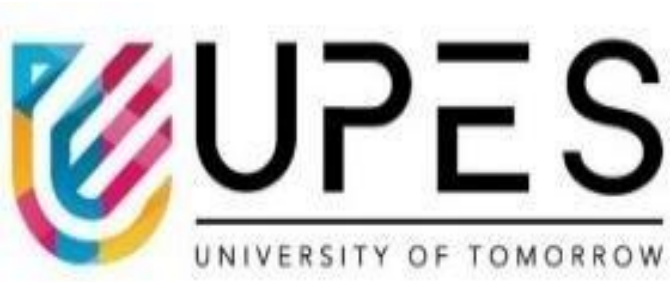
Containerization Project Report

Dockerized Web Application using Node.js, Express and PostgreSQL



Name: Vanshika Munjal

SAP-ID: 500121784

Batch: B2 CCVT

Course: Containerization and Devops

Containerization Project Report

Dockerized Web Application using Node.js, Express and PostgreSQL

1. Introduction

Containerization is a lightweight virtualization technique that allows applications and their dependencies to be packaged together and deployed consistently across different environments. Containers ensure portability, scalability, and efficient resource utilization.

In this project, a multi-container application was developed using Docker and Docker Compose. The system consists of:

- A Node.js Express backend API
- A PostgreSQL database
- Docker volumes for persistent storage
- Docker networking for container communication

The backend container communicates with the database container through Docker networking, allowing the application to perform database operations such as inserting and retrieving student records.

2. System Architecture

The project uses a two-container architecture where each service runs in its own isolated container.

Components:

Backend Container

- Built using Node.js and Express
- Provides REST API endpoints
- Handles HTTP requests and database communication

