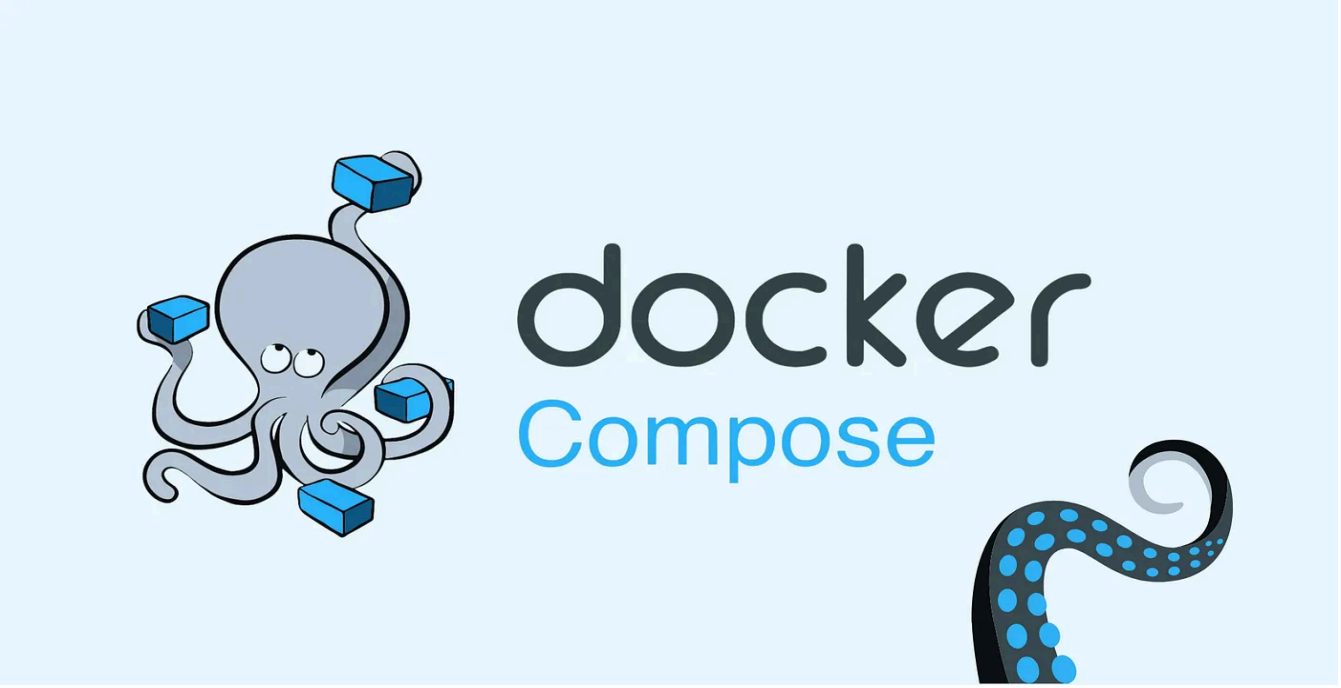
Building a Simple 3-Tier Architecture with Docker Compose



Rediet Teshome

March ,2025

1.Introduction

In this project, we are setting up a 3-tier application using Docker for the Frontend, Backend, and Database.

This solution leverages Docker for containerization, ensuring consistency across environments, and is built around the following technologies:

- Frontend: React (Node.js)
- Backend: Node.js with Express.js
- Database: MongoDB
- Container Orchestration: Docker Compose

-

2. Prerequisites

Before starting, ensure that you have the following installed on your machine:

A. Docker: Docker is required to create and manage containers.

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.

