Source code available:
https://github.com/francis-github-official/docker-app-php-mysql-phpMyAdmin-composer.git

Here's a text-based representation of the folder structure for your Dockerized PHP application, along with a detailed explanation of its workflow.

---

1. Folder Structure

```
php-docker-app/
├── index.php
├── composer.json
├── composer.lock  (Generated by 'composer install')
├── .env.example
├── .env          (Created by copying .env.example and filling in details)
├── Dockerfile
├── docker-compose.yml
├── database.sql
└── vendor/       (Generated by 'composer install' inside the Docker build or locally)
```

Explanation of Files/Folders:

- **`php-docker-app/`**: The root directory for your entire project.
- **`index.php`**: The main PHP application file. This is your web application's entry point, containing the logic for connecting to the database and displaying content.
- **`composer.json`**: Defines your PHP project's dependencies (e.g., `vlucas/phpdotenv`). Composer uses this file to know which packages to install.
- **`composer.lock`**: (Generated) This file is created by Composer after `composer install` is run. It locks the exact versions of all your dependencies, ensuring that everyone working on the project (and the Docker build process) uses the same dependency versions. **Crucial for reproducible builds.**
- **`.env.example`**: A template file showing the environment variables required by your application. It's used as a guide.
- **`.env`**: Contains the actual environment variables (e.g., database credentials) for your specific environment. This file is loaded by your PHP application using Dotenv and is passed to Docker Compose services. **It should not be committed to version control.**
- **`Dockerfile`**: Instructions for Docker on how to build the `app` (PHP application) Docker image. It specifies the base PHP image, installs system dependencies, PHP extensions, Composer, copies your application code, and installs PHP dependencies.
- **`docker-compose.yml`**: Defines and configures the multi-container Docker application. It orchestrates the `app` (PHP application), `mysql` (database), and `phpmyadmin`

services, defining their images, ports, volumes, environment variables, and dependencies.

- **`database.sql`**: An SQL script that is executed when the `mysql` container starts for the first time. It's used to create your database and initial tables (e.g., the `users` table).
- **`vendor/`**: (Generated) This directory contains all the PHP libraries and dependencies installed by Composer (e.g., the `phpdotenv` library).

---

2. Workflow

The workflow describes the steps to set up, run, and manage your Dockerized PHP application.

Phase 1: Initial Setup & Development (on your local machine)

1. **Create Project Structure:** You start by creating the `php-docker-app` directory and placing `index.php`, `composer.json`, `.env.example`, `Dockerfile`, `docker-compose.yml`, and `database.sql` within it.
2. **Generate `composer.lock`:**
   - Open your terminal and navigate into the `php-docker-app` directory.
   - Run `composer install` (or `docker run --rm -v $(pwd):/app composer install` if you don't have Composer locally).
   - This command reads `composer.json`, downloads `phpdotenv` into the `vendor/` directory, and generates the `composer.lock` file. This `composer.lock` file is essential for the Docker build process.
3. **Configure Environment Variables:**
   - Copy `.env.example` to `.env`: `cp .env.example .env`.
   - Verify or update the database credentials in `.env` to match those defined in `docker-compose.yml` for the `mysql` service.

Phase 2: Building and Running with Docker Compose

1. **Start Docker Engine:** Ensure Docker Desktop (or your Docker daemon) is running on your laptop.
2. **Build & Run Containers:**

From the `php-docker-app` directory in your terminal, execute:
 Bash

```
docker-compose up --build -d
```

   -
   - **`--build`**: This flag tells Docker Compose to build the `app` service's image (using the `Dockerfile`) before starting the containers.

- **`Dockerfile` Execution:**
    - Docker pulls the `php:8.2-apache` base image.
    - It installs necessary system packages (`git`, `unzip`, `libzip-dev`, etc.).
    - It installs PHP extensions (`pdo_mysql`, `gd`, `zip`, `mbstring`).
    - It copies the `composer` executable into the image.
    - **Crucially, it copies `composer.json` and `composer.lock` into the image.**
    - It runs `composer install --no-dev --optimize-autoloader --no-interaction` *inside the building image*. This creates the `vendor/` directory with all PHP dependencies within the image itself.
    - It then copies the rest of your application code (`index.php`, `.env.example`, etc.) into the image.
    - It sets file permissions.
- **`-d`**: This flag runs the containers in "detached" mode, meaning they run in the background, freeing up your terminal.

