**SENG8070 – Final Project Report : Customs Freight Database RESTful API**

**Name:** Sheetal Rana

**Course:** SENG8070

**Submission Date:** August 13, 2025

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Introduction**

This project extends the *Customs Freight Company* database from the midterm into a fully functional RESTful API built with TypeScript, Express.js, and TypeORM on top of PostgreSQL. The API provides CRUD operations for the core entities — *Vehicles, Employees, Repairs, Shipments,* and *Trips* — enabling efficient and reliable management of customs freight operations data. The solution is production‑ready, including validation, error handling, unit and integration testing, Docker containerization, and TypeORM migrations to seed initial data.

**Project Objectives**

* Leverage the midterm PostgreSQL schema for persistence.
* Implement complete CRUD operations for:
* Vehicles
* Employees
* Shipments
* Repairs
* Trips
* Ensure quality through Jest-based testing (unit and integration).
* Containerize the system with Docker for consistent deployment.
* Provide comprehensive documentation and setup instructions.

**Technologies Used**

* **Language:** TypeScript
* **Framework:** Express.js
* **ORM:** TypeORM
* **Database:** PostgreSQL
* **Testing:** Jest + Supertest
* **Containerization:** Docker & Docker Compose
* **Version Control:** Git

**Database Overview**

We used the *Customs Freight Company* database schema from the midterm with the following tables:

**TABLE: Vehicle -** brand, load, capacity, year, numberOfRepairs

CREATE TABLE Vehicle (

VehicleID SERIAL PRIMARY KEY,

VehicleType VARCHAR(20) CHECK (VehicleType IN ('Cargo Plane', 'In-City Truck', 'Long Haul Truck')),

Brand VARCHAR(50) NOT NULL,

LoadCapacityKg INT NOT NULL CHECK (LoadCapacityKg > 0),

Year INT NOT NULL CHECK (Year BETWEEN 2000 AND EXTRACT(YEAR FROM NOW())),

NumRepairs INT DEFAULT 0 CHECK (NumRepairs >= 0)

);

**TABLE: Employee** - name, surname, seniority, vehicleTypeCertifications, mechanicCertification

CREATE TABLE Employee (

EmployeeID SERIAL PRIMARY KEY,

FirstName VARCHAR(50) NOT NULL,

LastName VARCHAR(50) NOT NULL,

SeniorityYears INT NOT NULL CHECK (SeniorityYears >= 0),

IsMechanic BOOLEAN NOT NULL

);

**TABLE: Shipment** - Customer info (name, address, phone1, phone2), shipment details (weight, value, origin, destination)

CREATE TABLE Shipment (

ShipmentID SERIAL PRIMARY KEY,

CustomerID INT REFERENCES Customer(CustomerID),

WeightKg INT NOT NULL CHECK (WeightKg > 0),

Value NUMERIC(10,2) NOT NULL CHECK (Value >= 0),

Origin VARCHAR(100) NOT NULL,

Destination VARCHAR(100) NOT NULL );

**TABLE: Repair -** Tracks vehicleID, mechanicID, estimatedRepairTime, actualRepairTime

CREATE TABLE Repair (

RepairID SERIAL PRIMARY KEY,

VehicleID INT NOT NULL REFERENCES Vehicle(VehicleID),

MechanicID INT NOT NULL REFERENCES Employee(EmployeeID),

EstimatedDays INT NOT NULL CHECK (EstimatedDays > 0),

ActualDays INT NOT NULL CHECK (ActualDays > 0),

RepairCost NUMERIC(10,2) CHECK (RepairCost >= 0),

StartDate DATE DEFAULT CURRENT\_DATE

);

**TABLE: Trip** – from, to, drivers (up to two), related shipments

CREATE TABLE Trip (

TripID SERIAL PRIMARY KEY,

RouteFrom VARCHAR(100) NOT NULL,

RouteTo VARCHAR(100) NOT NULL

);

**How Code Works**

**1. Entry Point**

* **src/server.ts**
  + Starts the Express server.
  + Connects to the PostgreSQL database via TypeORM.
  + Loads routes for all entities (Vehicles, Employees, Shipments, Repairs, Trips).

**2. Application Setup**

* **src/app.ts**
  + Initializes the Express app.
  + Sets up middleware for:
    - JSON parsing
    - Error handling
    - Validation
  + Registers routes from src/routes/.

**3. Database Connection**

* **src/data-source.ts**
  + Configures TypeORM connection using .env variables:
    - Host, port, username, password, database name.
  + Loads **entities** from src/entities/.