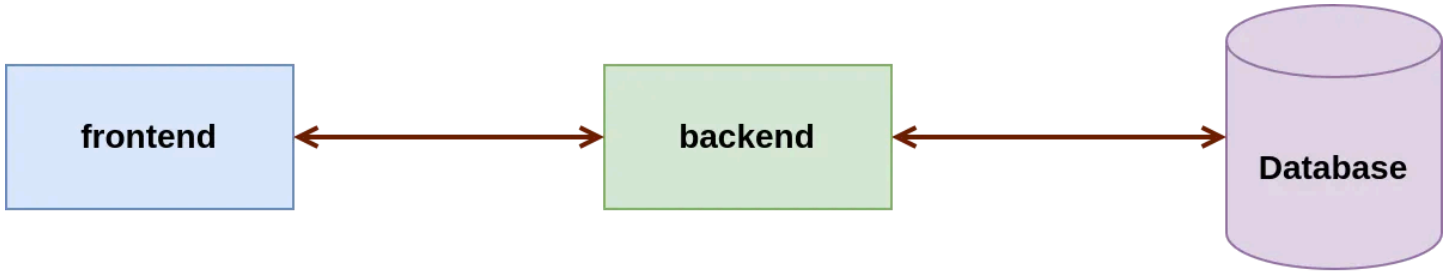
```
$ docker version
$ docker compose version
```

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.

B. Git: Ensure that Git is installed for version control and cloning the repository.

```
$ git --version
```



A simple 3-tier architecture

3. Docker Configuration

This section explains the Dockerfile setup for the Frontend, Backend, and MongoDB, as well as how they are containerized.

Frontend Dockerfile

```
dockerfile

FROM node:22 AS build

WORKDIR /app

COPY package*.json ./
RUN npm install

COPY . .
RUN npm run build

EXPOSE 3000
CMD ["npm", "run", "dev"]
```

- **FROM node:22:** Uses the official Node.js 22 image as the base for building the frontend.
- **WORKDIR /app:** Sets the working directory inside the container.
- ***COPY package*.json ./**:** Copies the `package.json` files, which are necessary for installing dependencies.
- **RUN npm install:** Installs the dependencies specified in `package.json`.
- **COPY . .**: Copies the remaining files of the project into the container.
- **RUN npm run build**: Builds the frontend for production.
- **EXPOSE 3000:** Exposes port 3000 for the frontend.
- **CMD ["npm", "run", "dev"]:** Runs the frontend application in development mode.

Backend Dockerfile:

```
FROM node:22

WORKDIR /app

COPY package*.json ./
RUN npm install

COPY . .

EXPOSE 5000
CMD ["npm", "start"]
```

- **FROM node:22:** Uses Node.js 22 as the base image.
- **WORKDIR /app:** Sets the working directory to /app inside the container.
- ***COPY package.json ./**:** Copies the package.json files to the container.
- **RUN npm install:** Installs backend dependencies.
- **COPY . .:** Copies the backend code into the container.
- **EXPOSE 5000**: Exposes port 5000 for the backend.
- **CMD ["npm", "start"]:** Starts the backend server using npm start.

Database Dockerfile:

```
FROM mongo:latest

EXPOSE 27017
```

- **FROM mongo:latest**: Uses the latest MongoDB official image as the base.
- **EXPOSE 27017**: Exposes MongoDB's default port (27017).

docker-compose.yml

```yaml
services:
  frontend:
    build:
      context: ./Frontend
    ports:
      - "3000:3000"
    networks:
      - app-network
    depends_on:
      - backend

  backend:
    build:
      context: ./Backend
    ports:
      - "5000:5000"
    environment:
      - MONGO_URI=mongodb://mongo:27017
    networks:
      - app-network
    depends_on:
      - mongo
    restart: unless-stopped
  mongo:
    image: mongo:latest
    ports:
      - "27017:27017"
    networks:
      - app-network
    restart: unless-stopped
networks:
  app-network:
```

The docker-compose.yml file defines the services for Frontend, Backend, and MongoDB.

- **frontend**: Builds the frontend container from the `Frontend` directory, exposing port 3000.
- **backend**: Builds the backend from the `Backend` directory, exposing port 5000. It uses the environment variable `MONGO_URI` to connect to MongoDB.
- **mongo**: Uses the official MongoDB image, exposing port 27017.
- **networks**: Defines an internal network (`app-network`) that ensures communication between containers.

