

# Containers

## Introdução Engenharia Informática

Mário Antunes

October 13, 2025

## Exercises

### Practical Lab: Working with Docker Compose

**Objective:** This lab will guide you through the fundamentals of creating, managing, and deploying applications using Docker (with focus on Compose files). You will apply the concepts of images, containers, volumes, and networking to build and run single and multi-service applications.

**Prerequisites:**

- A computer with a modern web browser and a text editor.
  - Docker and Docker Compose installed.
- 

### Installing Docker on Debian

If you are using a Linux host, follow these steps in your terminal to install the latest version of Docker. Based on these [instructions](#).

1. **Set up Docker's apt repository:**

```
# Remove non-official docker packages
sudo apt remove docker.io docker-doc \
docker-compose podman-docker containerd runc

# Update package index and install prerequisites
sudo apt update
sudo apt install ca-certificates curl

# Add Docker's official GPG key
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/debian/gpg \
-o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] \
https://download.docker.com/linux/debian \
$(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt update
```

2. **Install the Docker packages:**

```
sudo apt install docker-ce docker-ce-cli containerd.io \
docker-buildx-plugin docker-compose-plugin
```

3. **Manage Docker as a non-root user (Recommended):** To run docker commands without sudo, add your user to the docker group.

```
sudo usermod -aG docker $USER
```

**Important:** You must log out and log back in for this change to take effect.

---

## Exercise 1: “Hello, World” with Docker Compose

**Goal:** Understand the basic structure of a compose .yml file and run a pre-built image.

1. Create a new folder for this exercise (e.g., ex1-helloworld).
  2. Inside the folder, create a new file named compose .yml with the following content:

```
services:
  hello:
    image: hello-world
```
  3. Open your terminal in this folder and run the application.

```
$ docker compose up
```
  4. Observe the output. The hello-world container will start, print its message, and then exit.
  5. Clean up the created container.

```
$ docker compose down
```
- 

## Exercise 2: Building a Custom Web Server Image

**Goal:** Use a Dockerfile with Docker Compose to create a self-contained application image.

1. Create a new folder (ex2-build) and a subfolder inside it named my-website.
  2. Inside my-website, create a file named index.html:

```
<!DOCTYPE html>
<html>
<body>
  <h1>This page was built into the Docker image!</h1>
</body>
</html>
```
  3. In the root of the ex2-build folder, create a Dockerfile:

```
FROM nginx:alpine
COPY ./my-website /usr/share/nginx/html
```
  4. Finally, create your compose .yml file:

```
services:
  webserver:
    build: .
    ports:
      - "8080:80"
```
  5. Build and start the service. The -d flag runs it in the background.

```
$ docker compose up --build -d
```
  6. Open your browser to <http://localhost:8080>. You should see your custom webpage.
- 

## Exercise 3: Live Development with Volumes

**Goal:** Understand how volumes allow you to change your website’s content without rebuilding the image.

1. Create a new folder (ex3-volumes) with the same my-website/index.html structure as the previous exercise.
2. Create a compose.yml file. This time, we will use the standard nginx:alpine image and mount our local folder as a volume. **No Dockerfile is needed.**

```
services:
  webserver:
    image: nginx:alpine
    ports:
      - "8080:80"
    volumes:
      - ./my-website:/usr/share/nginx/html
```

3. Start the service: `docker compose up -d`.
4. Open your browser to `http://localhost:8080` to confirm it's working.
5. **Live Update:** While the container is running, **edit the index.html file** on your host machine. Change the heading to `<h1>Live update with a Volume!</h1>`.
6. Save the file and **refresh your browser**. The change appears instantly!

## Exercise 4: Caching Rich Content with Varnish & NGINX ⚡

**Goal:** Build a two-tier web application with a Varnish cache serving a rich webpage from an NGINX backend.

1. **Create the File Structure:**
  - Create a new folder (e.g., ex4-varnish-cache).
  - Inside, create two subfolders: varnish and my-dynamic-website.
2. **Create the Web Content:**
  - Find a fun animated GIF online and save it inside my-dynamic-website as animation.gif.
  - Inside my-dynamic-website, create an index.html file to display the GIF:
 

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Varnish Cache Test</title>
  <style> body { font-family: sans-serif; text-align: center; } </style>
</head>
<body>
  <h1>This page is being cached by Varnish!</h1>
  
</body>
</html>
```
3. **Create the Varnish Configuration:**
  - Inside the varnish folder, create a file named default.vcl. This tells Varnish where to find the NGINX server.
 

```
vcl 4.1;
backend default {
  .host = "nginx";
  .port = "80";
}
```
4. **Create the Compose File:**
  - In the root of your ex4-varnish-cache folder, create the compose.yml:
 

```
services:
  cache:
    image: varnish:stable
    volumes:
      - ./varnish:/etc/varnish
    ports:
```

```

- "8080:80"
depends_on:
- nginx

nginx:
  image: nginx:alpine
  volumes:
    - ./my-dynamic-website:/usr/share/nginx/html

```

#### 5. Run and Verify:

- Start the services: `docker compose up -d`.
- Open your browser to `http://localhost:8080`. You should see your webpage with the GIF. The key here is that **Varnish** served you the page, not NGINX directly.
- **See the cache in action:** Check the NGINX logs for the first request.  
`$ docker compose logs nginx`
- Now, refresh your browser page several times. Check the nginx logs again. You should see **no new log entries**, because Varnish is serving the content from its cache without contacting the NGINX backend.

## Exercise 5: Deploying a Real-World Application

**Goal:** Learn to read official documentation and deploy a complex, self-hosted service of your choice.

1. **Choose a Service:** Go to [LinuxServer.io](https://linuxserver.io) and browse their list of popular images. Choose one that interests you, for example:
  - **Jellyfin:** A media server for your movies and music.
  - **Nextcloud:** A personal cloud for files, contacts, and calendars.
  - **Home Assistant:** An open-source home automation platform.
2. **Read the Documentation:** On the page for your chosen image, find the “Docker Compose” section. Read it carefully, paying close attention to the required **volumes** and **environment variables**.
  - **Volumes** (`- ./config:/config`): This is where the application’s configuration will be stored on your host.
  - **Environment Variables** (PUID, PGID, TZ): These are critical. TZ sets your timezone (e.g., Europe/Lisbon). PUID and PGID ensure that files created by the container have the correct ownership. On Linux/macOS, find your ID by running the `id` command in your terminal. A common value is `1000`.
3. **Create Your compose.yml:** Based on the documentation, create the file. Here is an example for Jellyfin:

```

services:
  jellyfin:
    image: lscr.io/linuxserver/jellyfin:latest
    container_name: jellyfin
    environment:
      - PUID=1000
      - PGID=1000
      - TZ=Europe/Lisbon
    volumes:
      - ./config:/config
      - ./tvshows:/data/tvshows
      - ./movies:/data/movies
    ports:
      - "8096:8096"
    restart: unless-stopped

```

#### 4. Prepare and Deploy:

- Create the local folders you defined in your volumes (e.g., `mkdir config tvshows movies`).
- Run the application: `docker compose up -d`.

5. **Explore:** Check the documentation for the default port number. For Jellyfin, it's 8096. Open your browser to `http://localhost:8096` and follow the setup wizard for your new service!