Table of contents

1 Getting started with docker ?

Chapter 1: Introduction to Docker

1.1 What is Docker?

Docker is a platform that enables developers to package, deploy, and run applications in containers. Containers include everything needed to run the application, making it portable and consistent across different environments.

1.2 Why Use Docker?

- Consistency across development, testing, and production environments
- Isolation and security
- Simplified dependency management
- Efficient resource utilization

Chapter 2: Installing Docker on Windows

2.1 Docker Installation

1. Download Docker Desktop from the Docker website (https://www.docker.com/products/docker-desktop).
2. Run the installer and follow the on-screen instructions.
3. After installation, launch Docker Desktop.
4. Ensure Docker is running by opening a terminal and typing:

   ```
   docker --version
   ```

Chapter 3: Docker Basics

3.1 Docker Architecture

- **Docker Client**: CLI to interact with Docker.
- **Docker Daemon**: Runs on the host machine, manages Docker objects.
- **Docker Images**: Read-only templates to create containers.
- **Docker Containers**: Running instances of Docker images.
- **Docker Registry**: Stores Docker images.

3.2 Hello World in Docker

1. Open PowerShell or Command Prompt.
2. Run your first container:

```
docker run hello-world
```

3.3 Docker CLI Basics

- List Docker CLI commands:

```
docker
```

- Get help on a command:

```
docker <command> --help
```

Chapter 4: Working with Docker Images

4.1 Pulling Images

- Pull an image from Docker Hub:

```
docker pull node
```

4.2 Listing Images

- List all images on your system:

```
docker images
```

4.3 Removing Images

- Remove an image:

```
docker rmi <image_id>
```

Chapter 5: Docker Containers

5.1 Running Containers

- Run a Node.js container interactively:

```
docker run -it node /bin/bash
```

- Run a container in the background:

```
docker run -d node
```

5.2 Listing Containers

- List all running containers:

```
docker ps
```

- List all containers (including stopped):

```
docker ps -a
```

5.3 Stopping Containers

- Stop a running container:

```
docker stop <container_id>
```

5.4 Removing Containers

- Remove a container:

  ```
  docker rm <container_id>
  ```

Chapter 6: Dockerfile

6.1 Introduction to Dockerfile

A Dockerfile is a text document that contains instructions for building a Docker image.

6.2 Creating a Dockerfile for Node.js

1. Create a directory for your Node.js application, e.g., `my-node-app`.
2. Inside this directory, create a file named `Dockerfile`.
3. Add the following content:

```
# Use an official Node.js runtime as a parent image
FROM node:14

# Set the working directory in the container
WORKDIR /usr/src/app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the rest of the application code
COPY . .

# Expose port 3000
EXPOSE 3000
```

```
# Command to run the application
CMD ["node", "index.js"]
```

6.3 Building an Image

- Build an image from the Dockerfile:

```
docker build -t my-node-app .
```

6.4 Running Your Image

- Run the image as a container:

```
docker run -p 3000:3000 my-node-app
```

Chapter 7: Docker Volumes

7.1 Introduction to Volumes

Volumes are used to persist data generated by and used by Docker containers.

7.2 Creating Volumes

- Create a volume:

```
docker volume create my-volume
```

7.3 Using Volumes

- Use a volume in a container:

```
docker run -d -v my-volume:/usr/src/app my-node-app
```

Chapter 8: Docker Compose

8.1 Introduction to Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications.

8.2 Creating a docker-compose.yml for Node.js and React

1. In your project directory, create a file named `docker-compose.yml`.

2. Add the following content:

```yaml
version: '3'

services:
  web:
    image: my-node-app
    build: .
    ports:
      - "3000:3000"
    volumes:
      - .:/usr/src/app
    environment:
      - NODE_ENV=development
  client:
    image: node:14
    working_dir: /usr/src/app
    volumes:
      - ./client:/usr/src/app
    command: npm start
    ports:
      - "3001:3001"
```

8.3 Running Docker Compose

- Start your application:

```
docker-compose up
```

- Stop your application:

```
docker-compose down
```

Chapter 9: Docker Networking

9.1 Introduction to Docker Networking

Docker provides a networking model to allow containers to communicate with each other and with non-Docker workloads.

9.2 Listing Networks

- List all Docker networks:

```
docker network ls
```

9.3 Creating a Network

- Create a custom network:

```
docker network create my-network
```

9.4 Connecting Containers to a Network

- Connect a container to a network:

```
docker network connect my-network <container_id>
```

9.5 Disconnecting Containers from a Network

- Disconnect a container from a network:

```
docker network disconnect my-network <container_id>
```

Chapter 10: Docker Swarm

10.1 Introduction to Docker Swarm

Docker Swarm is a container orchestration tool that allows you to manage a cluster of Docker nodes.

10.2 Initializing a Swarm

- Initialize a swarm:

```
docker swarm init
```

10.3 Joining a Swarm

- Get the join command from the manager node and run it on the worker node:

```
docker swarm join --token <token> <manager_ip>:2377
```

10.4 Deploying a Service

- Deploy a service in the swarm:

```
docker service create --name my-web-service -p 3000:3000 my-node-app
```

10.5 Listing Services

- List all services in the swarm:

```
docker service ls
```

10.6 Removing a Service

- Remove a service:

```
docker service rm my-web-service
```

Chapter 11: Docker Best Practices

11.1 Writing Efficient Dockerfiles

- Use official images as a base.
- Minimize the number of layers.
- Use multi-stage builds for optimized images.

11.2 Managing Secrets

- Use Docker secrets to manage sensitive data:

  ```
  echo "my_secret_password" | docker secret create my_secret -
  ```

11.3 Security Practices

- Run containers as a non-root user.
- Keep the host and Docker up to date.

Chapter 12: Advanced Topics

12.1 Docker with Kubernetes

- Install and configure Kubernetes.
- Deploy Docker containers using Kubernetes.

12.2 CI/CD with Docker

- Use Docker in your CI/CD pipeline.
- Example with Jenkins:

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                script {
                    dockerImage = docker.build("my-node-app")
                }
            }
        }
        stage('Test') {
            steps {
                script {
                    dockerImage.inside {
                        sh 'npm test'
                    }
                }
            }
        }
        stage('Deploy') {
            steps {
                script {
                    dockerImage.push('my-repo/my-node-app')
                }
            }
        }
    }
}
```

2 Dockerizing MongoDB

Chapter 1: Introduction

This guide will walk you through the steps of Dockerizing MongoDB, setting up a user with specific credentials, persisting data using Docker volumes, and performing basic CRUD (Create, Read, Update, Delete) operations. We will cover everything from pulling the MongoDB image to running queries.

Chapter 2: Setting Up Docker

2.1 Installing Docker on Windows

1. Download Docker Desktop from the Docker website (https://www.docker.com/products/docker-desktop).
2. Run the installer and follow the instructions.
3. After installation, start Docker Desktop.
4. Verify the installation:

```
docker --version
```

Chapter 3: Dockerizing MongoDB

3.1 Pulling the MongoDB Image

- Open a command prompt and run the following command to pull the official MongoDB image:

```
docker pull mongo
```

3.2 Creating a Docker Volume

- Create a volume to persist MongoDB data:

```
docker volume create mongodb-data
```

3.3 Running MongoDB Container with Authentication and Persistent Storage

- Run the MongoDB container with environment variables to set the username and password, and use the volume for data persistence:

```
docker run --name mongodb_container -d -p 27017:27017 -e
MONGO_INITDB_ROOT_USERNAME=admin -e MONGO_INITDB_ROOT_PASSWORD=pass -v
mongodb-data:/data/db mongo
```

- Verify the container is running:

```
docker ps
```

Chapter 4: Connecting to MongoDB

4.1 Using MongoDB Shell

You have two options to connect to MongoDB shell:

Option 1: Direct Command

- Start the MongoDB shell directly:

```
docker exec -it mongodb_container bash -c 'mongosh -u admin -p pass --
authenticationDatabase admin'
```

Option 2: Entering the Container First

1. Start a bash shell inside the running MongoDB container:

```
docker exec -it mongodb_container /bin/bash
```

2. Once inside the container, start the MongoDB shell:

```
mongosh -u admin -p pass --authenticationDatabase admin
```

3. You should now be in the MongoDB shell:

```
>
```

4. List all databases:

```
> show databases
```

Chapter 5: CRUD Operations

5.1 Creating a Database and Collection

1. Create a new database called `mydatabase`:

```
use mydatabase
```

2. Create a new collection called `mycollection`:

```
db.createCollection("mycollection")
```

5.2 Create (Insert) Documents

1. Insert a single document into `mycollection`:

```
db.mycollection.insertOne({ name: "John Doe", age: 30, occupation:
"Engineer" })
```

2. Insert multiple documents:

```
db.mycollection.insertMany([
  { name: "Jane Doe", age: 25, occupation: "Teacher" },
  { name: "Steve Smith", age: 40, occupation: "Chef" }
])
```

5.3 Read (Query) Documents

1. Find one document:

```
db.mycollection.findOne({ name: "John Doe" })
```

2. Find all documents:

```
db.mycollection.find()
```

3. Find documents with a condition:

```
db.mycollection.find({ age: { $gt: 30 } })
```

5.4 Update Documents

1. Update a single document:

```
db.mycollection.updateOne({ name: "John Doe" }, { $set: { age: 31 } })
```

2. Update multiple documents:

```
db.mycollection.updateMany({ occupation: "Chef" }, { $set: {
occupation: "Head Chef" } })
```

5.5 Delete Documents

1. Delete a single document:

```
db.mycollection.deleteOne({ name: "John Doe" })
```

2. Delete multiple documents:

```
db.mycollection.deleteMany({ age: { $lt: 30 } })
```

Chapter 6: Accessing MongoDB from an Application

6.1 Using MongoDB with Node.js

1. Install Node.js from the official website (https://nodejs.org/).

2. Create a new project directory and navigate into it:

```
mkdir my-mongo-app
cd my-mongo-app
```

3. Initialize a new Node.js project:

```
npm init -y
```

4. Install the MongoDB driver:

```
npm install mongodb
```

5. Create an `index.js` file and add the following code:

```
const { MongoClient } = require('mongodb');

async function main() {
  const uri = "mongodb://admin:pass@localhost:27017/?
authSource=admin";
  const client = new MongoClient(uri);

  try {
    await client.connect();

    const database = client.db('mydatabase');
    const collection = database.collection('mycollection');

    // Insert a document
    const insertResult = await collection.insertOne({ name: "Alice",
age: 28, occupation: "Designer" });
    console.log('Inserted document:', insertResult.insertedId);

    // Find a document
    const findResult = await collection.findOne({ name: "Alice" });
    console.log('Found document:', findResult);

    // Update a document
    const updateResult = await collection.updateOne({ name: "Alice" },
{ $set: { age: 29 } });
    console.log('Updated document:', updateResult.modifiedCount);

    // Delete a document
    const deleteResult = await collection.deleteOne({ name: "Alice"
});
    console.log('Deleted document:', deleteResult.deletedCount);
  } finally {
    await client.close();
  }
}

main().catch(console.error);
```

6. Run the application:

```
node index.js
```

Chapter 7: Cleaning Up

7.1 Stopping and Removing the MongoDB Container

- Stop the container:

```
docker stop mongodb_container
```

- Remove the container:

```
docker rm mongodb_container
```

7.2 Removing the MongoDB Image

- Remove the MongoDB image:

```
docker rmi mongo
```

7.3 Removing the Docker Volume

- Remove the Docker volume:

```
docker volume rm mongodb-data
```

3 Dockerizing PostgreSQL

Chapter 1: Introduction

This guide will walk you through the steps of Dockerizing PostgreSQL, setting up a user with specific credentials, persisting data using Docker volumes, and performing basic CRUD (Create, Read, Update, Delete) operations. We will cover everything from pulling the PostgreSQL image to running queries.

Chapter 2: Setting Up Docker

2.1 Installing Docker on Windows

1. Download Docker Desktop from the Docker website (https://www.docker.com/products/docker-desktop).
2. Run the installer and follow the instructions.
3. After installation, start Docker Desktop.
4. Verify the installation:

    ```
    docker --version
    ```

Chapter 3: Dockerizing PostgreSQL

3.1 Pulling the PostgreSQL Image

- Open a command prompt and run the following command to pull the official PostgreSQL image:

    ```
    docker pull postgres
    ```

3.2 Creating a Docker Volume

- Create a volume to persist PostgreSQL data:

```
docker volume create postgres-data
```

3.3 Running PostgreSQL Container with Authentication and Persistent Storage

- Run the PostgreSQL container with environment variables to set the username and password, and use the volume for data persistence:

```
docker run --name postgres_container -d -p 5432:5432 -e
POSTGRES_USER=admin -e POSTGRES_PASSWORD=pass -v postgres-
data:/var/lib/postgresql/data postgres
```

- Verify the container is running:

```
docker ps
```

Chapter 4: Connecting to PostgreSQL

4.1 Using psql Shell

You have two options to connect to the PostgreSQL shell:

Option 1: Direct Command

- Start the PostgreSQL shell directly:

```
docker exec -it postgres_container psql -U admin
```

Option 2: Entering the Container First

1. Start a bash shell inside the running PostgreSQL container:

```
docker exec -it postgres_container /bin/bash
```

2. Once inside the container, start the PostgreSQL shell:

```
psql -U admin
```

3. You should now be in the PostgreSQL shell:

```
admin=#
```

4. In the PostgreSQL shell (psql), you can list all databases using the following command:

   - List All Databases

```
\l
```

   - or

```
\list
```

5. Additional Useful Commands

- List all tables in the current database:

```
\dt
```

- List all schemas in the current database:

```
\dn
```

- List all users:

```
\du
```

- List all indexes:

```
\di
```

Chapter 5: CRUD Operations

5.1 Creating a Database and Table

1. Create a new database called `mydatabase`:

```
CREATE DATABASE mydatabase;
```

2. Connect to the new database:

```
\c mydatabase
```

3. Create a new table called `mytable`:

```
CREATE TABLE mytable (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    age INT,
    occupation VARCHAR(100)
);
```

5.2 Create (Insert) Records

1. Insert a single record into `mytable`:

```
INSERT INTO mytable (name, age, occupation) VALUES ('John Doe', 30,
'Engineer');
```

2. Insert multiple records:

```
INSERT INTO mytable (name, age, occupation) VALUES
('Jane Doe', 25, 'Teacher'),
('Steve Smith', 40, 'Chef');
```

5.3 Read (Query) Records

1. Select one record:

```
SELECT * FROM mytable WHERE name = 'John Doe';
```

2. Select all records:

```
SELECT * FROM mytable;
```

3. Select records with a condition:

```
SELECT * FROM mytable WHERE age > 30;
```

5.4 Update Records

1. Update a single record:

```
UPDATE mytable SET age = 31 WHERE name = 'John Doe';
```

2. Update multiple records:

```
UPDATE mytable SET occupation = 'Head Chef' WHERE occupation = 'Chef';
```

5.5 Delete Records

1. Delete a single record:

```
DELETE FROM mytable WHERE name = 'John Doe';
```

2. Delete multiple records:

```
DELETE FROM mytable WHERE age < 30;
```

Chapter 6: Accessing PostgreSQL from an Application

6.1 Using PostgreSQL with Node.js

1. Install Node.js from the official website (https://nodejs.org/).

2. Create a new project directory and navigate into it:

```
mkdir my-postgres-app
cd my-postgres-app
```

3. Initialize a new Node.js project:

```
npm init -y
```

4. Install the `pg` package:

```
npm install pg
```

5. Create an `index.js` file and add the following code:

```
const { Client } = require('pg');

async function main() {
  const client = new Client({
    user: 'admin',
    host: 'localhost',
    database: 'mydatabase',
    password: 'pass',
    port: 5432,
  });

  await client.connect();

  try {
    // Insert a record
    const insertResult = await client.query("INSERT INTO mytable
(name, age, occupation) VALUES ('Alice', 28, 'Designer') RETURNING
id");
    console.log('Inserted record ID:', insertResult.rows[0].id);

    // Select a record
    const selectResult = await client.query("SELECT * FROM mytable
WHERE name = 'Alice'");
    console.log('Selected record:', selectResult.rows[0]);

    // Update a record
    const updateResult = await client.query("UPDATE mytable SET age =
29 WHERE name = 'Alice'");
    console.log('Updated record count:', updateResult.rowCount);

    // Delete a record
    const deleteResult = await client.query("DELETE FROM mytable WHERE
name = 'Alice'");
    console.log('Deleted record count:', deleteResult.rowCount);
  } finally {
    await client.end();
  }
}

main().catch(console.error);
```

6. Run the application:

```
node index.js
```

Chapter 7: Cleaning Up

7.1 Stopping and Removing the PostgreSQL Container

- Stop the container:

```
docker stop postgres_container
```

- Remove the container:

```
docker rm postgres_container
```

7.2 Removing the PostgreSQL Image

- Remove the PostgreSQL image:

```
docker rmi postgres
```

7.3 Removing the Docker Volume

- Remove the Docker volume:

```
docker volume rm postgres-data
```

4 Dockerizing MySQL

Chapter 1: Introduction

This guide will walk you through the steps of Dockerizing MySQL, setting up a user with specific credentials, persisting data using Docker volumes, and performing basic CRUD (Create, Read, Update, Delete) operations. We will cover everything from pulling the MySQL image to running queries.

Chapter 2: Setting Up Docker

2.1 Installing Docker on Windows

1. Download Docker Desktop from the Docker website (https://www.docker.com/products/docker-desktop).
2. Run the installer and follow the instructions.
3. After installation, start Docker Desktop.
4. Verify the installation:

```
docker --version
```

Chapter 3: Dockerizing MySQL

3.1 Pulling the MySQL Image

- Open a command prompt and run the following command to pull the official MySQL image:

```
docker pull mysql
```

3.2 Creating a Docker Volume

- Create a volume to persist MySQL data:

```
docker volume create mysql-data
```

3.3 Running MySQL Container with Authentication and Persistent Storage

- Run the MySQL container with environment variables to set the username and password, and use the volume for data persistence:

```
docker run --name mysql_container -d -p 3306:3306 -e
MYSQL_ROOT_PASSWORD=pass -e MYSQL_USER=admin -e MYSQL_PASSWORD=pass -e
MYSQL_DATABASE=mydatabase -v mysql-data:/var/lib/mysql mysql
```

- Verify the container is running:

```
docker ps
```

Chapter 4: Connecting to MySQL

4.1 Using MySQL Shell

You have two options to connect to the MySQL shell:

Option 1: Direct Command

- Start the MySQL shell directly:

```
docker exec -it mysql_container mysql -u admin -p
```

- you will be prompted to Enter password:

```
Enter password: pass
```

Option 2: Entering the Container First

1. Start a bash shell inside the running MySQL container:

```
docker exec -it mysql_container /bin/bash
```

2. Once inside the container, start the MySQL shell:

```
mysql -u admin -p
```

3. Enter the password when prompted (`pass` in this example).

4. You should now be in the MySQL shell:

```
mysql>
```

5. In the MySQL shell, you can list all databases using the following command:

- List All Databases:

```
SHOW DATABASES;
```

Chapter 5: CRUD Operations

5.1 Creating a Database and Table

1. Create a new database called `mydatabase`:

```
CREATE DATABASE mydatabase;
```

2. Select the new database:

```
USE mydatabase;
```

3. Create a new table called `mytable`:

```
CREATE TABLE mytable (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100),
    age INT,
    occupation VARCHAR(100)
);
```

5.2 Create (Insert) Records

1. Insert a single record into `mytable`:

```
INSERT INTO mytable (name, age, occupation) VALUES ('John Doe', 30,
'Engineer');
```

2. Insert multiple records:

```
INSERT INTO mytable (name, age, occupation) VALUES
('Jane Doe', 25, 'Teacher'),
('Steve Smith', 40, 'Chef');
```

5.3 Read (Query) Records

1. Select one record:

```
SELECT * FROM mytable WHERE name = 'John Doe';
```

2. Select all records:

```
SELECT * FROM mytable;
```

3. Select records with a condition:

```
SELECT * FROM mytable WHERE age > 30;
```

5.4 Update Records

1. Update a single record:

```
UPDATE mytable SET age = 31 WHERE name = 'John Doe';
```

2. Update multiple records:

```
UPDATE mytable SET occupation = 'Head Chef' WHERE occupation = 'Chef';
```

5.5 Delete Records

1. Delete a single record:

```
DELETE FROM mytable WHERE name = 'John Doe';
```

2. Delete multiple records:

```
DELETE FROM mytable WHERE age < 30;
```

5.6 Additional Useful Commands

- List all tables in the current database:

```
SHOW TABLES;
```

- Describe the structure of a table:

```
DESCRIBE mytable;
```

- List all users:

```
SELECT User, Host FROM mysql.user;
```

Chapter 6: Accessing MySQL from an Application

6.1 Using MySQL with Node.js

1. Install Node.js from the official website (https://nodejs.org/).

2. Create a new project directory and navigate into it:

```
mkdir my-mysql-app
```

```
cd my-mysql-app
```

3. Initialize a new Node.js project:

```
npm init -y
```

4. Install the `mysql` package:

```
npm install mysql2
```

5. Create an `index.js` file and add the following code:

```
const mysql = require('mysql2/promise');

 async function main() {
 const connection = await mysql.createConnection({
 host: 'localhost',
 user: 'admin',
 password: 'pass',
 database: 'mydatabase'
 });

 try {
 console.log('connected as id ' + connection.threadId);

     // Insert a record
     const [insertResults] = await connection.execute(
       "INSERT INTO mytable (name, age, occupation) VALUES (?, ?, ?)",
       ['Alice', 28, 'Designer']
     );
     console.log('Inserted record ID:', insertResults.insertId);

     // Select a record
     const [selectResults] = await connection.execute(
       "SELECT * FROM mytable WHERE name = ?",
       ['Alice']
```

```
    );
    console.log('Selected record:', selectResults[0]);

    // Update a record
    const [updateResults] = await connection.execute(
      "UPDATE mytable SET age = ? WHERE name = ?",
      [29, 'Alice']
    );
    console.log('Updated record count:', updateResults.affectedRows);

    // Delete a record
    const [deleteResults] = await connection.execute(
      "DELETE FROM mytable WHERE name = ?",
      ['Alice']
    );
    console.log('Deleted record count:', deleteResults.affectedRows);
} catch (err) {
console.error('error:', err.stack);
} finally {
await connection.end();
}
}

main();
```

6. Run the application:

```
node index.js
```

Chapter 7: Cleaning Up

7.1 Stopping and Removing the MySQL Container

- Stop the container:

```
docker stop mysql_container
```

- Remove the container:

```
docker rm mysql_container
```

7.2 Removing the MySQL Image

- Remove the MySQL image:

```
docker rmi mysql
```

7.3 Removing the Docker Volume

- Remove the Docker volume:

```
docker volume rm mysql-data
```

5 Dockerizing MSSQL

Chapter 1: Introduction

This guide will walk you through the steps of Dockerizing Microsoft SQL Server (MSSQL), setting up a user with specific credentials, persisting data using Docker volumes, and performing basic CRUD (Create, Read, Update, Delete) operations on a Windows system. We will cover everything from pulling the MSSQL image to running queries.

Chapter 2: Setting Up Docker

2.1 Installing Docker on Windows

1. Download Docker Desktop from the Docker website (https://www.docker.com/products/docker-desktop).
2. Run the installer and follow the instructions.
3. After installation, start Docker Desktop.
4. Verify the installation:

```
docker --version
```

Chapter 3: Dockerizing MSSQL Server

3.1 Pulling the MSSQL Server Image

- Open a command prompt or PowerShell and run the following command to pull the official MSSQL Server image:

```
docker pull mcr.microsoft.com/mssql/server
```

3.2 Creating a Docker Volume

- Create a volume to persist MSSQL Server data:

```
docker volume create mssql-data
```

3.3 Running MSSQL Server Container with Authentication and Persistent Storage

- Run the MSSQL Server container with environment variables to set the SA password and use the volume for data persistence:

```
docker run -e "ACCEPT_EULA=Y" -e "SA_PASSWORD=yourStrong(!)Password" -
p 1433:1433 --name mssql_container -v mssql-data:/var/opt/mssql -d
mcr.microsoft.com/mssql/server
```

- Verify the container is running:

```
docker ps
```

Chapter 4: Installing MSSQL Command Line Tools

4.1 Download and Install MSSQL Tools

1. Download the Microsoft ODBC Driver 17 for SQL Server from the Microsoft website (https://docs.microsoft.com/en-us/sql/connect/odbc/download-odbc-driver-for-sql-server).
2. Install the ODBC driver by running the downloaded installer.
3. Download the SQL Server Command Line Tools (`sqlcmd` and `bcp`) from the Microsoft website (https://docs.microsoft.com/en-us/sql/tools/sqlcmd-utility).
4. Install the SQL Server Command Line Tools by running the downloaded installer.

Chapter 5: Connecting to MSSQL Server

5.1 Using MSSQL Server Command Line Tools

1. Open Command Prompt or PowerShell.
2. Connect to the MSSQL Server using `sqlcmd`:

```
sqlcmd -S localhost -U SA -P "yourStrong(!)Password"
```

3. You should now be in the MSSQL command line:

```
1>
```

4. In the MSSQL command line, you can list all databases using the following command:

```
SELECT name FROM sys.databases;
GO
```

Example Session

- Here's how an example session might look:

```
C:\> sqlcmd -S localhost -U SA -P "yourStrong(!)Password"
1> SELECT name FROM sys.databases;
2> GO
name
----------------------------------------------------------------------
----------------------------------------------------------
master
tempdb
model
msdb
mydatabase

(5 rows affected)
1>
```

Chapter 6: CRUD Operations

6.1 Creating a Database and Table

1. Create a new database called `mydatabase`:

```
CREATE DATABASE mydatabase;
GO
```

2. Use the new database:

```
USE mydatabase;
GO
```

3. Create a new table called `mytable`:

```
CREATE TABLE mytable (
    id INT PRIMARY KEY IDENTITY(1,1),
    name NVARCHAR(100),
    age INT,
    occupation NVARCHAR(100)
);
GO
```

6.2 Create (Insert) Records

1. Insert a single record into `mytable`:

```
INSERT INTO mytable (name, age, occupation) VALUES ('John Doe', 30,
'Engineer');
GO
```

2. Insert multiple records:

```
INSERT INTO mytable (name, age, occupation) VALUES
('Jane Doe', 25, 'Teacher'),
('Steve Smith', 40, 'Chef');
GO
```

6.3 Read (Query) Records

1. Select one record:

```
SELECT * FROM mytable WHERE name = 'John Doe';
GO
```

2. Select all records:

```
SELECT * FROM mytable;
GO
```

3. Select records with a condition:

```
SELECT * FROM mytable WHERE age > 30;
GO
```

6.4 Update Records

1. Update a single record:

```
UPDATE mytable SET age = 31 WHERE name = 'John Doe';
GO
```

2. Update multiple records:

```
UPDATE mytable SET occupation = 'Head Chef' WHERE occupation = 'Chef';
GO
```

6.5 Delete Records

1. Delete a single record:

```
DELETE FROM mytable WHERE name = 'John Doe';
GO
```

2. Delete multiple records:

```
DELETE FROM mytable WHERE age < 30;
```

```
GO
```

6.6 Additional Useful Commands

- List all tables in the current database:

```
SELECT * FROM sys.Tables;
GO
```

- Describe the structure of a table:

```
sp_help mytable;
GO
```

- List all users:

```
SELECT name FROM sys.sql_logins;
GO
```

Chapter 7: Accessing MSSQL Server from an Application

7.1 Using MSSQL Server with Node.js

1. Install Node.js from the official website (https://nodejs.org/).

2. Create a new project directory and navigate into it:

```
mkdir my-mssql-app
cd my-mssql-app
```

3. Initialize a new Node.js project:

```
npm init -y
```

4. Install the `mssql` package:

```
npm install mssql
```

5. Create an `index.js` file and add the following code:

```
const sql = require('mssql');

const config = {
  user: 'sa',
  password: 'yourStrong(!)Password',
  server: 'localhost',
  database: 'mydatabase',
  options: {
    encrypt: true, // Use encryption
    trustServerCertificate: true // For self-signed certificate
  }
};

async function main() {
  try {
    let pool = await sql.connect(config);

    // Insert a record
    let insertResult = await pool.request()
      .query("INSERT INTO mytable (name, age, occupation) VALUES
('Alice', 28, 'Designer')");
    console.log('Inserted record:', insertResult);

     // Insert a many
     let insertResult = await pool.request()
         .query(`INSERT INTO mytable (name, age, occupation) VALUES
             ('jane doe', 30, 'Designer'),
             ('kyle Smith', 40, 'Chef')`);
     console.log('Inserted record:', insertResult);

    // Select a record
    let selectResult = await pool.request()
      .query("SELECT * FROM mytable WHERE name = 'Alice'");
    console.log('Selected record:', selectResult.recordset);

    // Update a record
    let updateResult = await pool.request()
      .query("UPDATE mytable SET age = 29 WHERE name = 'Alice'");
    console.log('Updated record:', updateResult);

    // Delete a record
    let deleteResult = await pool.request()
      .query("DELETE FROM mytable WHERE name = 'Alice'");
    console.log('Deleted record:', deleteResult);

  } catch (err) {
    console.error('SQL error', err);
  }
}

main();
```

6. Run the application:

```
node index.js
```

Chapter 8: Cleaning Up

8.1 Stopping and Removing the MSSQL Server Container

- Stop the container:

```
docker stop mssql_container
``
```

- Remove the container:

```
docker rm mssql_container
```

8.2 Removing the MSSQL Server Image

- Remove the MSSQL Server image:

```
docker rmi mcr.microsoft.com/mssql/server
```

8.3 Removing the Docker Volume

- Remove the Docker volume:

```
docker volume rm mssql-data
```

Restore the old Context Menu in Windows 11

1. Right-click the Start button and choose Windows Terminal.
2. Copy the command from below, paste it into Windows Terminal Window, and press enter.

3.
   ```
   reg.exe add "HKCU\Software\Classes\CLSID\{86ca1aa0-34aa-4e8b-a509-
   50c905bae2a2}\InprocServer32" /f /ve
   ```
4. Restart File Explorer or your computer for the changes to take effect.
5. You would see the Legacy Right Click Context menu by default.

Restore Modern Context menus in Windows 11

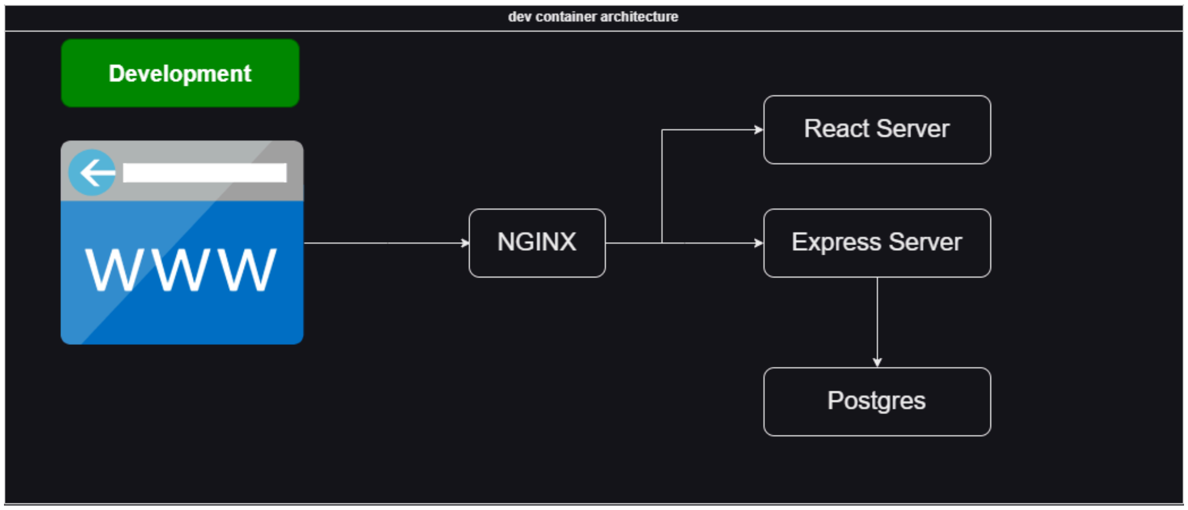
- To undo this change, in a Terminal Window, execute this command:

  ```
  reg.exe delete "HKCU\Software\Classes\CLSID\{86ca1aa0-34aa-4e8b-a509-
  50c905bae2a2}" /f
  ```

Dockerizing a React App with Node.js, Postgres, and Nginx <Dev & Prod>

This guide provides a step-by-step approach to Dockerizing a React application using Vite, with Node.js, Postgres, and Nginx, for both development and production environments.

Development Mode



dev.png

In your development environment, the architecture works as follows:

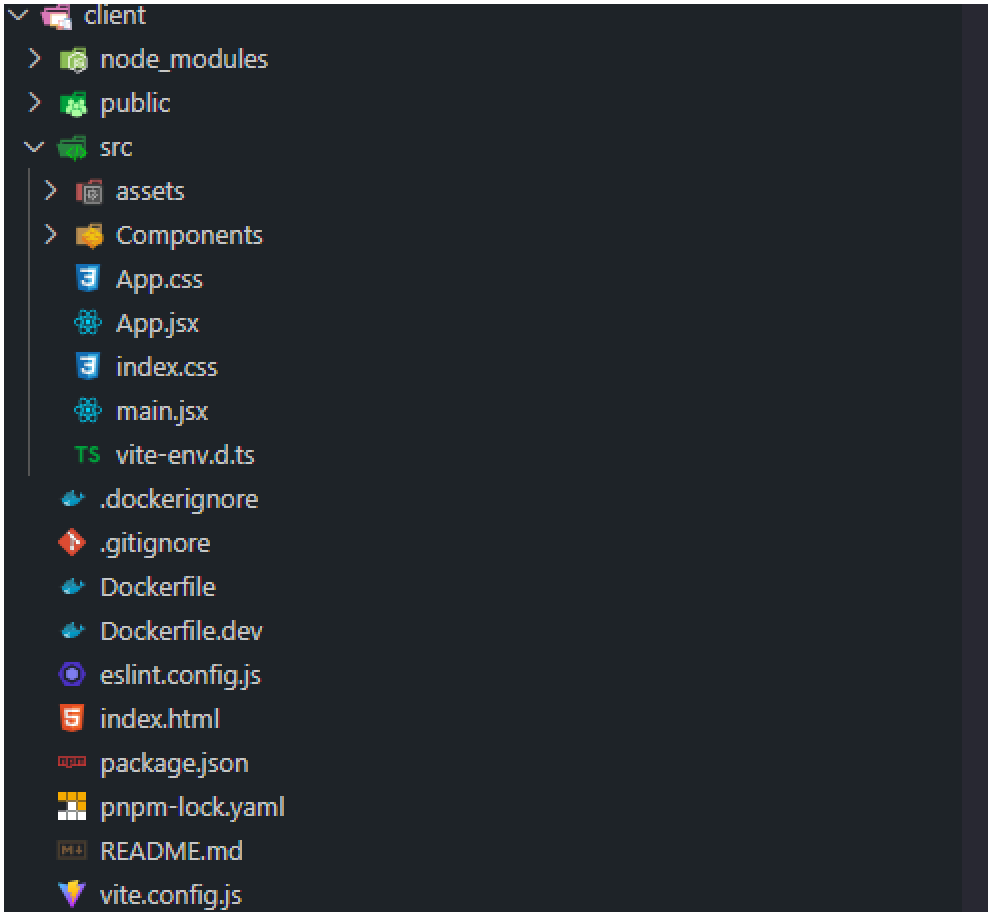
1. **Nginx** acts as a reverse proxy, routing incoming HTTP requests to either the **React server** or the **Express server** based on the request path.
2. **React Server** handles all requests for the frontend (i.e., your React application). When you develop, changes here are served by the React development server, often with hot-reloading.
3. **Express Server** processes backend API requests. These requests might involve querying or updating data in the **Postgres** database.
4. **Nginx** forwards requests:

- Requests to the base URL (e.g., `http://localhost/`) are sent to the React server.
- Requests to paths starting with `/api` are forwarded to the Express server, which may interact with the Postgres database.

This setup allows you to work on both the frontend and backend in a unified environment, ensuring that you can test how they interact together while still enjoying the benefits of a development-focused workflow like hot-reloading and quick feedback.

React Folder & File structure

1. Client: React + Vite



```
client
  node_modules
  public
  src
    assets
    Components
    App.css
    App.jsx
    index.css
    main.jsx
    vite-env.d.ts
  .dockerignore
  .gitignore
  Dockerfile
  Dockerfile.dev
  eslint.config.js
  index.html
  package.json
  pnpm-lock.yaml
  README.md
  vite.config.js
```

react-structure.png

- **Create a Vite React project:**

```
npm create vite@latest my-react-app -- --template react
cd my-react-app
```

```
npm install
```

- **Install necessary dependencies:** You'll need to install `react-router-dom` and `axios` for routing and HTTP requests.