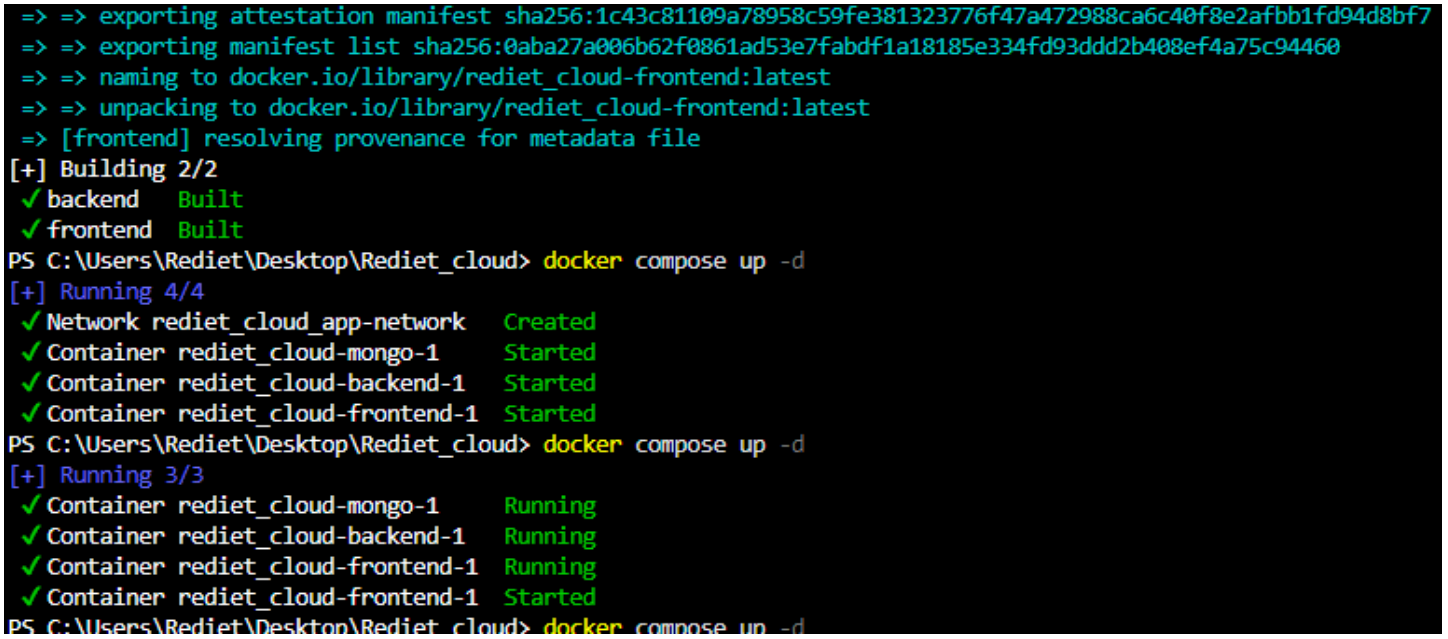
4. Building and Running the Application

Clone the Repository: Clone the repository to your local machine

```
$ git clone
https://github.com/Redieteshome/cloud-engineering-pathway-assessment.git
$ cd cloud-engineering-pathway-assessment
```

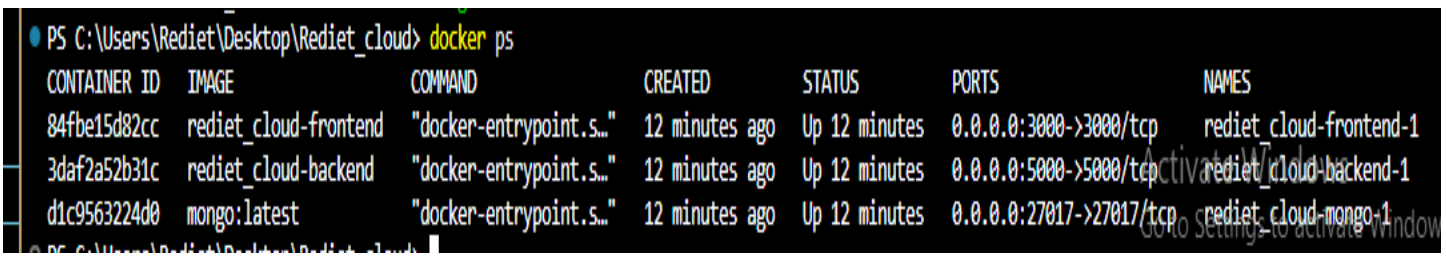
Build and Run Containers: Build and run the containers using Docker Compose

```
$ docker-compose up --build
```



