.UTF-8
RUN locale-gen de_DE.UTF-8

# config PHP
# we want a dev server which shows PHP errors
RUN sed -i -e 's/^error_reporting\s*=.*/error_reporting = E_ALL/' /etc/php/8.0/apache2/php.ini
RUN sed -i -e 's/^display_errors\s*=.*/display_errors = On/' /etc/php/8.0/apache2/php.ini
RUN sed -i -e 's/^zlib.output_compression\s*=.*/zlib.output_compression = Off/' /etc/php/8.0/apache2/php.ini

# to be able to use "nano" with shell on "docker exec -it [CONTAINER ID] bash"
ENV TERM xterm

# Apache conf
# allow .htaccess with RewriteEngine
RUN a2enmod rewrite
# to see live logs we do : docker logs -f [CONTAINER ID]
# without the following line we get "AH00558: apache2: Could not reliably determine the server's fully qualified domain name"
RUN echo "ServerName localhost" >> /etc/apache2/apache2.conf
# autorise .htaccess files
RUN sed -i '/<Directory \/var\/www\/>/,/<\/Directory>/ s/AllowOverride None/AllowOverride All/' /etc/apache2/apache2.conf

RUN chgrp -R www-data /var/www
RUN find /var/www -type d -exec chmod 775 {} +
RUN find /var/www -type f -exec chmod 664 {} +

EXPOSE 80

# start Apache2 on image start
CMD ["/usr/sbin/apache2ctl","-DFOREGROUND"]
```

After above change, your folder structure should look like this.

```
.
■■■ api                    # Backend api files.
■   ■■■ index.php          # php page
■■■ frontend               # Frontend files. (.html/.js/.css)
■   ■■■ index.html         # html page
■■■ docker-compose.yml
■■■ Dockerfile
```

Adding the backend service to the docker-compose.yml

Now Let's update the `docker-compose.yml` file with below code.

You should add the below code snippet under the `services` section in your `docker-compose.yml` file. If you have any difficulties, feel free to visit the Github repository to the finalized version. **You can find the GitHub repository link and all other resource links in the end of this article.**

```
  backend:
   container_name: simple-backend
   build:
     context: ./
     dockerfile: Dockerfile
   volumes:
     - "./api:/var/www/html/"
   ports:
     - 5000:80
```

- **`backend:`**: This is the name of the service. It can be any name you choose.
- **`container_name: simple-backend`**: This line sets the name of the container to be created as "simple-backend". You can use any desired name for your container.
- **`build:`**: This section indicates that we will be building the Docker image for the backend service.
- **`context: ./`**: It specifies the build context, which is the path to the directory containing the Dockerfile. Since we have our docker file in the main project directory it is set to the current directory (`./`).
- **`dockerfile: Dockerfile`**: This line specifies the filename of the Dockerfile to be used for building the image. In this case, it is named "Dockerfile".
- **`volumes:`**: This section is used to mount the local filesystem directory to the container.
- **`- "./api:/var/www/html/"`**: This line maps the `./api` directory from the local filesystem (Your computer) to the `/var/www/html/` directory inside the container. (`/var/www/html/` path is the default web root in apache web servers) It allows the container to access the files in the `api` directory.
- **`ports:`**: This section defines the port mapping between the host and the container.
- **`- 5000:80`**: This line maps port `5000` of the host machine to port `80` of the container. It means that when the container is running, you can access the backend API on `http://localhost:5000` in your application.

After adding it, Let's rerun the terminal command `docker compose up` to build and run our `frontend` and `backend` containers.

If everything works well, You should be able to see your simple todos API at http://localhost:5000/ .

todos API endpoint at http://localhost:5000

Accessing the Backend from the Frontend

In order to call our backend API from frontend. We should allow access to our frontend in our backend code. We have already put it in the code. Below line helps frontend to call our simple todo api.

./api/index.php

```
header('Access-Control-Allow-Origin: http://localhost:3000');
```

> **Note!**
> If you're trying to expose the frontend from a different port instead of port `3000`. Please change the above line in below example.
> **Example:** If you're trying to expose it in port 3001.
> Update it as below.

```
header('Access-Control-Allow-Origin: http://localhost:3001');
```

Let's update our frontend code as below to call the backend API

```
<html lang="en">
 <head>
   <meta charset="UTF-8" />
```

```
   <meta name="viewport" content="width=device-width, initial-scale=1.0" />
   <title>HTML Web Page</title>
 </head>
 <body>
   <h1>Hello, World!</h1>
   <div id="todos"></div>

