Install Docker in Ubuntu 22.04

WHAT'S DOCKER & HOW DOES IT WORK:

Docker is a platform that uses OS-level virtualization to deliver software in packages called containers. These containers are isolated from each other and bundle their own software, libraries, and system tools, allowing you to run your applications consistently across different computing environments.

Docker Images: Docker images are lightweight, standalone, executable packages that include everything needed to run a piece of software. This includes the application's code, a runtime, libraries, environment variables, and configuration files. Docker images are read-only and immutable, meaning they do not change once created. They serve as the blueprint for creating Docker containers. Docker images can be personalized using a Dockerfile, which specifies the base image to use and defines the steps to create a new image. Docker Hub serves as a public registry where you can find a wide variety of images created by the community.

Docker Containers: A Docker container is a runtime instance of a Docker image. It provides an isolated environment where applications can run. Containers are lightweight and portable, encapsulating everything an application needs to run. This ensures consistent behavior, regardless of the environment in which they are run. While containers themselves are ephemeral, their state can be preserved by committing changes to a new Docker image.

Volumes: Docker volumes are the preferred mechanism for persisting data generated by and used by Docker containers. Unlike the container's file system, volumes are designed to persist data across container lifecycle, allowing data to survive even after a container is deleted. Volumes can be shared and reused among containers, and are stored in a part of the host filesystem managed by Docker (/var/lib/docker/volumes/ on Linux). They provide a way to backup data, share data among containers, and even with the host system, making them an essential tool for data persistence and sharing in Docker.

FRESH INSTALLATION:

- **Update your existing list of packages:**
  sudo apt Update.
- **Uninstall any old versions of Docker:**
  sudo apt remove docker docker-engine docker.io containerd runc
- **Install a few prerequisite packages which let apt use packages over HTTPS:**
  sudo apt install apt-transport-https ca-certificates curl software-properties-common
- **Add the GPG key for the official Docker repository to your system:**
  curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add –
- **Add the Docker repository to APT sources:**
  sudo add-apt-repository "deb [arch=amd64]
  https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
- **Update the package database with the Docker packages from the newly added repo:**
  sudo apt Update
- **Finally, install Docker:**
  sudo apt install docker-ce
- **Docker should now be installed, the daemon started, and the process enabled to start on boot. Check that it's running:**
  sudo systemctl status docker
- **Remember that if you want to avoid typing sudo whenever you run the docker command, add your username to the docker group:**
  sudo usermod -aG docker ${USER}

- **To check Docker version:**
  docker --version

You'll need to log out and log back in for this to take effect.

START DOCKER SERVER & SOME BASIC COMMANDS:

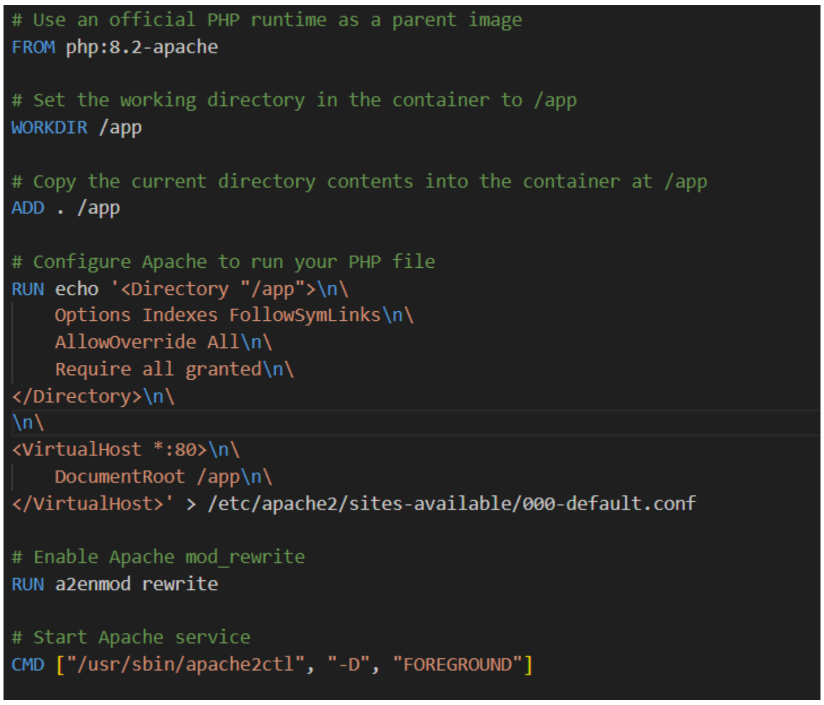
- **“sudo service docker start”** → Start the Docker service.
- **“sudo service docker status”** → Check the status of the Docker service.
- **“sudo service docker stop”** → Stop the Docker service.
- **“docker version”** → To see Docker version and also if it’s connected to server.
- **“docker images”** → See the information of the different images we’ve created.
- **“docker ps”** → List of the containers that are running and see information.
- **“docker ps -a”** → List of all the containers (running or not) and see information.
- **“docker volume ls”** → To see the volumes of type “automatic” that we’ve created.
- **“docker start -ID-”** → Start the container with the provided ID or container name.
- **“docker stop -ID-”** → Stop the container with the provided ID or container name.
- **“docker container prune”** → Remove all the stopped containers.
- **“docker system prune”** → Remove all the things that are pending are not needed anymore.
- **“docker rm -ID-”** → To remove one or more Docker containers providing ID or name of the containers. Adding **“-f”** at the end will force the Image to stop if it’s running.
- **“docker rmi -ID-”:** To remove one or more Docker images giving ID or name. Adding **“-f”** at the end will force the Image to stop if it’s running.
- **“docker tag source_image:tag target_image:tag”:** To assign a tag to a Docker image, it’s also used to change the full name and tag of the image.
- **“docker push” I “docker push”:** Need more information, but is to push the image to the docker hub or pull from the hub to your system.