**4. Entities (Database Models)**

* Located in **src/entities/**
* Each entity maps to a table in PostgreSQL:
  + **Vehicle.ts**
  + **Employee.ts**
  + **Shipment.ts**
  + **Repair.ts**
  + **Trip.ts**
  + Additional join tables for many-to-many relationships (e.g., TripDriver, TripShipment).

**5. Controllers**

* Located in **src/controllers/**
* Each controller handles incoming HTTP requests and calls the appropriate service functions:
  + **VehicleController.ts**
  + **EmployeeController.ts**
  + **ShipmentController.ts**
  + **RepairController.ts**
  + **TripController.ts**

Example:

async create(req: Request, res: Response) {

const newVehicle = await vehicleService.create(req.body);

res.status(201).json(newVehicle);

}

**6. Services**

* Located in **src/services/**
* Contain **business logic** for CRUD operations.
* Inherit from a **BaseService.ts** that provides generic create, read, update, delete methods.

**7. Routes**

* Located in **src/routes/**
* Map HTTP endpoints to controller functions.  
  Example from vehicleRoutes.ts:

router.get('/', VehicleController.getAll);

router.post('/', VehicleController.create);

router.get('/:id', VehicleController.getById);

router.put('/:id', VehicleController.update);

router.delete('/:id', VehicleController.delete);

**8. Testing**

* **tests/unit/** – Unit tests for service layer logic.
* **tests/integration/** – Integration tests that send HTTP requests to the API.

**9. How the Flow Works**

1. **Request** (via Postman/curl) → hits a route.
2. **Route** calls a controller function.
3. **Controller** calls a service function.
4. **Service** talks to TypeORM repository → runs SQL queries.
5. **Response** returned to the client.

**Method : Run with Docker**

This is the easiest and ensures your environment matches everyone else’s.

**1️⃣ Install Prerequisites**

* Docker Desktop
* Node.js v18+

**2️⃣ Unzip**

seng8071-final

**3️⃣ Create the .env File**

Write the Name of your database saved on PgAdmin as well as password

Create a .env file in the root folder:

DB\_HOST=postgres

DB\_PORT=5432

DB\_USERNAME=postgres

DB\_PASSWORD=postgres

DB\_NAME=customs\_freight

PORT=3000

**And data-source.ts**

import "reflect-metadata";

import { DataSource } from "typeorm";

import \* as dotenv from "dotenv";

import path from "path";

dotenv.config();

export const AppDataSource = new DataSource({

  type: "postgres",

  host: process.env.DB\_HOST || "localhost",

  port: parseInt(process.env.DB\_PORT || "5432"),

  username: process.env.DB\_USER || "postgres",

  password: process.env.DB\_PASS || "postgres",

  database: process.env.DB\_NAME || "Project",

  synchronize: false,

  logging: false,

  entities: [path.join(\_\_dirname, "entities/\*.{ts,js}")]

});

**4️⃣ Start the Application**

docker-compose up –build

A screenshot of a computer

AI-generated content may be incorrect.

This will:

* Launch **PostgreSQL** in a container.
* Build and run the **Node.js REST API**.

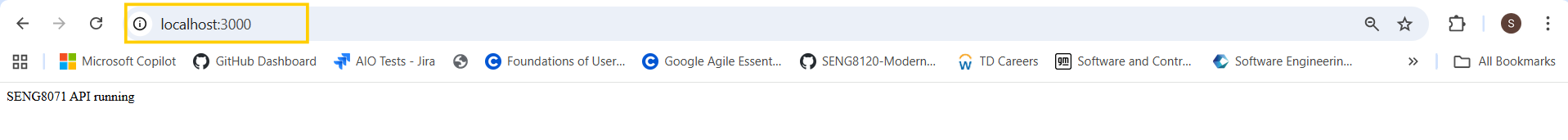
A screenshot of a computer program

AI-generated content may be incorrect.

**5️⃣ Verify the API is Running**

* Open <http://localhost:3000/vehicles> in your browser or Postman.
* You should see JSON output for the vehicles table.