Once all containers are up and running, you can run the below code:

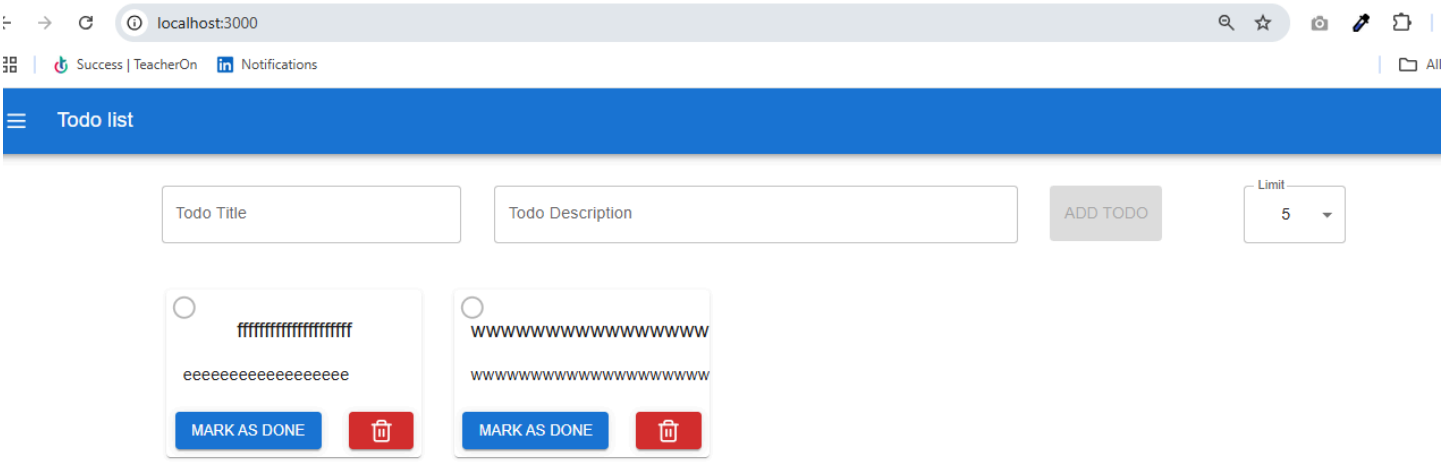
```
$Docker ps
```



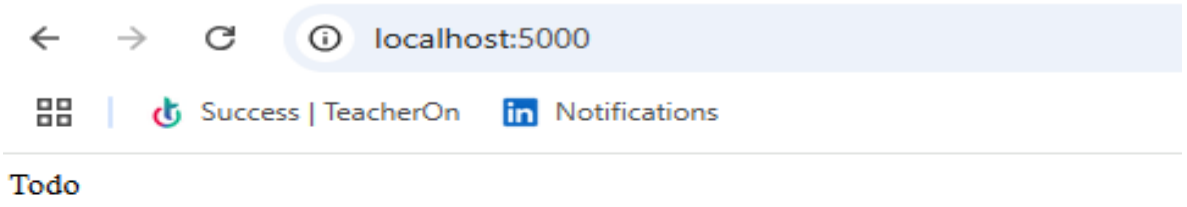
You can access

- **Frontend: http://localhost:3000**
- **Backend API: http://localhost:5000**
- **Database: http://localhost:5000/api/todos**