  ```
  npm install react-router-dom@4.3.1 axios
  ```

2. Create the Project Structure

Your project should be structured as follows:

```
my-react-app/
├── node_modules/
├── public/
├── src/
│   ├── assets/
│   ├── Components/
│   │   ├── MainComponent.jsx
│   │   ├── MainComponent.css
│   │   ├── OtherPage.jsx
│   ├── App.css
│   ├── App.jsx
│   ├── index.css
│   ├── main.jsx
│   └── vite-env.d.ts
├── .dockerignore
├── .gitignore
├── Dockerfile.dev
├── Dockerfile
├── eslint.config.js
├── index.html
├── package.json
├── pnpm-lock.yaml
├── README.md
└── vite.config.js
```

3. Add the Required Code

- **src/App.jsx**

```jsx
import './App.css';
import { BrowserRouter as Router, Route, Link } from "react-router-dom";
import MainComponent from './Components/MainComponent';
import OtherPage from './Components/OtherPage';

function App() {
  return (
    <Router>
      <>
        <header className="header">
          <div>This is a multicontainer application</div>
          <Link to="/">Home</Link>
          <Link to="/otherpage">Other page</Link>
        </header>
        <div className="main">
          <Route exact path="/" component={MainComponent} />
          <Route path="/otherpage" component={OtherPage} />
        </div>
      </>
    </Router>
  );
}

export default App;
```

- **src/App.css**

```css
.header {
  background: #eee;
}

.header a {
  margin-left: 20px;
}
```

```
.main {
  padding: 10px;
  background: #ccc;
}
```

- **src/index.css**

```
body {
  margin: 0;
  font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', 'Roboto',
'Oxygen',
    'Ubuntu', 'Cantarell', 'Fira Sans', 'Droid Sans', 'Helvetica Neue',
    sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
}

code {
  font-family: source-code-pro, Menlo, Monaco, Consolas, 'Courier New',
    monospace;
}
```

- **src/main.jsx**

```
import { StrictMode } from 'react';
import { createRoot } from 'react-dom/client';
import App from './App.jsx';
import './index.css';

createRoot(document.getElementById('root')).render(
  <StrictMode>
    <App />
  </StrictMode>,
);
```

- **src/Components/MainComponent.jsx**

```
import { useCallback, useState, useEffect } from "react";
import axios from "axios";
import "./MainComponent.css";

function MainComponent() {
  const [values, setValues] = useState([]);
  const [value, setValue] = useState("");

  const getAllNumbers = useCallback(async () => {
    const data = await axios.get("/api/values/all");
    setValues(data.data.rows.map(row => row.number));
  }, []);

  const saveNumber = useCallback(
    async (event) => {
      event.preventDefault();
      await axios.post("/api/values", { value });

      setValue("");
      getAllNumbers();
    },
    [value, getAllNumbers]
  );

  useEffect(() => {
    getAllNumbers();
  }, []);

  return (
    <div>
      <button onClick={getAllNumbers}>Get all numbers</button>
```

```
        <br />
        <span className="title">Values</span>
        <div className="values">
          {values.map((value, index) => (
            <div className="value" key={index}>{value}</div>
          ))}
        </div>
        <form className="form" onSubmit={saveNumber}>
          <label>Enter your value: </label>
          <input
            value={value}
            onChange={event => {
              setValue(event.target.value);
            }}
          />
          <button>Submit</button>
        </form>
      </div>
    );
}

export default MainComponent;
```

- **src/Components/MainComponent.css**

```
.title {
  font-weight: bold;
}

.values {
  margin-top: 20px;
  background: yellow;
}

.value {
  margin-top: 10px;
  border-top: 1px dashed black;
}

.form {
  margin-top: 20px;
}
```

- **src/Components/OtherPage.jsx**

```
import { Link } from "react-router-dom";

function OtherPage() {
  return (
    <div>
      I'm another page!
      <br />
      <br />
      <Link to="/">Go back to home screen</Link>
    </div>
  );
}

export default OtherPage;
```

4. Configuration Files

- **.dockerignore**

```
node_modules
build
.git
.gitignore
Dockerfile
Dockerfile.dev
.dockerignore
```

```
*.log
.DS_Store
.git
```

- **.gitignore**

```
# Logs
logs
*.log
npm-debug.log*
yarn-debug.log*
yarn-error.log*
pnpm-debug.log*
lerna-debug.log*

node_modules
dist
dist-ssr
*.local

# Editor directories and files
.vscode/*
!.vscode/extensions.json
.idea
.DS_Store
*.suo
*.ntvs*
*.njsproj
*.sln
*.sw?
```

4. Adjust the vite.config.js config

- **vite.config.js**

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'
export default defineConfig({
  plugins: [react()],
  server: {
    host: true,
    port: 5173,
    watch: {
      usePolling: true
    }
  }
})
```

5. Add the docker files

- **Docker.dev**

```
# Use an official Node.js runtime as a parent image
FROM node:20.16.0-alpine
# Set the working directory in the container
WORKDIR /app
# Install dependencies
COPY package.json pnpm-lock.yaml ./
# Copy the rest of the application code
COPY . .
# Expose port 5173 (default Vite port)
EXPOSE 5173
# Command to run the development server
CMD ["npm", "run", "dev"]
```

Build the Docker Image

Navigate to the root directory of your project where the Dockerfile.dev is located. Run the following command to build the Docker image for development:

```
docker build -t my-react-app-dev -f Dockerfile.dev .
```

This command tells Docker to:

-t client: Tag the image as my-react-app-dev. -f Dockerfile.dev: Use the Dockerfile.dev file for building the image.

Run the Docker Container

After building the image, you can run the container with the following command:

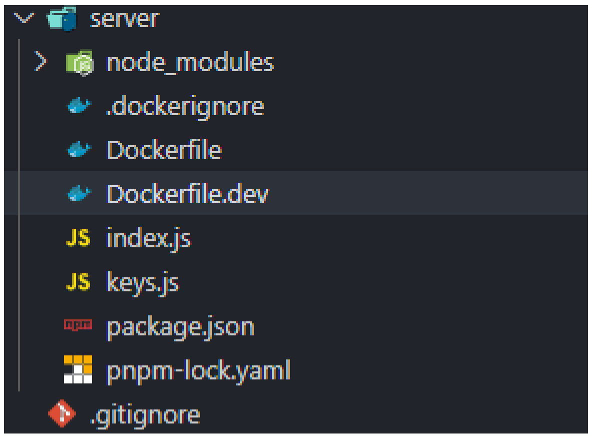
```
docker run -it --rm -p 5173:5173 client
```

Explanation:

- **-it**: Runs the container interactively.
- **--rm**: Automatically removes the container when it exits. -p 5173:5173: Maps port 5173 on your local machine to port 5173 in the container, which is where the Vite development server will be running.

Node Express Folder & File Structure

Here's a guide to create an Express.js application, dockerizing it for development, and managing environment variables securely using a keys.js file.



```
server
  node_modules
  .dockerignore
  Dockerfile
  Dockerfile.dev
  index.js
  keys.js
  package.json
  pnpm-lock.yaml
.gitignore
```

serverfolderstructure.png

Step 1: Set Up the Project Directory

```
server/
├── node_modules/
├── .dockerignore
├── Dockerfile.dev
├── index.js
├── keys.js
├── package.json
├── pnpm-lock.yaml
└── .gitignore
```

Create a new directory for your project:

```
mkdir express-postgres-app
cd express-postgres-app
```

Step 2: Initialize the Node.js Project

Run the following command to initialize a new Node.js project:

```
npm init -y
```

This will create a `package.json` file with default settings.

Step 3: Install Dependencies

Install the required dependencies: Express, PostgreSQL client (pg), CORS, and body-parser:

```
npm install express pg cors body-parser
```

Step 4: Create the Application Files

1. Create index.js

In the root of your project directory, create a file named `index.js` and add the following code:

```
const keys = require("./keys");
const express = require("express");
const bodyParser = require("body-parser");
const cors = require("cors");
const { Pool } = require("pg");

const app = express();

// Middlewares
app.use(cors());
app.use(bodyParser.json());

// Postgres client setup
const pgClient = new Pool({
  user: keys.pgUser,
  host: keys.pgHost,
  database: keys.pgDatabase,
  password: keys.pgPassword,
  port: keys.pgPort
});

pgClient.on("connect", client => {
  client.query("CREATE TABLE IF NOT EXISTS values (number INT)")
    .catch(err => console.log("PG ERROR", err));
});

// Express route definitions
app.get("/", (req, res) => {
  res.send("Hi");
});

// Get all values
app.get("/values/all", async (req, res) => {
  const values = await pgClient.query("SELECT * FROM values");
  res.send(values.rows);
});

// Insert a new value
app.post("/values", async (req, res) => {
  if (!req.body.value) return res.send({ working: false });
  await pgClient.query("INSERT INTO values(number) VALUES($1)",
[req.body.value]);

  res.send({ working: true });
});

app.listen(8000, () => {
  console.log("Listening on port 8000");
});
```

2. Create keys.js

Create a file named `keys.js` in the root directory with the following content:

```
module.exports = {
  pgUser: process.env.PGUSER,
  pgHost: process.env.PGHOST,
  pgDatabase: process.env.PGDATABASE,
  pgPassword: process.env.PGPASSWORD,
  pgPort: process.env.PGPORT
};
```

This file securely references environment variables for your PostgreSQL connection.