You don't need to move your files into the Docker container manually. Docker can do this for you when it builds the image. You specify this in the Dockerfile with the ADD or COPY instruction. In the Dockerfile, the line ADD . /app copies the contents of the current directory (where the Dockerfile is located) into the /app directory in the Docker image.

The Dockerfile should be named Dockerfile with no file extension. It's case-sensitive, so make sure the 'D' is capitalized.

CREATING A SIMPLE DOCKERFILE:

- **Here's an example of a simple Dockerfile that you can use as a starting point, you can use another php-apache version for a more updated image:**



```
# Use an official PHP runtime as a parent image
FROM php:8.2-apache

# Set the working directory in the container to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
ADD . /app

# Configure Apache to run your PHP file
RUN echo '<Directory "/app">\n\
    Options Indexes FollowSymLinks\n\
    AllowOverride All\n\
    Require all granted\n\
</Directory>\n\
\n\
<VirtualHost *:80>\n\
    DocumentRoot /app\n\
</VirtualHost>' > /etc/apache2/sites-available/000-default.conf

# Enable Apache mod_rewrite
RUN a2enmod rewrite

# Start Apache service
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

This Dockerfile does the following:

- Starts from an official PHP image with Apache included.
- Sets the working directory in the Docker image to be /app.
- Copies your current directory (i.e., your PHP project) into the Docker image.
- Exposes port 80 of the Docker container to the outside world.
- Configures Apache to serve your PHP application.
- Enables Apache's mod_rewrite module, which is often used for URL rewriting.
- Starts the Apache service when the Docker container is run.

- **You can build this Docker image with the following command:**
  docker build -t **my-php-app** . (-t to tag the name of the build)
- **And then run it with:**
  docker run -p 8080:80 **my-php-app** (-p to publish into port)
  docker run -d -p 8080:80 **my-php-app** (add -d and will run it detached to the terminal)

If you make changes to your application code, you'll need to rebuild your Docker image so that the changes are included in the new containers that you start from the image. A way to avoid rebuilding the image, it to use **Docker volumes** to share files between your host system and your Docker container. This allows your container to "see" changes to your files as soon as you save them, without having to rebuild the Docker image.

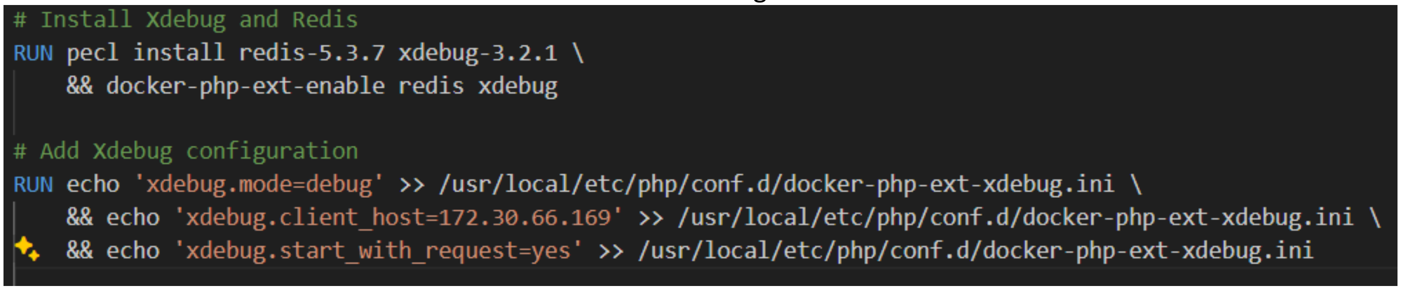
- **Here's how you can modify your `docker run` command to use a Docker volume:**
  docker run -d -p 8080:80 -v $(pwd):/app **my-php-app**

Command `-v $(pwd):/app` tells Docker to create a volume that maps the current directory (`$(pwd)`) on your host system to the `/app` directory in the Docker container. This volume is a bind volume, that is different than the ones that are named. They don't appear in the volume listing.

Note: This assumes that your Dockerfile sets `/app` as the working directory and that's where your application code lives in the Docker container. If your application code is in a different directory in the Docker container, you should adjust the `/app` part of the `-v` option to match that directory.

X-DEBUG INSTALL IN DOCKER:

- First add this line for Docker to install X-Debug:

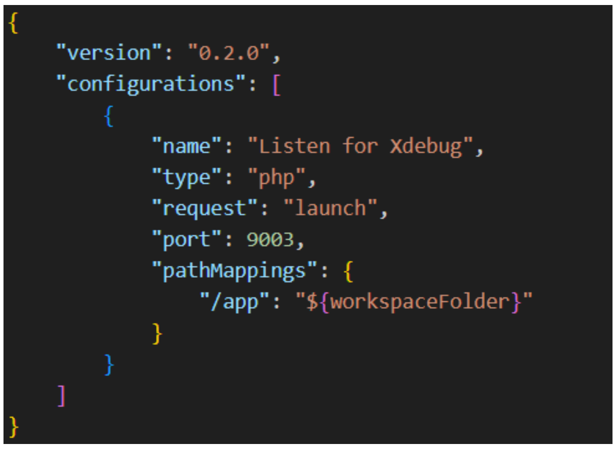