Start API

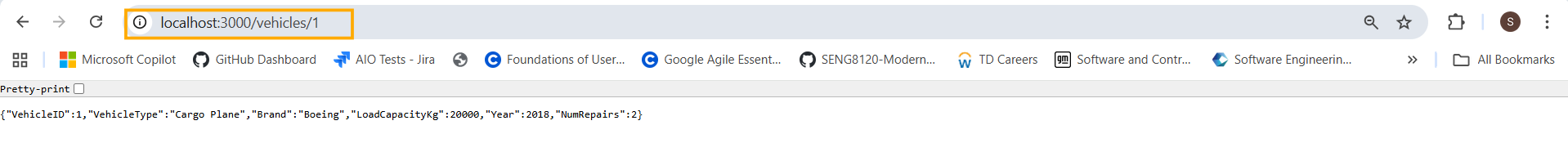


Find all Vehicles

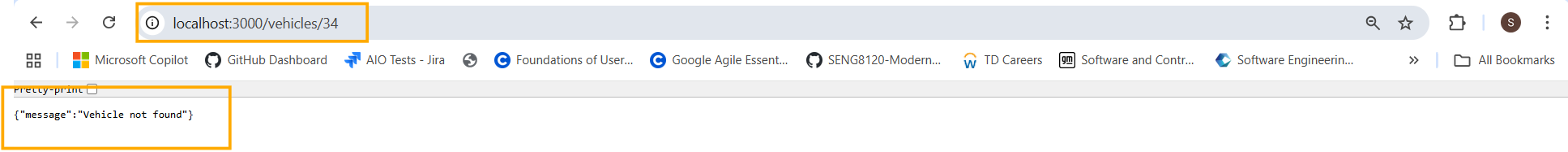
A screen shot of a computer

AI-generated content may be incorrect.

Vehicles by ID



Vehicle ID not found – with error message “Vehicle not found” when ID is not present in the data table

****

**Testing the API**

Test the endpoints using:

* **Postman**: Import your endpoints and run CRUD operations.

A screenshot of a computer

AI-generated content may be incorrect.

* **WSL curl**:

curl -X GET http://localhost:3000/vehicles

* **Jest**:

npm run test

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

**WSL Terminal with curl**

This section demonstrates the use of curl commands within the WSL (Windows Subsystem for Linux) terminal to interact with the RESTful API endpoints developed for the Customs Freight Database System. Each operation was executed against the backend services for core entities such as Vehicles, Employees, Shipments, Repairs, and Trips.

**Create** – Used to add new records to the database (e.g., registering a new vehicle or employee).  
*Purpose:* To test POST endpoints and validate data insertion.

A computer screen shot of a computer program

AI-generated content may be incorrect.

**Read All** – Retrieves all records from a specific entity (e.g., listing all shipments).  
*Purpose:* To verify GET endpoints and ensure data retrieval works correctly.

A screenshot of a computer program

AI-generated content may be incorrect.

**View Single Item by ID** – Fetches a specific record using its unique identifier.  
*Purpose:* To confirm that individual record access via GET with parameters is functioning.

A screen shot of a computer program

AI-generated content may be incorrect.

**Update** – Modifies existing records (e.g., updating repair details or employee seniority).  
*Purpose:* To test PUT endpoints and validate update logic and constraints.

A screenshot of a computer program

AI-generated content may be incorrect.

**Delete** – Removes records from the database (e.g., deleting a trip or vehicle entry).  
*Purpose:* To ensure DELETE endpoints are correctly implemented and data is removed safely.

A screen shot of a computer

AI-generated content may be incorrect.

**RESTful API Implementation**

**Endpoints**

Example for Vehicles:

* GET /vehicles – list all
* GET /vehicles/:id – fetch one
* POST /vehicles – create new
* PUT /vehicles/:id – update existing
* DELETE /vehicles/:id – remove

Similar patterns for Employees, Repairs, Shipments, and Trips.

**CRUD Operations tested using Postman**

**TABLE: Vehicles**

**GET all**

**A screenshot of a computer

AI-generated content may be incorrect.**

**GET by ID**

**A screenshot of a computer

AI-generated content may be incorrect.**

**POST**

**A screenshot of a computer

AI-generated content may be incorrect.**

**PUT**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**DELETE**

**A screenshot of a computer

AI-generated content may be incorrect.**

A screenshot of a computer

AI-generated content may be incorrect.

**TABLE: Employee**

**GET all**

**A screenshot of a computer

AI-generated content may be incorrect.**

**GET by ID**

**A screenshot of a computer

AI-generated content may be incorrect.**

**POST**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**PUT**

**A screenshot of a computer

AI-generated content may be incorrect.**

**DELETE**

**A screenshot of a computer

AI-generated content may be incorrect.**

**TABLE: Shipment**

**GET all**

**Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.**

**GET by ID**

**Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.**

**POST**

**Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.**

**PUT**

**Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.**

**DELETE**

**Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.**

**TABLE: Repair**

**GET all**

**Imagen que contiene Calendario

El contenido generado por IA puede ser incorrecto.**

**GET by ID**

**Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.**

**POST**

**Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.**

**PUT**

**Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.**

**DELETE**

**Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.**

**TABLE: Trip**

**GET all**

**Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.**

**GET by ID**

**Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.**

**POST**

**Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.**

**PUT**

**Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.**

**DELETE**

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

**Validation & Error Handling**

* class-validator used for request body checks
* Centralized middleware for consistent JSON error responses

**UNIT TESTS**

Unit testing was conducted using Jest in a TypeScript environment, with each service module tested in isolation to ensure correctness and reliability of business logic. The image shows the successful execution of unit tests for the following service classes:

* EmployeeService
* ShipmentService
* RepairService
* VehicleService
* TripService

Each service was tested for core operations such as findAll and create, verifying that:

* Data retrieval functions return expected results.
* Creation logic correctly inserts new records.
* Services handle valid and invalid inputs gracefully.
* All constraints and validation rules are respected.