Step 5: Create Docker Configuration

1. Create Dockerfile.dev

Create a file named `Dockerfile.dev` in the root directory with the following content:

```
# Use Alpine Node.js runtime as a parent image
FROM node:20.16.0-alpine

# Set the working directory in the container
WORKDIR /app

# Copy package.json and pnpm-lock.yaml (or package-lock.json)
COPY package*.json pnpm-lock.yaml ./
```

```
# Install dependencies
RUN npm install

# Copy the rest of the application code
COPY . .

# Expose port 8000
EXPOSE 8000

# Command to run the application
CMD ["npm", "run", "dev"]
```

2. Create .dockerignore

Create a `.dockerignore` file to ensure unnecessary files aren't copied into the Docker image:

```
node_modules
build
.git
.gitignore
Dockerfile
Dockerfile.dev
.dockerignore
*.log
.DS_Store
.git
```

Step 6: Define the NPM Scripts

Open your `package.json` and add a script for running the development server:

```
{
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js"
  }
}
```

Ensure you have nodemon installed globally or as a dev dependency:

```
npm install -g nodemon
```

Or:

```
npm install --save-dev nodemon
```

Step 7: Running the Application with Docker

1. Build the Docker Image

Run the following command to build the Docker image using the development Dockerfile:

```
docker build -t server -f Dockerfile.dev .
```

2. Run the Docker Container

Run the Docker container:

```
docker run -it --rm -p 8000:8000 server
```

Explanation:

- `-p 8000:8000`: Maps port 8000 on your machine to port 8000 in the container.

Step 8: Test the Application

Open your browser or use a tool like Postman to test the endpoints:

- **GET** request to `http://localhost:8000/` should return "Hi".

NGINX & File structure

Nginx as a reverse proxy to route traffic between a React frontend (client) and an Express backend (API), using Docker. The configuration involves two files: default.conf (Nginx configuration) and Dockerfile.dev (Docker configuration for Nginx).

- Step 1: The structure looks like this:

```
nginx/
├── default.conf          # Nginx configuration file
└── Dockerfile.dev        # Dockerfile for building the Nginx image
```

- Step 2: Configure default.conf for Nginx The default.conf file is an Nginx configuration file that routes traffic based on the request paths. Here's what the configuration does:

```
upstream client {
    server client:5173;
}

upstream api {
    server api:8000;
}

server {
    listen 80;

    location / {
        proxy_pass http://client;
    }

    location /sockjs-node {
        proxy_pass http://client;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "Upgrade";
    }

    location /api {
        rewrite /api/(.*) /$1 break;
        proxy_pass http://api;
    }
}
```

Explanation:

- Upstream Sections:
  - upstream client: Defines a group of servers for the React frontend running on port 5173.
  - upstream api: Defines a group of servers for the Express backend running on port 8000.
- Server Block:
  - listen 80: Configures Nginx to listen on port 80 (standard HTTP port).
  - location /: Routes root URL requests to the React client.
  - location /sockjs-node: Special handling for WebSocket connections, typically used for hot-reloading in development.
  - location /api: Routes /api requests to the Express backend, rewriting the URL to strip the /api prefix.
- Step 3: Create the Dockerfile.dev for Nginx The Dockerfile.dev will build a Docker image for Nginx that uses your custom configuration. The Dockerfile content is:

```
FROM nginx
COPY ./default.conf /etc/nginx/conf.d/default.conf
```

Step-by-Step Guide for Configuring Nginx with Docker

This guide walks you through setting up Nginx as a reverse proxy to route traffic between a React frontend (client) and an Express backend (API), using Docker. The configuration involves two files: `default.conf` (Nginx configuration) and `Dockerfile.dev` (Docker configuration for Nginx).

Step 1: Understand the Project Structure

From the provided image, the structure looks like this:

```
nginx/
├── default.conf          # Nginx configuration file
└── Dockerfile.dev        # Dockerfile for building the Nginx image
```

Step 2: Configure default.conf for Nginx

The `default.conf` file is an Nginx configuration file that routes traffic based on the request paths. Here's what the configuration does:

```
upstream client {
    server client:5173;
}

upstream api {
    server api:8000;
}

server {
    listen 80;

    location / {
        proxy_pass http://client;
    }

    location /sockjs-node {
        proxy_pass http://client;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "Upgrade";
    }

    location /api {
        rewrite /api/(.*) /$1 break;
        proxy_pass http://api;
    }
}
```

Explanation:

- **Upstream Sections**:
  - `upstream client`: Defines a group of servers for the React frontend running on port 5173.
  - `upstream api`: Defines a group of servers for the Express backend running on port 8000.
- **Server Block**:
  - **listen 80**: Configures Nginx to listen on port 80 (standard HTTP port).
  - **location /**: Routes root URL requests to the React client.
  - **location /sockjs-node**: Special handling for WebSocket connections, typically used for hot-reloading in development.
  - **location /api**: Routes `/api` requests to the Express backend, rewriting the URL to strip the `/api` prefix.

Step 3: Create the Dockerfile.dev for Nginx

The `Dockerfile.dev` will build a Docker image for Nginx that uses your custom configuration. The Dockerfile content is:

```
FROM nginx
COPY ./default.conf /etc/nginx/conf.d/default.conf
```

Explanation:

- **FROM nginx**: This uses the official Nginx image as the base image.

- **COPY ./default.conf /etc/nginx/conf.d/default.conf**: Copies your custom Nginx configuration file (default.conf) into the appropriate directory inside the Nginx container (/etc/nginx/conf.d/default.conf), replacing the default configuration.

Combined Project Structure with Docker Compose

Here's how to combine your client, server, and nginx components into a single project structure with a docker-compose.yml file that orchestrates the services. This structure is typical for a full-stack application setup using Docker Compose.

Project Structure

```
my-app/
├── client/                  # React frontend
│   ├── Dockerfile.dev       # Dockerfile for the React app
(development)
│   ├── src/                 # Source code for React
│   ├── public/              # Public assets for React
│   ├── package.json         # Package configuration for React
│   └── ...                  # Other React files and folders
├── server/                  # Express backend
│   ├── Dockerfile.dev       # Dockerfile for the Express app
(development)
│   ├── index.js             # Main entry point for the Express server
│   ├── keys.js              # Environment configuration file
│   ├── package.json         # Package configuration for Express
│   └── ...                  # Other server files and folders
├── nginx/                   # Nginx reverse proxy
│   ├── default.conf         # Nginx configuration file
│   └── Dockerfile.dev       # Dockerfile for Nginx (development)
└── docker-compose.yml       # Docker Compose configuration file
```

docker-compose.yml

Below is the docker-compose.yml file that defines all the services:

```yaml
version: "3.8"
services:
  postgres:
    image: "postgres:latest"
    environment:
      - POSTGRES_PASSWORD=postgres_password
    ports:
      - "5432:5432"
    restart: always

  nginx:
    container_name: nginx
    build:
      context: ./nginx
      dockerfile: Dockerfile.dev  # Change to `Dockerfile` for
production
    ports:
      - "3500:80"
    depends_on:
      - api
      - client
    restart: always

  api:
    build:
      context: "./server"
      dockerfile: Dockerfile.dev # Change to `Dockerfile` for production
    volumes:
      - /app/node_modules
      - ./server:/app
    environment:
      - PGUSER=postgres
      - PGHOST=postgres
      - PGDATABASE=postgres
      - PGPASSWORD=postgres_password
      - PGPORT=5432
    depends_on:
      - postgres
    ports:
      - "8000:8000"
    restart: always
  client:
    stdin_open: true
    build:
      context: ./client
      dockerfile: Dockerfile.dev # Change to `Dockerfile` for production
    volumes:
      - /app/node_modules
      - ./client:/app
    ports:
      - "5173:5173"
    depends_on:
      - api
    restart: always
```

Explanation of the docker-compose.yml Services

1. **Postgres**:

- This service runs the PostgreSQL database.
- `environment`: Sets the environment variable `POSTGRES_PASSWORD` to `postgres_password` for the database.
- `ports`: Exposes port `5432` for database connections.
- `restart: always`: Ensures the container restarts automatically in case of a failure.