```
# Install Xdebug and Redis
RUN pecl install redis-5.3.7 xdebug-3.2.1 \
    && docker-php-ext-enable redis xdebug

# Add Xdebug configuration
RUN echo 'xdebug.mode=debug' >> /usr/local/etc/php/conf.d/docker-php-ext-xdebug.ini \
    && echo 'xdebug.client_host=172.30.66.169' >> /usr/local/etc/php/conf.d/docker-php-ext-xdebug.ini \
    && echo 'xdebug.start_with_request=yes' >> /usr/local/etc/php/conf.d/docker-php-ext-xdebug.ini
```

This Dockerfile installs Xdebug and configures it for remote debugging. The xdebug.client_host=host.docker.internal line tells Xdebug to connect to the host machine, which is where your IDE runs. **You need to ALWAYS UPDATE the client_host with the IP from your computer (ip addr show) and create a new image for it to work. Other option is to use Docker Compose (see below).**

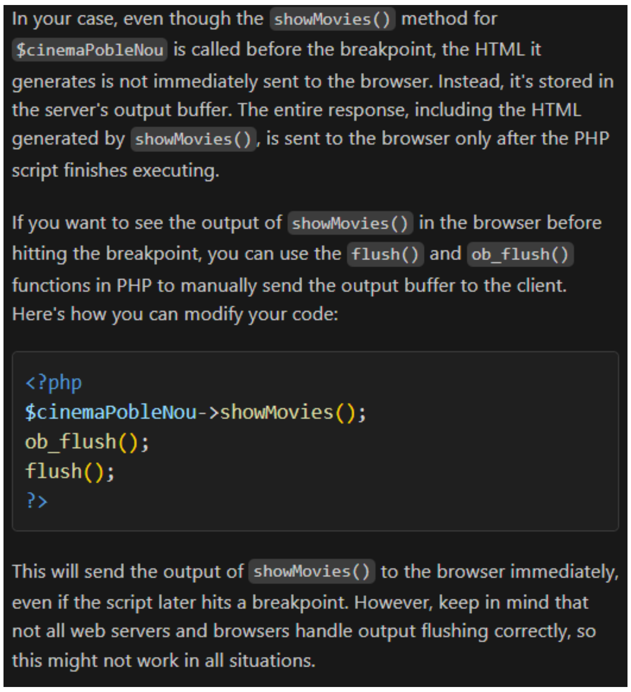
- You need to install the PHP Debug extension in VSC.
- After this, go to the play with the bug icon in VSC, create a launch.json file if there is no file created and add the following:



```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Listen for Xdebug",
            "type": "php",
            "request": "launch",
            "port": 9003,
            "pathMappings": {
                "/app": "${workspaceFolder}"
            }
        }
    ]
}
```

- Start PHP container, set breakpoints and then run the X-Debug.

TO TAKE INTO ACCOUNT WHEN USING XDEBUG



In your case, even though the `showMovies()` method for `$cinemaPobleNou` is called before the breakpoint, the HTML it generates is not immediately sent to the browser. Instead, it's stored in the server's output buffer. The entire response, including the HTML generated by `showMovies()`, is sent to the browser only after the PHP script finishes executing.

If you want to see the output of `showMovies()` in the browser before hitting the breakpoint, you can use the `flush()` and `ob_flush()` functions in PHP to manually send the output buffer to the client. Here's how you can modify your code:

```php
<?php
$cinemaPobleNou->showMovies();
ob_flush();
flush();
?>
```

This will send the output of `showMovies()` to the browser immediately, even if the script later hits a breakpoint. However, keep in mind that not all web servers and browsers handle output flushing correctly, so this might not work in all situations.

<u>COMPOSER INSTALL IN DOCKER:</u>

- First add this line for Docker to install Composer:

```
# Copy Composer from official Composer image (latest version)
COPY --from=composer:latest /usr/bin/composer /usr/bin/composer
```

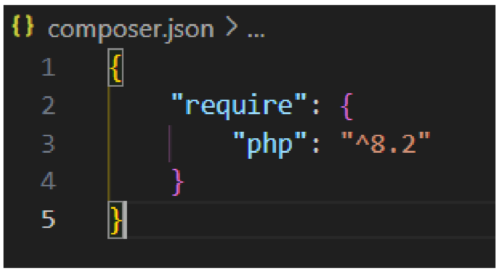
- And also this:

```
# Copy composer.json and Install PHP dependencies
COPY composer.json ./
RUN composer install
```

- As composer downloads packages and dependencies, you need to add in your Dockerfile also git, unzip:

```
# Install git and unzip
RUN apt-get update && apt-get install -y git unzip
```

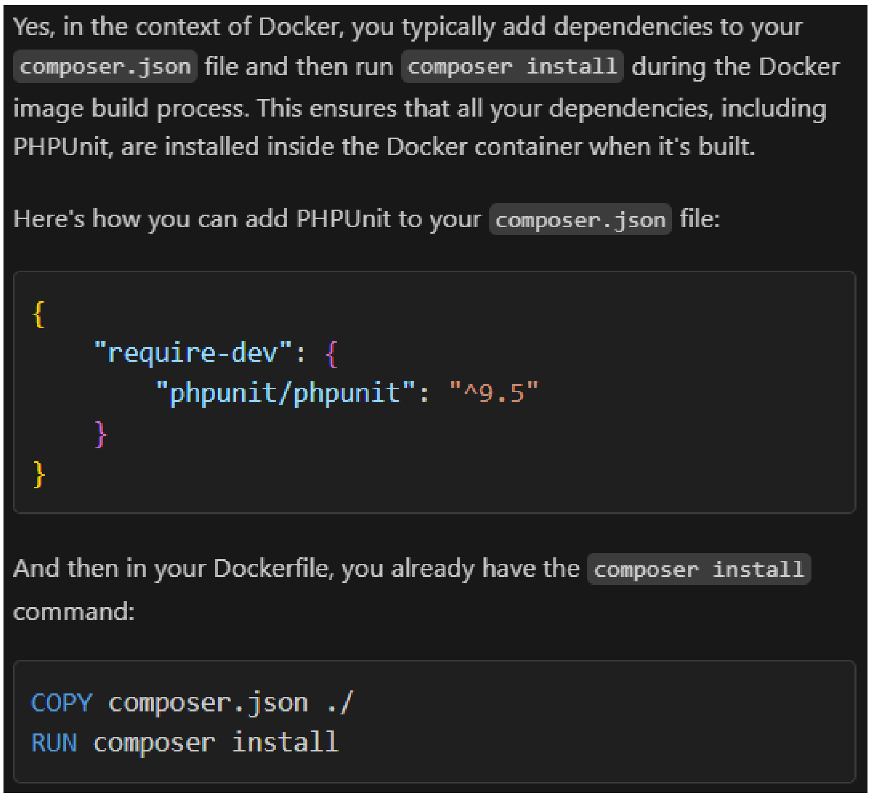
- Finally create a composer.json file in you projects root, or same directory as Dockerfile. You should add all the dependencies needed in that json, minimum specify the PHP version:



```json
{
    "require": {
        "php": "^8.2"
    }
}
```

- Also, you need to create a src folder in the root, same place as Dockerfile, mainly if you need PHP Unit or any other dependency that uses that folder.
- Execute a new build for the new Dockerfile.

TO TAKE INTO ACCOUNT WHEN USING COMPOSER



Yes, in the context of Docker, you typically add dependencies to your `composer.json` file and then run `composer install` during the Docker image build process. This ensures that all your dependencies, including PHPUnit, are installed inside the Docker container when it's built.

Here's how you can add PHPUnit to your `composer.json` file:

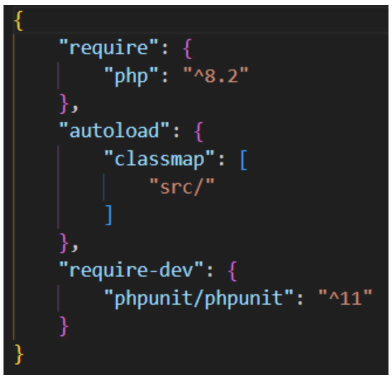
```
{
    "require-dev": {
        "phpunit/phpunit": "^9.5"
    }
}
```

And then in your Dockerfile, you already have the `composer install` command:

```
COPY composer.json ./
RUN composer install
```

INSTALL PHPUNIT USING COMPOSER IN DOCKER:

- Add the dependencies to your composer.json:



```
{
    "require": {
        "php": "^8.2"
    },
    "autoload": {
        "classmap": [
            "src/"
        ]
    },
    "require-dev": {
        "phpunit/phpunit": "^11"
    }
}
```

- Make sure you have created src folder in the root, same place Dockerfile is.
- Build the container.
- To run a test with PHPUnit, you have to run docker with this command:
  docker run --rm **my-php-app ./vendor/bin/phpunit --bootstrap vendor/autoload.php tests**

This command will start a Docker container from your `my-php-app` image, then run the PHPUnit tests inside that container. The `--rm` option tells Docker to automatically remove the container when it exits.

The `./vendor/bin/phpunit --bootstrap vendor/autoload.php tests` part is the command to run PHPUnit. It assumes that your tests are located in a directory named `tests` in the root of your project. If your tests are located in a different directory, you should adjust the command accordingly.

DOCKER DEV VSC EXTENSION

It’s an extension that allows you to open the Docker container remotely, as a new VSC window, so you can access all the folders an run the commands in the terminal as if you had all the programs, frameworks, etc. installed.

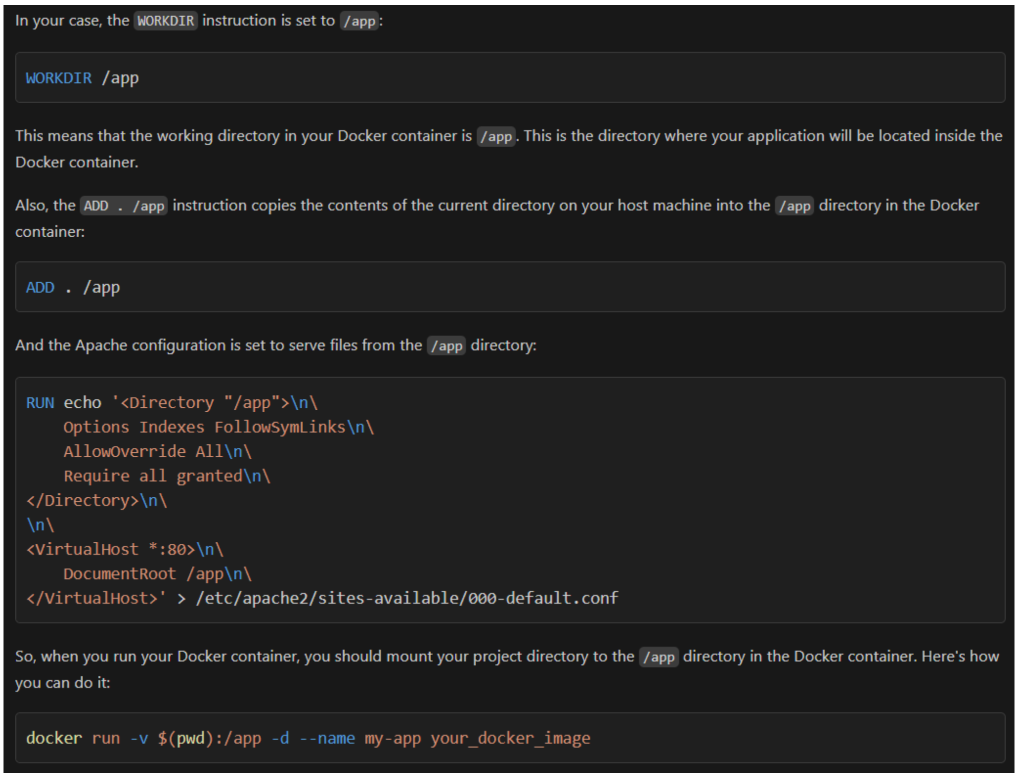
- First download the extension.
- Then start docker and run your container (my-php-app is the build, my-app is the name you’re giving to the container) -d for detached:
  docker run -d --name **my-app **my-php-app****

IMPORTANT: If you want the container to open but at the same time update your files in your local project, you need to run docker in this way:

docker run -v $(pwd):/app -d --name **my-app **your_docker_image****

-v: This option mounts a volume, which is a directory from your host machine, into the Docker container. Any changes made to the files in this directory will be reflected in both the container and your host machine.

This is because:



In your case, the `WORKDIR` instruction is set to `/app`:

```
WORKDIR /app
```

This means that the working directory in your Docker container is `/app`. This is the directory where your application will be located inside the Docker container.

Also, the `ADD . /app` instruction copies the contents of the current directory on your host machine into the `/app` directory in the Docker container:

```
ADD . /app
```

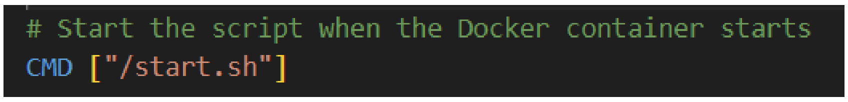
And the Apache configuration is set to serve files from the `/app` directory:

```
RUN echo '<Directory "/app">\n\
    Options Indexes FollowSymLinks\n\
    AllowOverride All\n\
    Require all granted\n\
</Directory>\n\
\n\
<VirtualHost *:80>\n\
    DocumentRoot /app\n\
</VirtualHost>' > /etc/apache2/sites-available/000-default.conf
```

So, when you run your Docker container, you should mount your project directory to the `/app` directory in the Docker container. Here's how you can do it:

```
docker run -v $(pwd):/app -d --name my-app your_docker_image
```

- It’s important to change in the Dockerfile the line of how and when the script should be run because as we’re copying our files into container’s app folder it may overwrite dependencies and other things (such as vendor folder). So change the CMD line into:



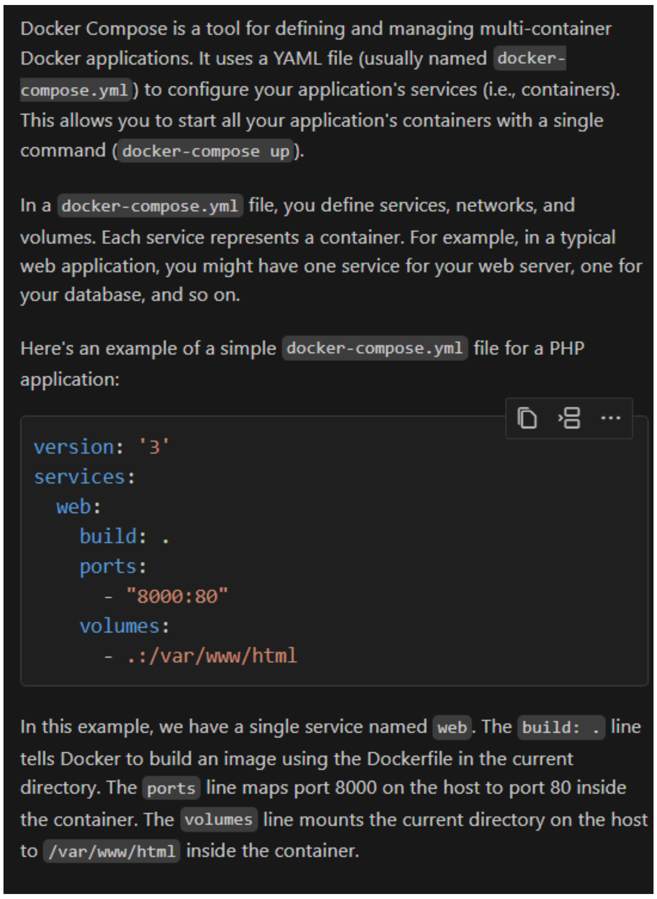
```
# Start the script when the Docker container starts
CMD ["/start.sh"]
```

- Now that your Docker container is running, you can use the Dev Containers extension in Visual Studio Code to attach to the running container and open your project inside it. Here's how:
  1. In Visual Studio Code, click on the green icon at the bottom-left corner of the status bar (or use the command palette `Ctrl+Shift+P` and search for "Remote-Containers: Attach to Running Container...").
  2. A list of running Docker containers will appear. Select the container you just started.
  3. Visual Studio Code will reload and open your project inside the Docker container. Now, your IDE is running in the same environment where your

Docker container is running. This means it has access to PHPUnit and should be able to recognize the PHPUnit classes and methods.
4. Now, you can run your PHPUnit tests directly from Visual Studio Code. If you have the PHPUnit extension installed, you should be able to run your tests by clicking on the "Run Test" link above your test methods.

- You need to install the extensions for VSC in your new Docker remote system.

OJO A ESTO PARA MAS ADELANTE:



Docker Compose is a tool for defining and managing multi-container Docker applications. It uses a YAML file (usually named `docker-compose.yml`) to configure your application's services (i.e., containers). This allows you to start all your application's containers with a single command (`docker-compose up`).

In a `docker-compose.yml` file, you define services, networks, and volumes. Each service represents a container. For example, in a typical web application, you might have one service for your web server, one for your database, and so on.

Here's an example of a simple `docker-compose.yml` file for a PHP application:

```
version: '3'
services:
  web:
    build: .
    ports:
      - "8000:80"
    volumes:
      - .:/var/www/html
```

In this example, we have a single service named `web`. The `build: .` line tells Docker to build an image using the Dockerfile in the current directory. The `ports` line maps port 8000 on the host to port 80 inside the container. The `volumes` line mounts the current directory on the host to `/var/www/html` inside the container.

DOCKER COMPOSE VS DOCKER NETWORK:

Docker Compose and Docker Network serve different purposes and are often used together, not as alternatives to each other.

Docker Compose is a tool for defining and running multi-container Docker applications. It uses a YAML file to configure your application's services and creates and starts all the services from your configuration.

Docker Network, on the other hand, is a feature of Docker that allows you to define and manage networks for connecting Docker containers. It's a part of Docker itself, not a separate tool like Docker Compose.

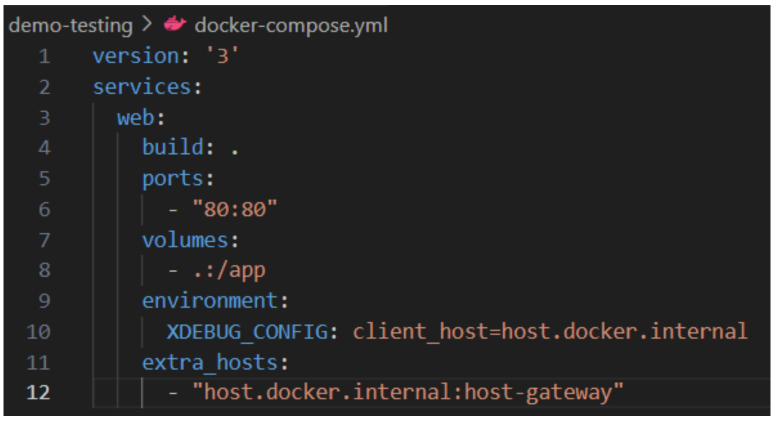
When you use Docker Compose, it automatically creates a default network for your application and connects all the services (containers) in your Compose file to this network. However, Docker Compose also allows you to define custom networks in the Compose file if you need more control over the networking.

So, it's not a matter of choosing Docker Network instead of Docker Compose. Rather, you would use Docker Network as part of your Docker setup to manage networking between containers, and you could use Docker Compose to simplify the process of defining, running, and managing multi-container applications, which includes defining networks.

In other words, Docker Compose is a higher-level tool that uses Docker Network (among other Docker features) under the hood. If you're manually running individual containers with docker run and want them to communicate, you'd use Docker Network to create and manage the networks. If you're defining a multi-container application with a Compose file, Docker Compose handles the networking for you, but you can still use Docker Network features to customize the networking if needed.

USING DOCKER COMPOSE TO CONFIGURE XDEBUG IP:

- First check the documentation and install it, if you already have Docker Compose installed you can check the version:
  docker-compose --version
- You need to create a **docker-compose.yml** file in you root directory, same as Dockerfile, this is a simple configuration for XDebug:



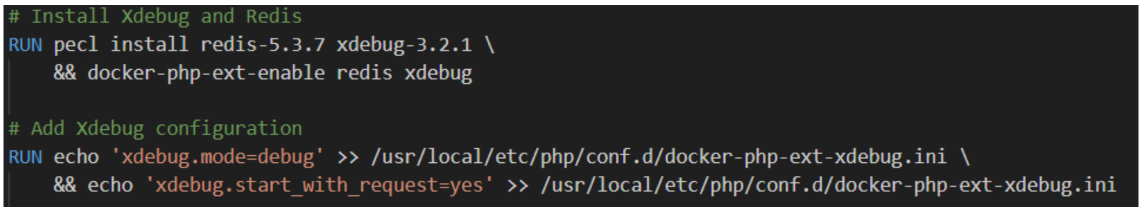
demo-testing > docker-compose.yml

