.

🛒 E-Commerce App: Full Stack Setup Guide

This guide walks you through setting up a **React frontend**, **Node.js backend**, and a **local MySQL database**.

---

1. Project Structure

Create a parent folder for your app with separate frontend and backend directories:

```
# Create main project folder
mkdir my-ecommerce-app
cd my-ecommerce-app
# Create backend and frontend folders
mkdir backend
mkdir frontend
```

2. Frontend: React Application

Navigate into the frontend folder and bootstrap a React app:

```
cd frontend
npx create-react-app my-ecommerce-store
cd my-ecommerce-store
```

Install essential packages:

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

- **react-router-dom**: Handles navigation/routing
- **axios**: Makes HTTP requests to the backend

Run the development server:

```
npm start
```

You should see your React app running at `http://localhost:3000`.

3. Backend Setup (Node.js + Express + MySQL)

Navigate to the backend folder:

```
cd ../../backend
npm init -y
```

Install required packages:

```
npm install express mysql2 cors dotenv
```

- **express**: Web server framework
- **mysql2**: Connects to your MySQL DB
- **cors**: Allows frontend to communicate with backend
- **dotenv**: Manages environment variables

4. Local MySQL Database Setup

Open your local MySQL in a terminal:

```
mysql -u root -p
```

Enter your password (`Ijse@1234`) and run:

```
CREATE DATABASE ecommerce_db;
USE ecommerce_db;

CREATE TABLE products (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(255) NOT NULL,
  price DECIMAL(10, 2) NOT NULL,
  description TEXT
);

INSERT INTO products (name, price, description) VALUES
('T-Shirt', 25.00, 'A comfortable cotton t-shirt.'),
('Jeans', 50.00, 'Classic blue jeans for everyday wear.'),
('Sneakers', 80.00, 'Stylish sneakers for a casual look.');
```

5. Backend Code (server.js)

Create a file `server.js` inside the `backend` folder and paste:

Run your backend:

```
node server.js
```

6. Frontend: Fetch Products with Axios

In your React app (inside `src/Products.js` for example):

```
import axios from 'axios';
import { useEffect, useState } from 'react';

function Products() {
  const [products, setProducts] = useState([]);

  useEffect(() => {
    axios.get('http://localhost:5000/api/products')
      .then(res => setProducts(res.data))
      .catch(err => console.error(err));
  }, []);

  return (
    <div>
      {products.map(p => (
        <div key={p.id}>
          <h3>{p.name}</h3>
          <p>{p.description}</p>
          <strong>${p.price}</strong>
        </div>
      ))}
    </div>
  );
}

export default Products;
```

Import `<Products />` somewhere in your React app and you'll see data from MySQL.

7. Verify Database from Terminal

Check your data anytime:

```
mysql -u root -p
USE ecommerce_db;
SHOW TABLES;
```

```
SELECT * FROM products;
```

📝 Special Note: Prisma (Optional ORM Layer)

Right now your backend uses `mysql2` directly and writes raw SQL queries.
That's totally fine for small projects, but as your app grows, writing and maintaining raw SQL becomes harder.

Prisma is an **ORM** (Object–Relational Mapper) that gives you:

- **Auto-generated database client** – you write JavaScript/TypeScript instead of SQL.
- **Type safety & autocompletion** in VS Code.
- **Migrations** – Prisma can create/update your DB schema automatically.
- **Clean & maintainable code** – easier to read than SQL strings.

🔹 How Prisma Fits in Your App

Instead of:

```
pool.query('SELECT * FROM products', ...)
```

You'd do:

```
const { PrismaClient } = require('@prisma/client')
const prisma = new PrismaClient()
const products = await prisma.product.findMany()
```

Prisma automatically maps your `Product` model to the `products` table in the database.

🔹 Setting up Prisma in Your Backend

1. **Install Prisma CLI and Client**

From your `backend` folder:

```
npm install prisma --save-dev
npm install @prisma/client
```

2. **Initialize Prisma**

```
npx prisma init
```

This creates:

- `.env` (holds your DB connection string)
- `prisma/schema.prisma` (define your models here)

3. **Update `.env` for your local MySQL**

```
DATABASE_URL="mysql://root:Ijse@1234@localhost:3306/ecommerce_db"
```

4. **Edit `prisma/schema.prisma`**

Replace the default content with something like:

```
datasource db {
  provider = "mysql"
  url      = env("DATABASE_URL")
}

generator client {
  provider = "prisma-client-js"
}

model Product {
  id          Int      @id @default(autoincrement())
  name        String
  price       Float
  description String?
}
```

5. **Run Migrations**

```
npx prisma migrate dev --name init
```

This will:

- Create the `products` table if it doesn't exist.
- Generate the Prisma client code.

6. **Generate Client (if you change the schema)**

```
npx prisma generate
```

7. **Use Prisma in Your Server**

Replace your SQL query with Prisma:

```
const { PrismaClient } = require('@prisma/client');
const prisma = new PrismaClient();

// Example API route
app.get('/api/products', async (req, res) => {
  try {
    const products = await prisma.product.findMany();
    res.json(products);
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Failed to fetch products' });
  }
});
```

Now you're fully using Prisma instead of `mysql2`.

