Example docker-compose.yml

It defines the format of the Compose file, by ensuring compatibility with specific Docker Compose features and syntax.

Services: The services section lists each containerised service required for the application. Each service can have configuration options, such as which image to use, environment variables, and resource limits.

Networks: In the network section, you can define custom networks that enable communication between containers. Additionally, it allows you to specify network drivers and custom settings for organizing container interactions.

Volumes: Volumes allow for data persistence across container restarts and can be shared between containers if needed. They enable you to store data outside the container's lifecycle, making it useful for shared storage or preserving application state

Here's a sample Compose file that defines two services, a shared network, and a volume

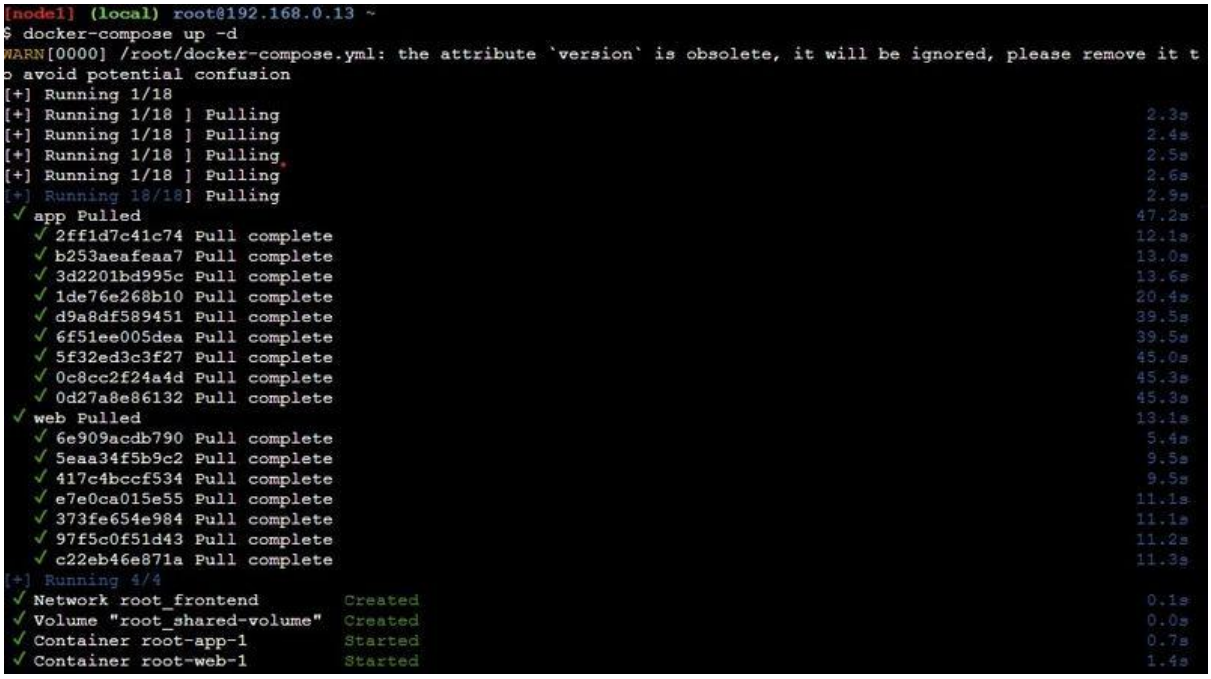
version: '3.8'

services:

```
  web:
    image: nginx:latest
    ports:
      - "80:80"
    networks:
      - frontend
    volumes:
      - shared-volume:/usr/share/nginx/html
    depends_on:
      - app

  app:
    image: node:14
    working_dir: /app
    command: node server.js   # Specify a command
    networks:
      - frontend
    volumes:
      - shared-volume:/app/data
networks:
```

```
  frontend:
    driver: bridge
volumes:
  shared-volume:  # Remove incorrect syntax
```



Docker Compose Tool To Run aMulti Container Applications

The article talks about how to run multi-container applications using a single command. Docker Compose is a tool for defining and running multi-container Docker applications. With Compose, you can configure a file (YAML file) to configure your docker containers. Then Once you configured the Yaml file with a single

command, you create and start all the services (containers) from your configuration.

Step 1: Create a directory gfg_docker_compose that holds our project

$ mkdir gfg_docker_compose

Step 2: Move to that directory making as your working directory by running the following command.

$ cd gfg_docker_compose

Step 3: Create the requirements.txt file

$ touch requirements.txt

Step 4: Copy the below-provided code in that requirements.txt file.

Flask

redis

Step 5: Create the file app.py. It will used to have the code for our Flask app

$ touch app.py

Step 6: Create the Dockerfile named file.