   <script>
     fetch("http://localhost:5000/")
       .then((response) => response.json())
       .then((data) => {
         const todos = document.getElementById("todos");

         data?.forEach((item, index) => {
           const todo = document.createElement("h2");
           todo.textContent = item._id + ". " + item.todo;
           todos.appendChild(todo);
         });
       })
       .catch((error) => {
         console.log("Error: ", error);
       });
   </script>
 </body>
</html>
```

Now you can check the result at http://localhost:3000/

frontend at http://localhost:3000/

Upto this point we've created our frontend container, backend container and called the backend API from the frontend. Next, Let's see how we can add a database container to this.

Adding a MySQL Database container

database container

First, Let's add the sql dump file that I've created for this tutorial. Create a folder named `db` in your main project directory and `dump.sql` file inside it. Then pase the below code to the `dump.sql` file and save it.

dump.sql

```
-- phpMyAdmin SQL Dump
-- version 5.2.1
-- https://www.phpmyadmin.net/
--
-- Host: database:3306
-- Generation Time: Jul 08, 2023 at 05:46 PM
-- Server version: 8.0.33
-- PHP Version: 8.1.17

SET SQL_MODE = "NO_AUTO_VALUE_ON_ZERO";
START TRANSACTION;
SET time_zone = "+00:00";


/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8mb4 */;

--
-- Database: `todo_app`
--

-- --------------------------------------------------------

--
-- Table structure for table `todos`
```

```
--

CREATE TABLE `todos` (
 `_id` int NOT NULL,
 `todo` varchar(255) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

--
-- Dumping data for table `todos`
--

INSERT INTO `todos` (`_id`, `todo`) VALUES
(1, 'I will wake up at 4 in the morining.'),
(2, 'I will practice docker for 1 hour.'),
(3, 'I will give time for 2 hours javascript.'),
(4, 'Then I will have breakfast.'),
(5, 'I will give time for 3 hours php.');

--
-- Indexes for dumped tables
--

--
-- Indexes for table `todos`
--
ALTER TABLE `todos`
 ADD PRIMARY KEY (`_id`);

--
-- AUTO_INCREMENT for dumped tables
--

--
-- AUTO_INCREMENT for table `todos`
--
ALTER TABLE `todos`
 MODIFY `_id` int NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=6;
COMMIT;
```

After above change, your folder structure should look like this.

```
.
■■■ api                     # Backend api files.
■   ■■■ index.php           # php page
■■■ db                      # SQL dump files.
■   ■■■ dump.sql            # sql dump file
■■■ frontend                # Frontend files. (.html/.js/.css)
■   ■■■ index.html          # html page
■■■ docker-compose.yml
■■■ Dockerfile
```

Let's append below code snippet to our `docker-compose.yml`. You should append it under `services` section.

```
  database:
   image: mysql:latest
   environment:
     MYSQL_DATABASE: todo_app
     MYSQL_USER: todo_admin
     MYSQL_PASSWORD: password
     MYSQL_ALLOW_EMPTY_PASSWORD: 1
   volumes:
     - "./db:/docker-entrypoint-initdb.d"
```

- **`database:`**: This is the name of the service. You can choose any name you prefer.
- **`image: mysql:latest`**: This line specifies the Docker image to be used for the database service. In this case, it uses the `mysql` image with the `latest` tag. The `mysql` image is the official image for MySQL.
- **`environment:`**: This section allows you to specify environment variables for the container.
- **`MYSQL_DATABASE: todo_app`**: This line sets the name of the MySQL database inside the container as "todo_app". You can change it to any desired database name.

- **`MYSQL_USER: todo_admin`**: This line specifies the username for the MySQL user inside the container as "todo_admin". You can modify it to any preferred username.
- **`MYSQL_PASSWORD: password`**: This line sets the password for the MySQL user as "password". You can choose a different password for your MySQL user.
- **`MYSQL_ALLOW_EMPTY_PASSWORD: 1`**: This line allows an empty password for the MySQL user. Setting it to `1` enables this behavior.
- **`volumes:`**: This section is used to mount the local filesystem directory to the container.
- **`- "./db:/docker-entrypoint-initdb.d"`**: This line maps the `./db` directory from the local filesystem to the `/docker-entrypoint-initdb.d` directory inside the container. It is typically used for initializing the database with SQL scripts or other initialization files.

Now rerun the terminal command `docker compose up`. Then open a new terminal and type `docker compose ps` in your project directory. If you've done everything correctly, you should be able to see all three containers running on your docker server.

PhpMyAdmin to Access the MySQL Database Container

Next, Let's add a browser interface to access your database. In this tutorial we use `phpmyadmin` to access our database. Let's add below code snippet to our `docker-compose.yml` file.