🔹 Bottom Line

- You **don't have to use Prisma**; your current code works fine.
- But Prisma gives you a cleaner API, migrations, type safety, and a built-in GUI.
- It's especially helpful if your project grows beyond a few tables or if you're using TypeScript.

📝 End of the Special Note

🐳 8. Containerizing Your App with Docker

So far you have a React frontend, Node.js backend, and local MySQL running directly on your machine.

Now you can containerize the frontend and backend using **Docker** and orchestrate them with **Docker Compose**.

8.1 Backend Dockerfile (Node.js API)

```
# Use a Node.js base image
FROM node:18-alpine

# Set the working directory
WORKDIR /app
```

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

# Install dependencies
RUN npm install

# Copy the rest of your application code
COPY . .

# Expose the port your app runs on
EXPOSE 5000

# Start the application
CMD ["node", "server.js"]
```

Explanation:

- Uses a lightweight Node.js base image.
- Copies only `package*.json` first for efficient caching.
- Installs dependencies, then copies your code.
- Exposes port **5000** for the backend API.
- Starts the app with `node server.js`.

This Dockerfile lives inside your **`backend/`** folder.

8.2 Frontend Dockerfile (React Build)

```
# Build stage
FROM node:18-alpine as builder

WORKDIR /app

COPY package*.json ./

RUN npm install

COPY . .

RUN npm run build
```

```
# Server stage
FROM node:18-alpine

WORKDIR /app

# Copy the build output from the builder stage
COPY --from=builder /app/build ./build

# Install a simple web server (e.g., serve)
RUN npm install -g serve

# Expose the port for the frontend
EXPOSE 3000

# Command to serve the static files
CMD ["serve", "-s", "build"]
```

Explanation:

- Uses a **multi-stage build**: first builds your React app, then copies the optimized `build` folder to a clean image.
- Installs the `serve` package globally to serve the built static files.
- Exposes port **3000** for the frontend.

This Dockerfile lives inside your **`frontend/my-ecommerce-store/`** folder.

8.3 Docker Compose File

```
version: '3.8'

services:
 backend:
   build: ./backend
   ports:
     - "5000:5000"
   environment:
     # These values must match the credentials in your server.js
     DB_HOST: host.docker.internal
     DB_USER: root
     DB_PASSWORD: Ijse@1234  # <-- Corrected password
     DB_NAME: ecommerce_db
```

```
    DB_PORT: 3306
  volumes:
    - ./backend:/app

frontend:
  build: ./frontend/my-ecommerce-store
  ports:
    - "3000:3000"
  volumes:
    - ./frontend:/app
  depends_on:
    - backend
```

Explanation:

- `backend` service:
  - Builds from the backend Dockerfile.
  - Exposes port **5000**.
  - Passes DB credentials as environment variables.
    - `host.docker.internal` is used so the container can connect to your local MySQL.
  - Mounts your backend folder as a volume for live code changes.

- `frontend` service:
  - Builds from the frontend Dockerfile.
  - Exposes port **3000**.
  - Mounts your frontend folder as a volume.
  - Depends on the backend so Docker starts the backend first.

8.4 Running the Containers

From the **root** of your project (where `docker-compose.yml` lives):

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

You'll see:

- Backend API at `http://localhost:5000/api/products`.
- Frontend React app at `http://localhost:3000`.

💡 **Tip:**
You can speed this up:

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

This rebuilds and starts everything in one go.

8.5 How the Stack Works Now

- **MySQL** still runs locally on your machine (not inside Docker).
- **Backend** container connects to `host.docker.internal` which resolves to the host machine's IP inside Docker.
- **Frontend** container calls the backend via `http://localhost:5000/api/products`.

💻 After a Restart (your scenario)

When you shut down and later boot up your computer:

You **don't** run `npm start` or `node server.js` manually anymore — Docker runs those commands inside the containers for you (because of the `CMD` you wrote in each Dockerfile).

1. Docker Desktop or the Docker service starts (make sure it's running).
2. In your project root (where `docker-compose.yml` lives), run:

```
docker-compose up
```

That's it ✅
It will start the **backend** and **frontend** containers with the commands specified in their Dockerfiles. You don't have to run `npm start` or `node server.js` yourself.

1️⃣ When you run in foreground

If you run:

```
docker-compose up
```

you'll see all the container logs in your terminal.

- To stop them: press **`Ctrl + C`** in that terminal.
- Docker Compose will then gracefully stop all the containers it started.

2️⃣ When you run in detached mode

If you run:

```
docker-compose up -d
```

(the `-d` = "detached"), your containers keep running in the background even if you close the terminal.

- To stop them later, in any terminal in the same project folder run:

```
docker-compose down
```

This stops and removes the containers that `docker-compose up` created.

3️⃣ When to rebuild

You only need `--build` again if:

- You changed a Dockerfile,
- You changed the package.json (new deps),
- Or you changed any file that's part of the image build context that would affect the build stage.