$ touch Dockerfile

Networking between containers

Docker Network

Docker networking allows containers to communicate with each other, the host system, and external networks. Docker creates and manages networks that containers can use, enabling seamless communication across different containers.

Bridge: The default network for containers. If no network is specified when creating a container, Docker places it in the bridge network. The default network type for containers on standalone hosts. Containers on the same bridge network can communicate with each other via IP addresses, and you can also map container ports to the host system.

Host: Bypasses Docker's virtualized network stack and allows the container to use the host's network directly. This network mode allows a container to share the host system's network stack. No isolation between the container and the host network.

None: Containers are isolated with no network interface, perfect for processes that don't need network connectivity.

Inspect od these network using this commend:

docker network ls

Running a Container with a Specific Network

docker run --name first_container --network bridge -p 8080:80 -d nginx

```
mehmetturgutgezgin@192 ~ % docker run --name first_container --network bridge -p
 8080:80 -d nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
92c3b3500be6: Pull complete
ee57511b3c68: Pull complete
33791ce134bf: Pull complete
cc4f24efc205: Pull complete
3cad04a21c99: Pull complete
486c5264d3ad: Pull complete
b3fd15a82525: Pull complete
Digest: sha256:04ba374043ccd2fc5c593885c0eacddebabd5ca375f9323666f28dfd5a9710e3
Status: Downloaded newer image for nginx:latest
4417ee2dbdedf6f0a8fe6d4ca749fab2134ed8f7daf6b7670a146d98ab307faa
```