**Negative test:** Test that findById throws an error when the entity is not found (like a 404 scenario at service level).

A screenshot of a computer

AI-generated content may be incorrect.

**INTEGRATION TESTS**

Integration testing was performed to validate the complete request-response cycle of the RESTful API, ensuring that each endpoint correctly interacts with the database and returns expected results. The tests were executed using Jest and Supertest, targeting all major entities: Vehicles, Shipments, Employees, Repairs, and Trips.

Each entity was tested for the following operations:

* POST – Verifies that new records can be successfully created.
* GET (list) – Ensures that all records are retrieved correctly.
* GET (by ID) – Confirms that individual records can be accessed using their unique identifiers.

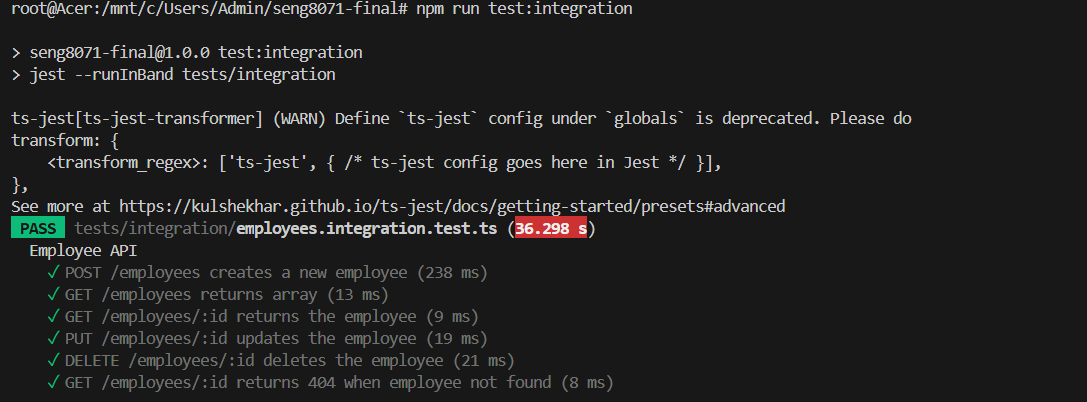
Test Summary from Image:

* Test Suites: 5 passed out of 5 total
* Tests Executed: 15 passed out of 15 total
* Execution Time: ~42.2 seconds
* Snapshot Tests: None used
* Entities Covered: Vehicles, Shipments, Employees, Repairs, Trips

Each test suite confirmed that the API endpoints are functioning correctly, with accurate data persistence and retrieval. The results also show that the system handles requests efficiently, with response times ranging from 3ms to 301ms.

This level of integration testing ensures that the backend is production-ready and capable of handling real-world data operations reliably.

A **negative test** is designed to check how your system behaves with **invalid inputs, non-existent resources, or failure scenarios**.



A screenshot of a computer program

AI-generated content may be incorrect.

**README.md**

The README.md file provides comprehensive documentation for users and developers:

* Project Overview – Purpose, features, and scope.
* Prerequisites – Required tools (Node.js, Docker).
* Installation Instructions – Setup steps, environment configuration.
* Usage – How to run the app and execute tests.
* API Reference – List of endpoints with examples.
* ER Diagram – Visual representation of database schema.

This documentation ensures smooth onboarding and effective usage of the system.

**Conclusion & Future Work**

The Customs Freight RESTful API delivers a robust, scalable, and maintainable backend solution, meeting all requirements for the SENG8071 final project.

Achievements:

* Full CRUD support for all entities.
* Comprehensive testing and validation.
* Containerized deployment for portability.
* Clean architecture and modular design.

Future Enhancements:

* Authentication & Role-Based Access Control – Secure access and user permissions.
* Pagination & Filtering – Improve performance and usability for large datasets.
* Hardware Integration – Connect with RFID/GPS for real-time tracking and logistics automation.
* Monitoring & Logging – Add observability tools for production readiness.
* Frontend Interface – Build a user-friendly UI for interacting with the API.