Lab 5: Implementing the Vehicle/Catalog Microservice

Bus Ticket Management System
System Name: - Vehicle/Catalog Microservice

Introduction

This lab focuses on the practical implementation of one of the core services identified in Lab 4: the **Vehicle/Catalog Service**. This service will be completely independent, owning its data (buses, companies, routes, stations) and exposing a RESTful API, adhering to the principles of Microservices Architecture.

Unlike the monolithic approach in Lab 3, this microservice:

- **Runs independently** on its own port
- **Owns its database tables** (buses, bus_companies, stations, routes)
- **Exposes well-defined API contracts** for other services to consume
- **Can be deployed separately** without affecting other services

Objectives

1. **Set up a standalone Express/TypeScript application** dedicated solely to vehicle/catalog management
2. **Implement the Catalog Service logic and persistence** using MySQL database
3. **Expose the defined Service Contract** (REST API) for reading and searching vehicles
4. **Test the service in isolation** using Postman or curl
5. **Demonstrate microservice independence** by running it separately from the main application

Technology & Tool Installation

We will use the same technology stack as the main application but structure it as an independent microservice.

Tool	Purpose	Installation/Setup Guide
Node.js 18+	JavaScript runtime environment	Ensure Node.js is installed: node --version
TypeScript	Static typing for JavaScript	Installed as dev dependency: npm install -D typescript
Express.js	Web framework for RESTful API	npm install express
MySQL	Relational database	Use existing MySQL server

Knex.js	SQL query builder and migration tool	npm install knex mysql2
Postman / curl	API testing tool	Install Postman or use built-in curl command

---

Activity Practice 1: Project Setup and Data Modeling

Goal

Create the microservice project structure and define the Vehicle database schema using TypeScript and Knex.js.

Step-by-Step Instructions & Coding Guide

1. Create Service Directory

```
# Navigate to your project root
cd "d:\Flance\Data\Customer12 (DatXeKhach ReactjsNodejs)\Duandatxekhach\Duandatxekhach"

# Create microservices directory
mkdir microservices
cd microservices

# Create vehicle-service directory
mkdir vehicle-service
cd vehicle-service

# Initialize Node.js project
npm init -y
```

2. Install Dependencies

```
# Install production dependencies
npm install express dotenv cors mysql2 knex

# Install TypeScript and development dependencies
npm install -D typescript @types/node @types/express @types/cors ts-node nodemon
```

3. Configure TypeScript

Create tsconfig.json:

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "lib": ["ES2020"],
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "resolveJsonModule": true,
    "moduleResolution": "node",
    "baseUrl": "./src",
    "paths": {
      "@/*": ["*"]
    }
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "dist"]
}
```

4. Create Project Structure

```
# Create directory structure
mkdir src
mkdir src\models
mkdir src\controllers
mkdir src\services
mkdir src\repositories
mkdir src\routes
mkdir src\config
```

5. Define the Vehicle Model

File: src/models/Vehicle.ts

```
// Vehicle entity representing a bus in the database
export interface Vehicle {
  id: number;
  name: string;
  description?: string;
  license_plate: string;
  capacity: number;
  company_id: number;
  company_name?: string; // Joined from bus_companies table
  featured_image?: string;
  is_active: boolean;
  created_at: Date;
  updated_at: Date;
}

// DTO for creating a new vehicle
export interface CreateVehicleDto {
  name: string;
  description?: string;
  license_plate: string;
  capacity: number;
  company_id: number;
  featured_image?: string;
}

// DTO for searching vehicles
export interface VehicleSearchParams {
  company_id?: number;
  min_capacity?: number;
  page?: number;
  limit?: number;
  search?: string;
}

// Standardized API response format
export interface ApiResponse<T> {
  success: boolean;
  message: string;
  data: T | null;
  pagination?: {
    page: number;
    limit: number;
    total: number;
    totalPages: number;
  };
}
```

6. Configure Database Connection

File: src/config/database.ts

```
import knex, { Knex } from 'knex';

// Database configuration for the Vehicle Service
const dbConfig: Knex.Config = {
  client: 'mysql2',
  connection: {
    host: process.env.DB_HOST || 'localhost',
    port: parseInt(process.env.DB_PORT || '3306'),
    user: process.env.DB_USER || 'root',
    password: process.env.DB_PASSWORD || '',
    database: process.env.DB_NAME || 'bus_booking'
  },
  pool: {
    min: 2,
    max: 10
  },
  migrations: {
    tableName: 'knex_migrations'
  }
};