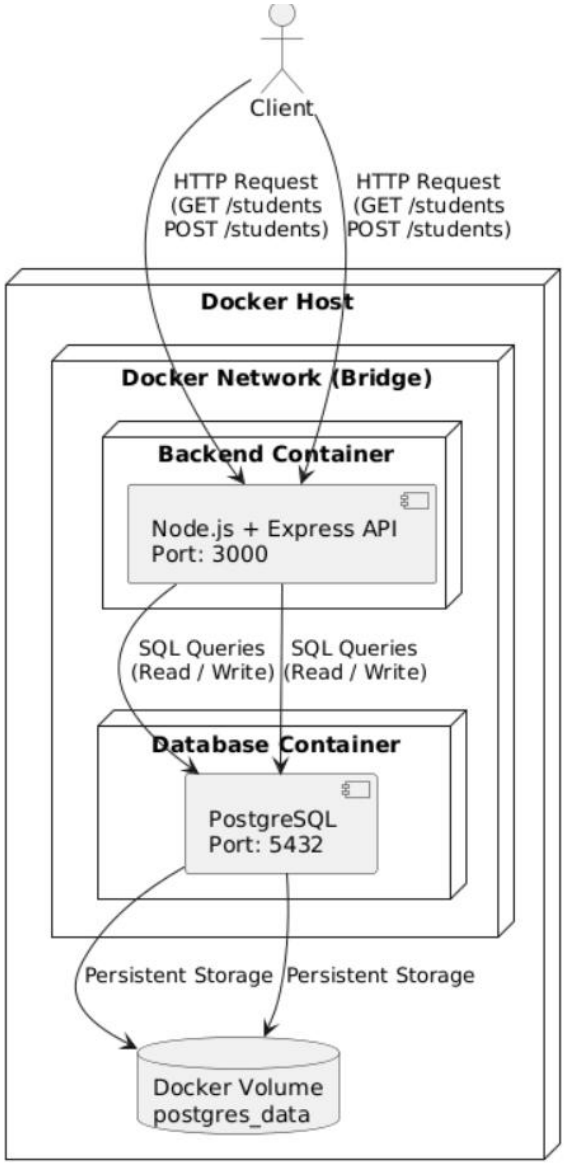
Database Container

- Uses PostgreSQL

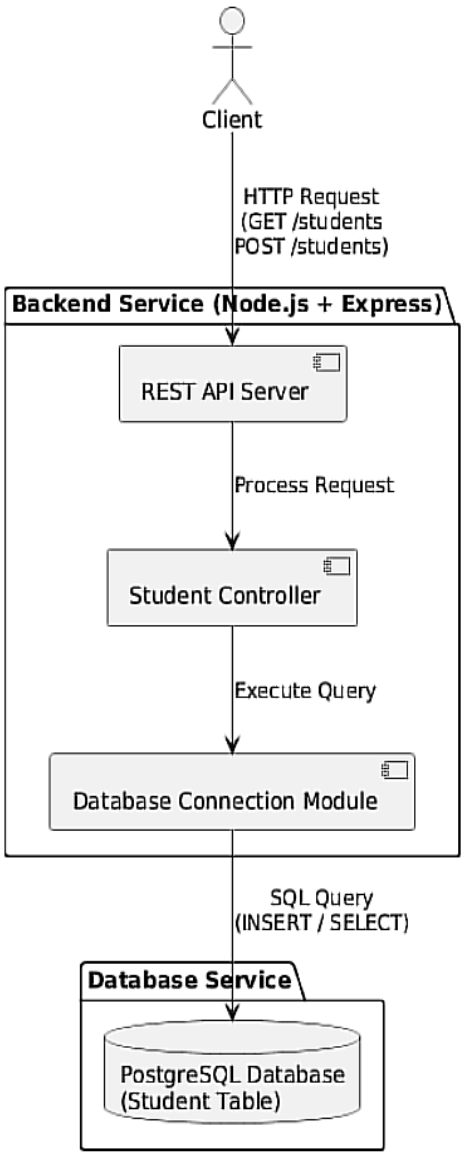
- Stores student records
- Persists data using Docker volumes

Architecture Flow

Deployment Diagram - Containerized Web Application



Component Diagram - Containerized Web Application



Component	Role
Client	Sends HTTP API requests
Docker Host	Runs Docker containers
Docker Compose	Manages multi-container setup
Docker Network	Enables container communication
Backend Container	Runs Node.js Express API
PostgreSQL Container	Stores application data
Docker Volume	Provides persistent database storage

3. Build Optimization Explanation

Docker images were optimized using several best practices to reduce size and improve performance.

3.1 Lightweight Base Images

The backend container uses the Node.js Alpine image, which is significantly smaller than standard images. Alpine Linux provides a minimal environment while maintaining required functionality.

Benefits:

- Reduced image size
- Faster build and deployment
- Lower storage requirements

3.2 Docker Layer Caching

Dockerfile instructions were structured in an optimized order:

1. Copy package.json
2. Install dependencies
3. Copy application files

This ensures that dependencies are not reinstalled unless package files change, improving build efficiency.

3.3 .dockerignore Usage

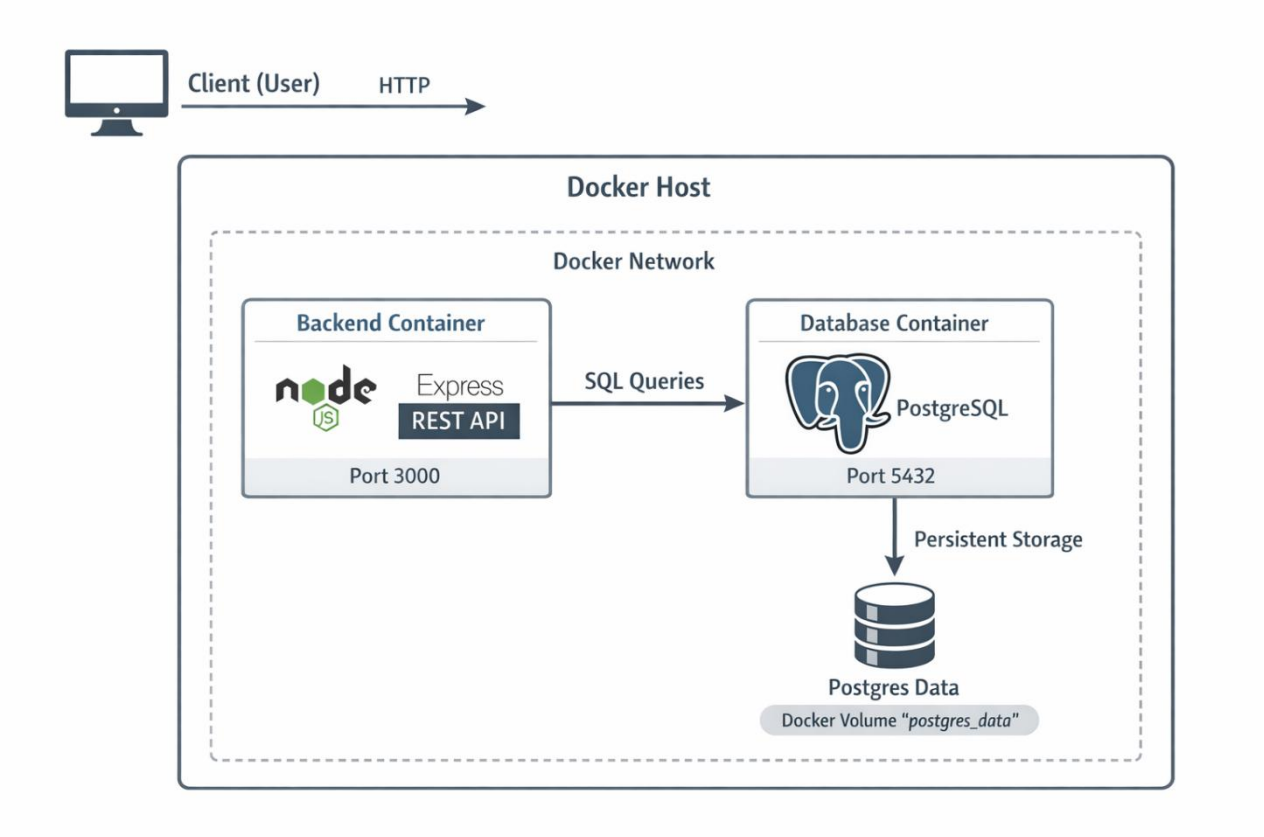
The .dockerignore file excludes unnecessary files from the build context such as:

- node_modules
- logs
- temporary files

This reduces build time and prevents unnecessary data from being added to the container image.

4. Network Design Diagram

The application uses Docker networking to allow communication between containers.



Network Characteristics:

1. Client–Server Communication
The client sends HTTP requests to the backend API running in the Docker container.

2. Docker Bridge Network
Both containers (Backend and PostgreSQL) are connected through a Docker network that allows them to communicate securely.

3. Container-to-Container Communication
Containers communicate internally using service names instead of IP addresses.

4. Port Exposure
The backend container exposes **port 3000** to allow external users to access the API.

5. Internal Database Communication
The backend container sends SQL queries to the PostgreSQL container through the internal Docker network.

6. Persistent Storage via Volume
PostgreSQL stores data in the Docker volume **postgres_data** to ensure data remains even if the container restarts.

Examples:

Example 1:
A user sends a request GET /students to the backend API on **port 3000**.

Example 2:
The backend container sends an **SQL query** to the PostgreSQL container on **port 5432** to retrieve student records.

Example 3:
The PostgreSQL container stores and retrieves data from the **Docker volume postgres_data**, ensuring persistent storage.

5. Image Size Comparison

Container image size is an important factor affecting build speed and storage requirements.

Image	Base Image	Approx Size
Backend API	node:18-alpine	~130 MB
PostgreSQL	postgres:15-alpine	~274 MB

Observations

- Alpine images significantly reduce container size.
- Smaller images improve deployment speed.
- Reduced attack surface improves security.

6. Macvlan vs IPvlan Networking Comparison

Docker supports multiple network drivers. Two commonly used drivers are **Macvlan** and **IPvlan**.

Macvlan Network

Macvlan assigns a **unique MAC address and IP address** to each container.

Characteristics:

- Containers appear as physical devices on the network
- Useful for applications requiring direct network access
- Suitable for legacy applications

Advantages:

- Direct communication with external network
- Improved network isolation

Disadvantages:

- Host machine cannot directly communicate with containers
- Requires proper network configuration

IPvlan Network

IPvlan assigns IP addresses to containers but uses the **same MAC address as the host**.

Characteristics:

- Lightweight networking mode
- Reduced network overhead
- Works well in restricted network environments

Advantages:

- Better performance
- Simplified network configuration
- Host communication supported

Disadvantages:

- Less isolation compared to macvlan

Comparison Summary

Feature	Macvlan	IPvlan
MAC Address	Unique per container	Same as host
Network Isolation	High	Moderate
Host Communication	Not direct	Supported
Performance	Good	Better

7. Testing and Verification

The application was tested using **curl commands** to verify API functionality.

Insert Record

```
curl -X POST http://localhost:3000/students \
-H "Content-Type: application/json" \
-d '{"name":"Vanshika","age":21}'
```

Fetch Records

```
curl http://localhost:3000/students
```

Results

- API successfully inserted records into the PostgreSQL database.
- Data retrieval confirmed proper backend-database communication.
- Docker volumes ensured data persistence even after container restart.

8. Conclusion

This project demonstrated how containerization can be used to deploy a multi-service application efficiently. Using Docker and Docker Compose allowed the backend application and database to run in isolated environments while maintaining seamless communication.

Key outcomes of the project include:

- Successful containerization of a Node.js application
- Integration with a PostgreSQL database
- Implementation of Docker networking
- Persistent storage using Docker volumes
- Efficient image builds using Alpine base images

The experiment highlights the importance of containerization in modern DevOps practices and scalable application deployment.