2. **Nginx**:

- Acts as a reverse proxy, routing requests to the `client` (React frontend) and `api` (Express backend) services.
- `build`: Builds the Nginx image from the `Dockerfile.dev` located in the `nginx` directory.
- `ports`: Exposes port `3500` on the host, mapping it to port `80` inside the container (where Nginx listens).

- `depends_on`: Ensures that the `api` and `client` services are started before Nginx.

3. **API (Express)**:

- Runs the Express server.
- `build`: Builds the Express image from the `Dockerfile.dev` located in the `server` directory.
- `volumes`: Mounts the `server` directory and `node_modules` for hot-reloading in development.
- `environment`: Passes PostgreSQL connection environment variables to the Express server.
- `depends_on`: Ensures that the PostgreSQL service is up and running before the API service starts.
- `ports`: Exposes port `8000` for the Express API.

4. **Client (React)**:

- Runs the React frontend.
- `build`: Builds the React image from the `Dockerfile.dev` located in the `client` directory.
- `volumes`: Mounts the `client` directory and `node_modules` for hot-reloading in development.
- `ports`: Exposes port `5173` for the React development server.
- `depends_on`: Ensures the API service is running before the Client starts.

Step-by-Step Setup and Execution

1. **Build and Start Services**:

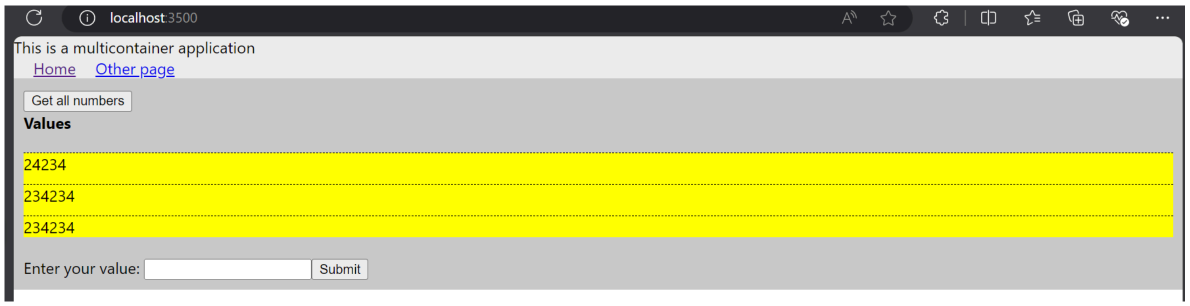
- Navigate to the root directory (`my-app/`) and run the following command to build and start all services:

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

- The `--build` flag ensures that Docker builds the images before starting the containers.

2. **Access the Application**:

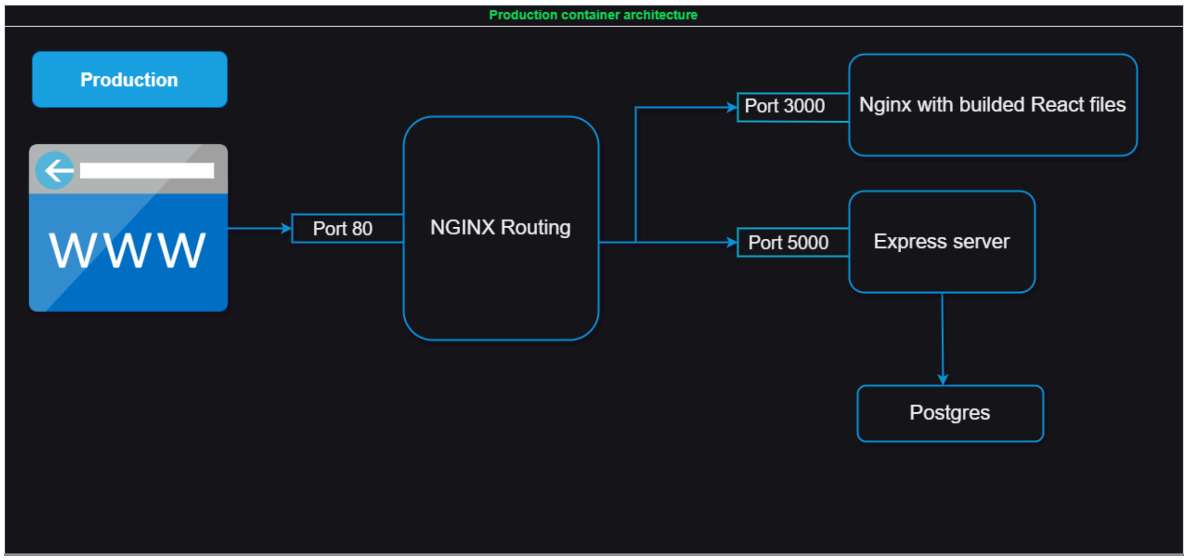
- Visit `http://localhost:3500` in your browser to access the frontend, which is proxied through Nginx.
- The React frontend interacts with the Express backend via the `/api` route, as defined in the Nginx configuration.



devwebpreview.png

Production Mode

We are going to use all files in the development mode but make a couple of changes.ie our docker files will be `Docker` instead of `Docker.dev`



prod.png

- Create Nginx folder in client folder and add `default.conf` with this code

```
server {
    listen 5173;

    location / {
        root /usr/share/nginx/html;
        index index.html index.html;
        try_files $uri/ $uri/ /index.html
    }
}
```

- inside the client folder, add a `Docker` file

```
#use alpine Node.js runtime as a parent image and name it as builder
FROM node:20.16.0-alpine as builder

# Set the working directory in the container
WORKDIR /app

# Copy package.json and package-lock.json
```

```
COPY package*.json ./

# Install dependencies
RUN npm i

# Copy the rest of the application code
COPY . .

RUN npm run build

FROM nginx

# Expose port 5173
EXPOSE 5173
# copy nginx configuration to the docker image
COPY ./nginx/default.conf /etc/nginx/conf.d/default.conf
# copy all builded files
COPY --from=builder /app/build /usr/share/nginx/html
```

Summary in bullet points:

- **Base Image (Builder Stage)**:
  - Uses `node:20.16.0-alpine` as the base image, named as `builder`.
  - Sets the working directory to `/app` inside the container.
- **Dependency Installation**:
  - Copies `package.json` and `package-lock.json` into the container.
  - Installs Node.js dependencies using `npm i`.
- **Application Code**:
  - Copies the entire application code into the container.
  - Runs the build process with `npm run build`.

- **Final Image (Nginx Stage)**:
  - Uses `nginx` as the final base image.
  - Exposes port `5173` for the frontend.
  - Copies the Nginx configuration file (`default.conf`) to the appropriate location in the Nginx container.
  - Copies the built files from the `builder` stage to the Nginx web root directory (`/usr/share/nginx/html`).

Build the Docker Image

- cd to client folder, and run

```
docker build -t client-prod .
```