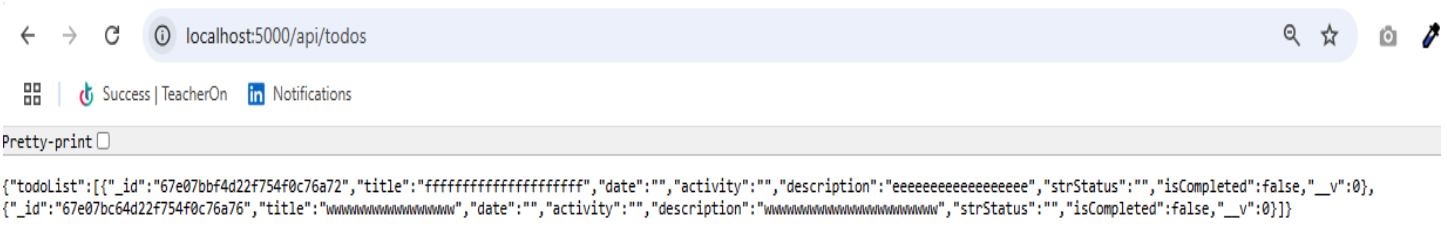
Frontend from browser



Backend from browser

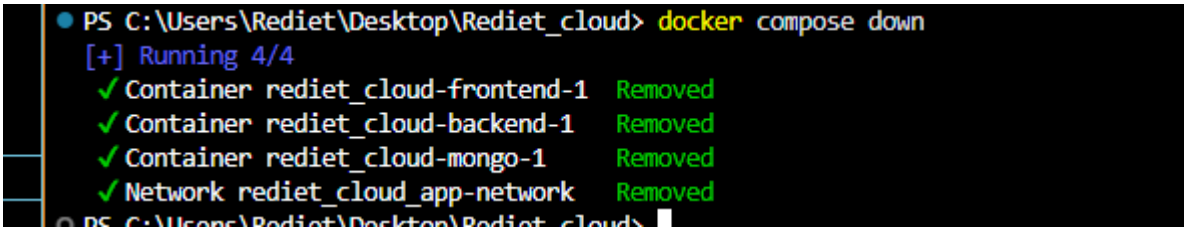


Database from browser



Stopping Containers: To stop and remove containers , run the below command

```
$ docker-compose down
```



It will Successfully remove the created files.

5. Network and Security Configurations

A. Network Configuration: All containers are connected to a custom bridge network (`app-network`), which allows them to communicate. The `depends_on` keyword ensures that the containers are started in the right order.

B. Port Exposure:

- **Frontend**: Port 3000
- **Backend**: Port 5000
- **MongoDB**: Port 27017

C.Security:

- **Environment Variables**: The backend uses the `MONGO_URI` to securely connect to MongoDB.

6. Troubleshooting Guide

Containers are not starting:

- Try rebuilding the containers with:

```
$ docker-compose up --build
```

Backend API not connecting to MongoDB:

- Ensure that MongoDB is running and check the logs with:

```
$ docker-compose logs backend
```

Port Conflicts:

- Ensure that the ports (3000, 5000, 27017) are not already in use on your host machine.

7 . Testing

The `test.sh` script is designed to verify that all components of your containerized 3-tier application (Frontend, Backend, and Database) are up and running correctly. It performs the following tasks:

1. **Test the Backend API**
   The backend API should be accessible via `http://localhost:5000`. The script performs an HTTP GET request to check if the backend service is responding.
   If the backend is running, it will return a successful response.

2. **Test MongoDB Connection**
   The script checks if MongoDB is running and can be accessed via `mongodb://mongo:27017`. The backend uses this URI to connect to the database. By checking the connection, we can ensure that the backend can communicate with MongoDB.

3. **Test the Frontend**
   The script checks if the frontend service is running at `http://localhost:3000`. If the frontend is running, it will respond with a

successful message.

How to Run the Test Script

To run the `test.sh` script, follow these steps:

Ensure that the containers are up and running.
If you haven't already done so, build and start the containers with:

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

1. **Make the `test.sh` script executable.**

Run the following command to make the script executable:

```
chmod +x test.sh
```

2. **Run the `test.sh` script.**

Now, you can execute the script to test the services:

```
./test.sh
```

3. **Expected Output of `test.sh`**

If all components are running correctly, the script should print the following output:

```
Backend is UP!
MongoDB is UP!
Frontend is UP!
```

6. Conclusion

This setup demonstrates how to containerize a 3-tier application using Docker. The system uses React for the frontend, Node.js with Express for the backend, and MongoDB for data storage.

For future improvements, consider deploying the system using Kubernetes for better scalability and management.