3. **Service Orchestration (managed by `docker-compose.yml`):**
    - **`mysql` Service Startup:** Docker Compose starts the `mysql` container.
        - It uses the `mysql:8.0` image.
        - It sets environment variables for root password, database name, user, and user password.
        - It mounts the `db_data` volume for data persistence.
        - It runs the `database.sql` script to initialize the database and create the `users` table.
        - **Health Check:** Docker Compose continuously runs the `healthcheck` defined for `mysql` (`mysqladmin ping`). It waits until MySQL is fully ready to accept connections.
    - **`phpmyadmin` Service Startup:** Once `mysql` is reported as `healthy`, Docker Compose starts the `phpmyadmin` container.
        - It uses the `phpmyadmin/phpmyadmin` image.
        - It maps host port `8081` to container port `80`.
        - It configures phpMyAdmin to connect to the `mysql` service (`PMA_HOST: mysql`).
    - **`app` Service Startup:** Once `mysql` is reported as `healthy`, Docker Compose starts the `app` (PHP application) container.
        - It uses the image built in step 2.
        - It maps host port `8080` to container port `80`.
        - **Volume Mounts:**

        - `.:/var/www/html`: This mounts your host's `php-docker-app` directory into the container's `/var/www/html`. This allows for live code changes to `index.php` (and other application files) without rebuilding the image.
        - `/var/www/html/vendor`: This anonymous volume ensures that the `vendor/` directory *generated inside the Docker image during the build* is preserved and used by the running container, preventing it from being overwritten by a potentially empty `vendor/` directory from the host.
    - **Environment Variables:** The `.env` file is passed to the `app` service, making `DB_HOST`, `DB_DATABASE`, etc., available to your `index.php` application.

Phase 3: Accessing the Applications

1. **PHP Application:** Open your web browser and navigate to `http://localhost:8080`. You should see the PHP application connecting to the database and displaying user data.
2. **phpMyAdmin:** Open another tab in your web browser and navigate to `http://localhost:8081`. You can log in using the credentials defined in your `docker-compose.yml` (e.g., `root` and `root_password` or `user` and `password`).

Phase 4: Stopping and Cleaning Up

Stop Containers: To stop all running services defined in `docker-compose.yml`:

Bash

```
docker-compose down
```

1.

Stop and Remove Volumes (Clean Up Data): To stop services and remove the volumes (including your MySQL database data), which is useful for a clean start:

Bash

```
docker-compose down -v
```

2.

This comprehensive workflow ensures that your PHP application, MySQL database, and phpMyAdmin are set up, run, and managed efficiently using Docker Compose.

In case there is an issue with composer or not installed on development machine:

Run Composer install on your host: If you don't have Composer installed globally on your host, you can use a temporary Docker container to run it:

Bash

```
docker run --rm -v $(pwd):/app composer install
```

If you have Composer installed globally, simply run:

Bash

```
composer install
```

Bring down your existing containers (if running) and then bring them up again with the rebuild flag:

Bash

```
docker-compose down
docker-compose up --build -d
```

—---

Beginners FAQ’s:

Are there size limits for Docker named volumes? Are Docker named volumes stored directly on the host filesystem? How does Docker manage storage for named volumes? Keyword search: Docker named volume storage location Keyword search: Docker volume size limits Keyword search: Docker named volume default size No, Docker named volumes like `db_data` do **not** have a fixed size limit imposed by Docker itself, beyond the available disk space of the host machine.

Here's a breakdown of how Docker named volumes work in terms of size:

1. **Storage Location:** Named volumes are managed by Docker and are typically stored in a dedicated directory on the Docker host's filesystem.
   - On Linux, this is usually `/var/lib/docker/volumes/`
   - On Docker Desktop (Windows/macOS), they are stored within the underlying virtual machine's disk image (e.g., `Docker.qcow2` on macOS, or a VHDX file on Windows).
2. **Size determined by Host Filesystem:** Since a named volume is essentially a directory on your host machine's filesystem (or within the Docker Desktop VM's disk image), its maximum size is effectively limited by the available disk space of the underlying filesystem where Docker stores its volumes. If you have 100GB free on your hard drive, your volume can theoretically grow up to 100GB.

3. **Difference from Container Writable Layer:** It's important to distinguish volumes from the writable layer of a container. A container's writable layer *can* have a default size limit (often around 10GB for some storage drivers or older Docker Desktop versions). However, volumes are designed to persist data *outside* of this container layer and do not share that specific container-layer size limit.

4. **No Direct Size Configuration in `docker-compose.yml` for `local` driver:** For the default `local` volume driver (which `db_data` uses unless specified otherwise), you generally cannot define a specific size limit directly in your `docker-compose.yml` or `docker volume create` command. The `size` option is typically available for specific volume drivers or for `tmpfs` (temporary in-memory filesystems), which are not suitable for persistent database storage.

In summary:

Your `db_data` volume will consume as much disk space as it needs, up to the total free space available on the disk partition where your Docker daemon stores its volumes. For a MySQL database, this is usually sufficient for most development and even many production scenarios unless you're dealing with truly massive datasets.

If you ever encounter "no space left on device" errors, the problem is likely that the host machine's disk where Docker volumes are stored has run out of space, not an inherent size limit on the Docker volume itself.