// Create and export the database connection
export const db: Knex = knex(dbConfig);

// Test database connection
export async function testConnection(): Promise<boolean> {
  try {
    await db.raw('SELECT 1');
    console.log(' Database connection successful');
    return true;
  } catch (error) {
    console.error(' Database connection failed:', error);
    return false;
  }
}
```

7. Create Environment Configuration

File: .env

```
# Server Configuration
PORT=5001
NODE_ENV=development

# Database Configuration
DB_HOST=localhost
DB_PORT=3306
DB_USER=root
DB_PASSWORD=your_password_here
DB_NAME=bus_booking
```

```
# Service Information
SERVICE_NAME=Vehicle-Catalog-Service
SERVICE_VERSION=1.0.0
```

File: .env.example

```
PORT=5001
NODE_ENV=development
DB_HOST=localhost
DB_PORT=3306
DB_USER=root
DB_PASSWORD=
DB_NAME=bus_booking
SERVICE_NAME=Vehicle-Catalog-Service
SERVICE_VERSION=1.0.0
```

Activity Practice 2: Implementing the Service API

Goal

Implement the REST API endpoints to read vehicle data, fulfilling the service contract defined in Lab 4.

Step-by-Step Instructions & Coding Guide

1. Implement the Repository Layer (Data Access)

File: src/repositories/VehicleRepository.ts

```
import { db } from '@/config/database';
import { Vehicle, VehicleSearchParams, CreateVehicleDto } from '@/models/Vehicle';

export class VehicleRepository {
  private tableName = 'cars'; // Main table for vehicles

  /**
   * Find all vehicles with optional filtering and pagination
   */
  async findAll(params: VehicleSearchParams): Promise<{ vehicles: Vehicle[]; total: number }>
{
```

```
const { company_id, min_capacity, page = 1, limit = 10, search } = params;
const offset = (page - 1) * limit;

// Build the query with joins
let query = db('cars as c')
  .leftJoin('bus_companies as bc', 'c.company_id', 'bc.id')
  .select(
    'c.id',
    'c.name',
    'c.description',
    'c.license_plate',
    'c.capacity',
    'c.company_id',
    'bc.company_name',
    'c.featured_image',
    'c.created_at',
    'c.updated_at'
  );

// Apply filters
if (company_id) {
  query = query.where('c.company_id', company_id);
}

if (min_capacity) {
  query = query.where('c.capacity', '>=', min_capacity);
}

if (search) {
  query = query.where(function() {
    this.where('c.name', 'like', `%${search}%`)
      .orWhere('c.license_plate', 'like', `%${search}%`);
    });
  }

  // Get total count
  const countQuery = query.clone();
  const [{ total }] = await countQuery.count('c.id as total');

  // Get paginated results
  const vehicles = await query
    .limit(limit)
    .offset(offset)
    .orderBy('c.created_at', 'desc');

  return {
    vehicles: vehicles as Vehicle[],
    total: total as number
  };
}

/**
 * Find a single vehicle by ID
 */
async findById(id: number): Promise<Vehicle | null> {
  const vehicle = await db('cars as c')
    .leftJoin('bus_companies as bc', 'c.company_id', 'bc.id')
    .select(
      'c.id',
      'c.name',
      'c.description',
      'c.license_plate',
      'c.capacity',
      'c.company_id',
        'bc.company_name',
        'c.featured_image',
        'c.created_at',
        'c.updated_at'
      )
      .where('c.id', id)
      .first();

    return vehicle || null;
  }

  /**
   * Create a new vehicle (for admin use)
   */
  async create(data: CreateVehicleDto): Promise<Vehicle> {
    const [id] = await db('cars').insert({
      ...data,
      created_at: new Date(),
      updated_at: new Date()
    });

    const newVehicle = await this.findById(id);
    if (!newVehicle) {
      throw new Error('Failed to create vehicle');
    }

    return newVehicle;
  }

  /**
   * Check if license plate already exists
   */
  async existsByLicensePlate(licensePlate: string): Promise<boolean> {
    const vehicle = await db('cars')
      .where('license_plate', licensePlate)
      .first();

    return !!vehicle;
  }
}
```

2. Implement the Service Layer (Business Logic)

File: src/services/VehicleService.ts