```
  phpmyadmin:
   image: phpmyadmin/phpmyadmin
   ports:
     - 8080:80
   environment:
     - PMA_HOST=database
     - PMA_PORT=3306
   depends_on:
     - database
```

Rerun the terminal command `docker compose up`

phpMyAdmin at http://localhost:8080

Now you can access the database container through http://localhost:8080/ by using credentials that we've given when creating our database container.

Wiring up the Backend API with Database container

To make it work you need to do some changes in your api folder.

1. Create a folder name `app` inside the api folder.
2. Add below 3 files inside that `app` folder.

config.php

```
<?php
define("DB_HOST", "database");
define("DB_USERNAME", "todo_admin");
define("DB_PASSWORD", "password");
define("DB_NAME", "todo_app");
```

Database.php

```
<?php
class Database
{
   protected $connection = null;

   public function __construct()
   {
       try {
           $this->connection = new mysqli(DB_HOST, DB_USERNAME, DB_PASSWORD, DB_NAME);
           if (mysqli_connect_errno()) {
               throw new Exception("Database connection failed!");
           }
       } catch (Exception $e) {
           throw new Exception($e->getMessage());
       }
   }

   private function executeStatement($query = "", $params = [])
   {
```

```
        try {
            $stmt = $this->connection->prepare($query);
            if ($stmt === false) {
                throw new Exception("Statement preparation failure: " . $query);
            }
            if ($params) {
                $stmt->bind_param($params[0], $params[1]);
            }
            $stmt->execute();
            return $stmt;
        } catch (Exception $e) {
            throw new Exception($e->getMessage());
        }
    }

    public function select($query = "", $params = [])
    {
        try {
            $stmt = $this->executeStatement($query, $params);
            $result = $stmt->get_result()->fetch_all(MYSQLI_ASSOC);
            $stmt->close();
            return $result;
        } catch (Exception $e) {
            throw new Exception($e->getMessage());
        }
        return false;
    }
}
```

todos.php

```
<?php
require_once "./app/Database.php";

class Todo extends Database
{
    public function getTodos($limit)
    {
        return $this->select("SELECT * FROM todos");
    }
}
```

To connect all together, let's update our `index.php` by replacing with the below code snippet.

index.php

```
<?php
// Set the response content type to JSON
header('Content-Type: application/json');
header('Access-Control-Allow-Origin: http://localhost:3000');
header('Access-Control-Allow-Methods: GET, POST, OPTIONS');

require "./app/config.php";
require_once "./app/todos.php";

$todoModel = new Todo();
$response = $todoModel->getTodos(10);
// Create a response array

// Encode the response array as JSON
$jsonResponse = json_encode($response);

// Output the JSON response
echo $jsonResponse;
```

Now, execute the command `docker compose up` in your terminal to see the changes. This will enable you to see the successful retrieval of todos from the backend to the frontend, as well as the backend's seamless interaction with the database for data retrieval.

Docker Compose Network

If you've followed the tutorial correctly, you should have a `docker-compose.yml` file like this.

```
version: "3"
services:
 frontend:
   image: httpd:latest
   volumes:
     - "./frontend:/usr/local/apache2/htdocs"
   ports:
     - 3000:80
 backend:
   container_name: simple-backend
   build:
     context: ./
     dockerfile: Dockerfile
   volumes:
     - "./api:/var/www/html/"
   ports:
     - 5000:80
 database:
   image: mysql:latest
   environment:
     MYSQL_DATABASE: todo_app
     MYSQL_USER: todo_admin
     MYSQL_PASSWORD: password
     MYSQL_ALLOW_EMPTY_PASSWORD: 1
   volumes:
     - "./db:/docker-entrypoint-initdb.d"
 phpmyadmin:
   image: phpmyadmin/phpmyadmin
   ports:
     - 8080:80
   environment:
     - PMA_HOST=database
     - PMA_PORT=3306
   depends_on:
     - database
```

Docker Compose understands the idea behind running services for one application on one network. When you deploy an app using Docker Compose file, even when there's no mention of specific networking parameters, Docker Compose will create a new bridge network and deploy the container over that network.

Let's discuss about docker networking in upcoming articles.

Summary

In this article, we covered several key concepts related to **Docker** and **Docker Compose**. We explored the fundamentals of Docker Compose and how it helps in managing multi-container applications. We discussed the role of **Docker Hub** as a centralized repository for **Docker images**. Additionally, we had a closer look into **Dockerfiles**, which allow us to define the configuration of a Docker image.

Furthermore, we examined how to add containers using Docker Compose, demonstrating how to define services within the `docker-compose.yml` file. We also explored how to establish connections between multiple containers in Docker Compose, enabling communication and interaction between them.

Lastly, we learned how to construct a simple 3-tier architecture using Docker Compose, incorporating frontend, backend API, and database containers. By leveraging the power of Docker Compose, we can easily orchestrate and manage the deployment of this architecture.

By gaining an understanding of these concepts and following the examples provided, you are now equipped to utilize Docker Compose effectively in building and deploying multi-container applications.

Resources

- **Docker Introduction Article:** https://medium.com/@kesaralive/from-os-hassles-to-team-harmony-how-docker-saved-our-group-project-e37c52ce8509
- **Docker Installation Guide:** https://docs.docker.com/engine/install/
- **Docker Compose :** https://docs.docker.com/compose/compose-file/compose-file-v3/
- **GitHub Repository:** https://github.com/kesaralive/docker-compose-101.git