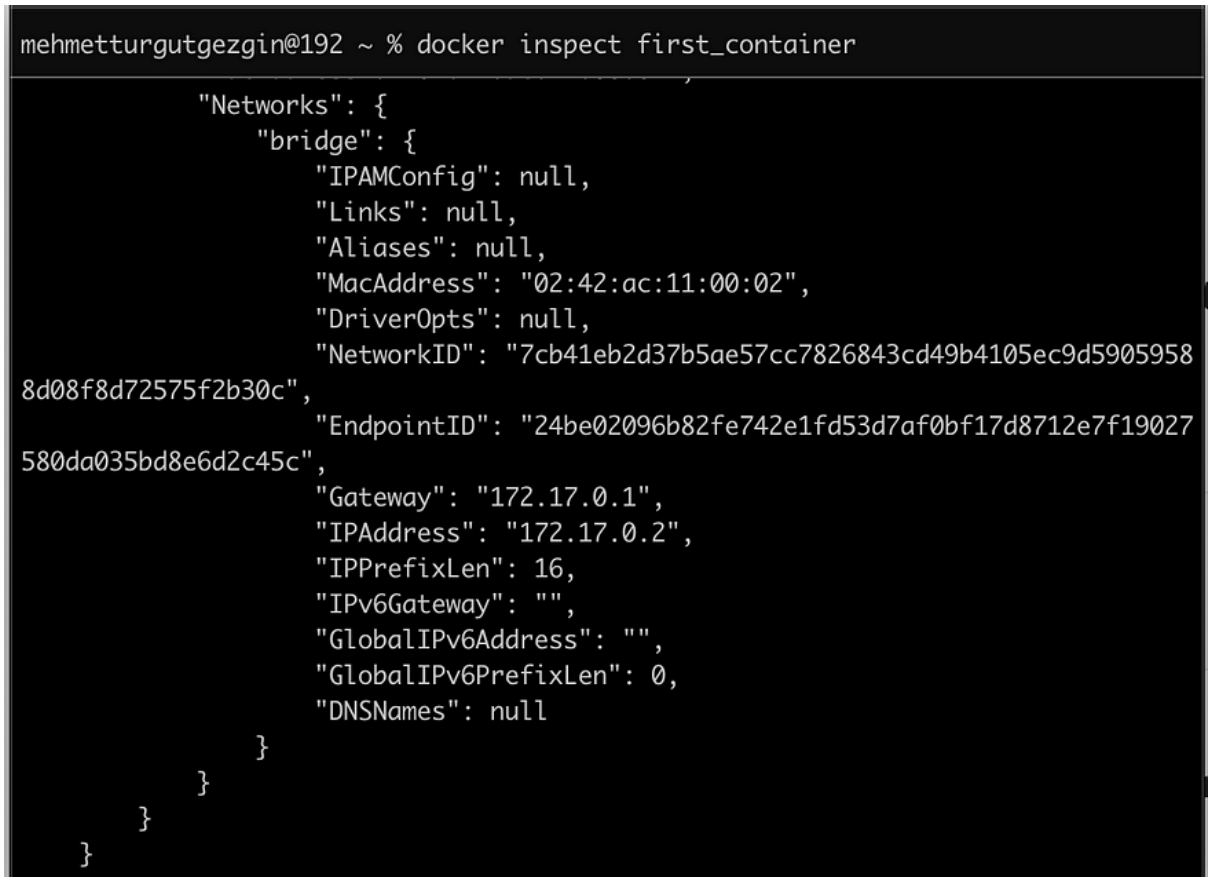
```
mehmetturgutgezgin@192 ~ % curl http://localhost:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Inspecting the Network and Container

docker inspect first_container



Create another container and communicate with each other

create another container (second_container) that can communicate with first_container on the same bridge network. We will use an Alpine Linux container with curl installed for this purpose.

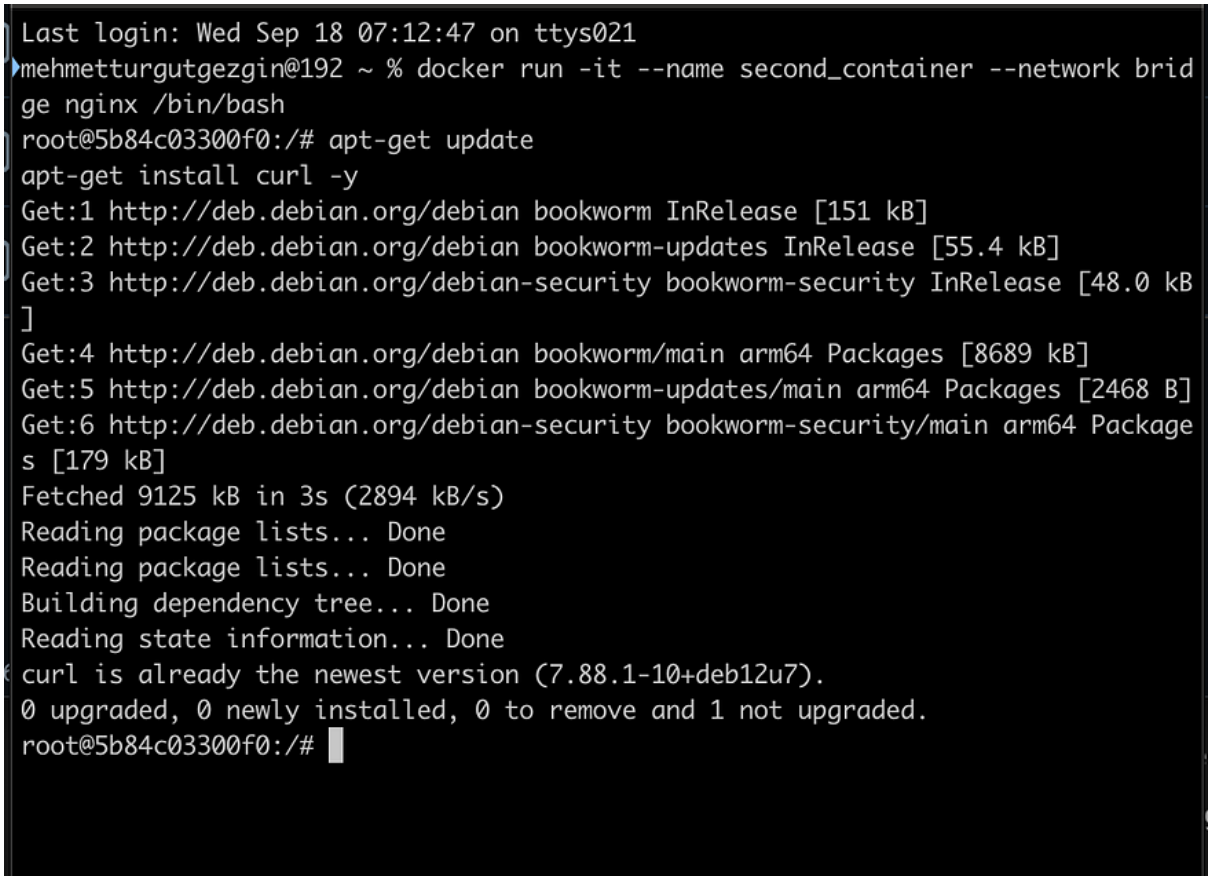
docker run -it --name second_container --network bridge nginx /bin/bash

Creates and runs the second_container with an interactive bash shell (/bin/bash).

The --network bridge flag ensures it is connected to the same default bridge network as first_container.

apt-get update

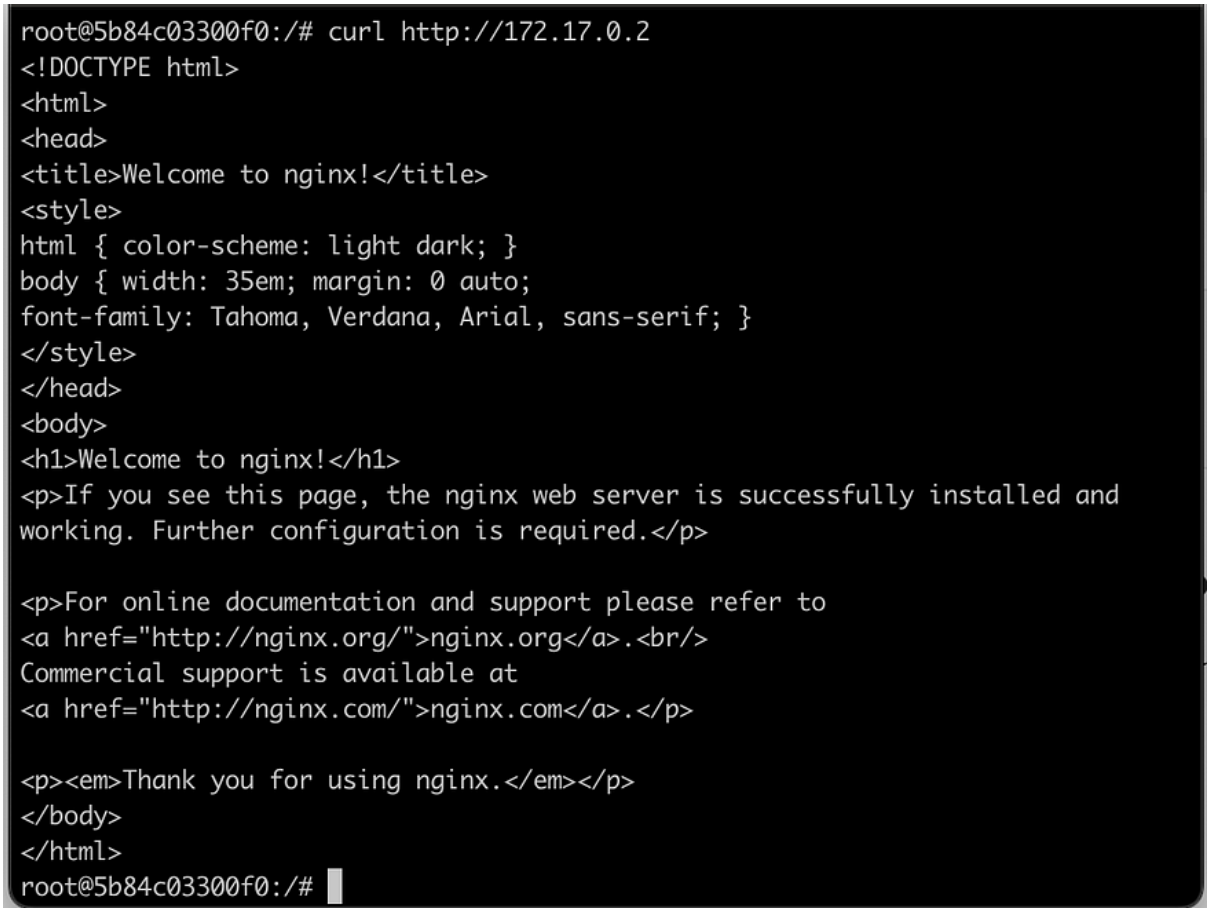
apt-get install curl -y



Now that both containers are connected to the same bridge network, we will curl from second container to the Nginx server running in first container.

In both cases, you should get the default Nginx welcome page as a response, indicating that the

second_container was able to communicate with first_container successfully:



To confirm both containers are connected to the bridge network, use the following command:

docker network inspect bridge

Volume & environment variable management

Introduction to Docker Volumes

Docker containers are ephemeral by nature. This means any data created inside a container is lost once that container is removed. Volumes are the solution. They're designed to persist data outside of the container's lifecycle, enabling us to store and manage data that must survive when a container is removed or the image is rebuilt.

Two types of Mounts in Docker:

Named Volumes: Created and managed by Docker. Data is stored in a part of the host file system which is managed by Docker (/var/lib/docker/volumes on Linux). This is the most common way of handling persistent data and the way to go in most production scenarios.

Bind Mounts: Ties a volume to a specific folder or file on the host machine. It provides the most flexibility but can cause discrepancies between host and container environments due to configuration or permission issues.

What Are Environment Variables

Environment variables are key-value pairs used to configure applications. Instead of hard-coding values (such as passwords or database URLs), you can store them as variables and access them from your code.

Why Use Environment Variables in Docker Compose?

When you're using Docker Compose to spin up multi-container applications, environment variables help you:

Avoid hardcoding sensitive data in your docker-compose.yml

Easily switch between development, staging, and production setups

Keep your configuration clean and reusable

Ways to Set Environment Variables in Docker

1. Using the environment attribute

You can set variables inline in your docker-**compose.yml file**

2. Using the env_file attribute to load variables

3. Using shell environment variables

4. Passing variables inline when running a container (less common in Compose)

docker run -e NODE_ENV=production node:23.11.0