```yaml
version: '3'
services:
  web:
    build: .
    ports:
      - "80:80"
    volumes:
      - .:/app
    environment:
      XDEBUG_CONFIG: client_host=host.docker.internal
    extra_hosts:
      - "host.docker.internal:host-gateway"
```

Here's what each part does:
build: . tells Docker Compose to build the Docker image using the Dockerfile in the current directory.
ports: - "80:80" maps port 80 in the container to port 80 on your host machine. This means that you can access the web server in the container at http://localhost:80.

volumes: - .:/app mounts the current directory (on your host machine) to the /app directory in the container. This is useful for development, as it allows you to edit your code on your host machine and have the changes immediately reflected in the container.
environment: sets environment variables in the container. Here, we're setting the XDEBUG_CONFIG variable to client_host=host.docker.internal. This tells Xdebug to connect back to your host machine.
extra_hosts: adds entries to the container's /etc/hosts file. Here, we're adding an entry for host.docker.internal, which resolves to the IP address of the Docker host. This is a workaround for the lack of host.docker.internal support on Linux.

- In your Dockerfile you should remove the line of the host of XDebug and should look like this:



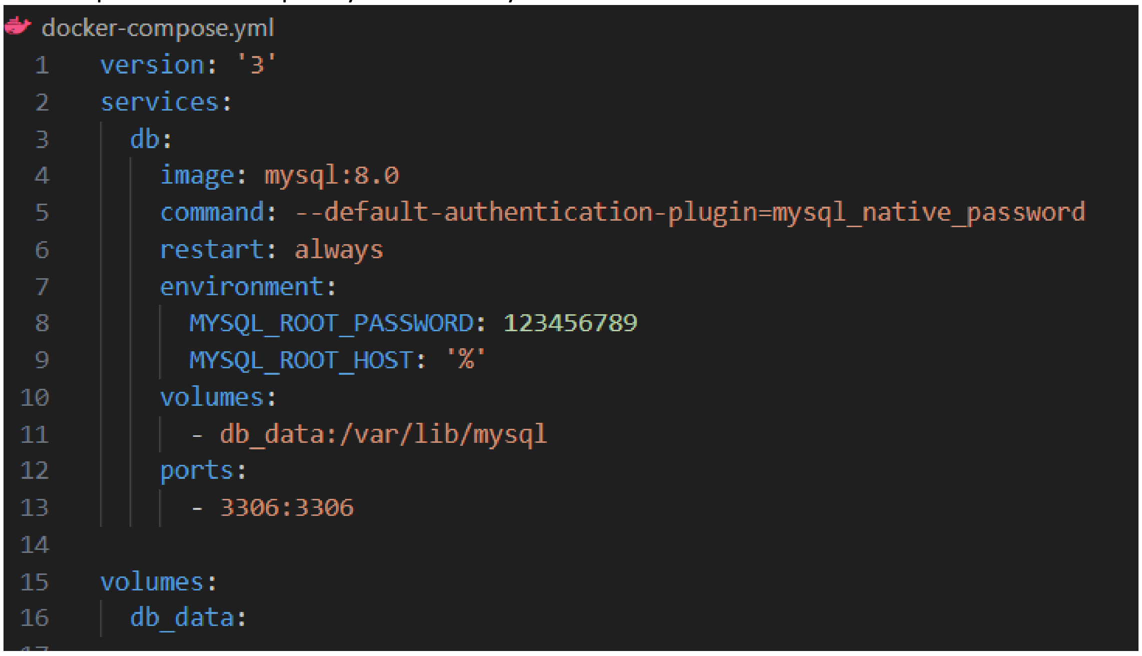
```
# Install Xdebug and Redis
RUN pecl install redis-5.3.7 xdebug-3.2.1 \
    && docker-php-ext-enable redis xdebug

# Add Xdebug configuration
RUN echo 'xdebug.mode=debug' >> /usr/local/etc/php/conf.d/docker-php-ext-xdebug.ini \
    && echo 'xdebug.start_with_request=yes' >> /usr/local/etc/php/conf.d/docker-php-ext-xdebug.ini
```

- **To start your application, you can run docker-compose up in the same directory as this file. Docker Compose will build your image (if necessary), create your container, and start your application.**
- **To stop your application, you can run docker-compose down. This will stop and remove your container.**

<u>**DOCKER COMPOSE FOR MY SQL:**</u>

- Simple docker-compose.yml file for MySQL connection:



docker-compose.yml

```
version: '3'
services:
  db:
    image: mysql:8.0
    command: --default-authentication-plugin=mysql_native_password
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: 123456789
      MYSQL_ROOT_HOST: '%'
    volumes:
      - db_data:/var/lib/mysql
    ports:
      - 3306:3306

volumes:
  db_data:
```

- After creating the file, **run docker compose up -d** and then connect through Workbench to the container.

<u>**OTHER DOCKER COMMANDS:**</u>

- **“docker inspect” + “...”** → Per mirar tota la informació d’un contenidor determinat. S’ha de posar el ID del contenidor del qual estem intentant treure la informació. Va bé per mirar la ruta d’un volum creat de forma automàtica.
- **“docker commit”** → Si hem realitzat canvas al cotnainer i els volem guardar a la imatge, podem fer un commit.
- **“docker exec” + “...”** → Per executar un command X que li diem en el contenidor en el que estem actualment. Ex: “docker exec -it “ID...” /bin/bash”. En aquest cas, -it per indicar el modo, el ID del contenidor i “/bin/bash” el command que hem escollit.

- **“cd /var/www/html”** → Per anar o crear el root del bin/bash en el contenidor de “orboan/docker-httpd”.