```
import { VehicleRepository } from '@/repositories/VehicleRepository';
import { Vehicle, VehicleSearchParams, CreateVehicleDto, ApiResponse } from
'@/models/Vehicle';

export class VehicleService {
  private repository: VehicleRepository;

  constructor() {
    this.repository = new VehicleRepository();
  }

  /**
   * Get all vehicles with pagination and filtering
   */
  async getVehicles(params: VehicleSearchParams): Promise<ApiResponse<Vehicle[]>> {
    try {
      const { vehicles, total } = await this.repository.findAll(params);
      const page = params.page || 1;
      const limit = params.limit || 10;
      const totalPages = Math.ceil(total / limit);

      return {
```

```
      success: true,
      message: 'Vehicles retrieved successfully',
      data: vehicles,
      pagination: {
        page,
        limit,
        total,
        totalPages
      }
    };
  } catch (error) {
    console.error('Error in getVehicles:', error);
    return {
      success: false,
      message: 'Failed to retrieve vehicles',
      data: null
    };
  }
}

/**
 * Get a single vehicle by ID
 */
async getVehicleById(id: number): Promise<ApiResponse<Vehicle>> {
  try {
    const vehicle = await this.repository.findById(id);

    if (!vehicle) {
      return {
        success: false,
        message: 'Vehicle not found',
        data: null
      };
    }

    return {
      success: true,
      message: 'Vehicle found',
      data: vehicle
    };
  } catch (error) {
    console.error('Error in getVehicleById:', error);
    return {
      success: false,
      message: 'Failed to retrieve vehicle',
      data: null
    };
  }
}

/**
 * Create a new vehicle
 */
async createVehicle(data: CreateVehicleDto): Promise<ApiResponse<Vehicle>> {
  try {
    // Check if license plate already exists
    const exists = await this.repository.existsByLicensePlate(data.license_plate);
    if (exists) {
      return {
        success: false,
        message: `Vehicle with license plate ${data.license_plate} already exists`,
        data: null
      };
    }
```

```
      const newVehicle = await this.repository.create(data);

      return {
        success: true,
        message: 'Vehicle created successfully',
        data: newVehicle
      };
    } catch (error) {
      console.error('Error in createVehicle:', error);
      return {
        success: false,
        message: 'Failed to create vehicle',
        data: null
      };
    }
  }
}
```

3. Implement the Controller Layer (Request Handling)

File: src/controllers/VehicleController.ts

```
import { Request, Response } from 'express';
import { VehicleService } from '@/services/VehicleService';
import { VehicleSearchParams, CreateVehicleDto } from '@/models/Vehicle';

export class VehicleController {
  private service: VehicleService;

  constructor() {
    this.service = new VehicleService();
  }

  /**
```

```
   * GET /api/vehicles
   * List all vehicles with optional filtering
   */
  async listVehicles(req: Request, res: Response): Promise<void> {
    try {
      const params: VehicleSearchParams = {
        company_id: req.query.company_id ? parseInt(req.query.company_id as string) : undefined,
        min_capacity: req.query.min_capacity ? parseInt(req.query.min_capacity as string) :
undefined,
        page: req.query.page ? parseInt(req.query.page as string) : 1,
        limit: req.query.limit ? parseInt(req.query.limit as string) : 10,
        search: req.query.search as string
      };

      const result = await this.service.getVehicles(params);

      res.status(result.success ? 200 : 500).json(result);
    } catch (error) {
      console.error('Error in listVehicles:', error);
      res.status(500).json({
        success: false,
        message: 'Internal server error',
        data: null
      });
    }
  }

  /**
   * GET /api/vehicles/:id
   * Get vehicle details by ID
   */
  async getVehicleDetails(req: Request, res: Response): Promise<void> {
```

```
  try {
    const id = parseInt(req.params.id);

    if (isNaN(id)) {
      res.status(400).json({
        success: false,
        message: 'Invalid vehicle ID',
        data: null
      });
      return;
    }

    const result = await this.service.getVehicleById(id);

    res.status(result.success ? 200 : 404).json(result);
  } catch (error) {
    console.error('Error in getVehicleDetails:', error);
    res.status(500).json({
      success: false,
      message: 'Internal server error',
      data: null
    });
  }
}

/**
 * POST /api/vehicles
 * Create a new vehicle (Admin only)
 */
async createVehicle(req: Request, res: Response): Promise<void> {
  try {
    const vehicleData: CreateVehicleDto = req.body;
```

```
      // Basic validation
      if (!vehicleData.name || !vehicleData.license_plate || !vehicleData.capacity ||
!vehicleData.company_id) {
        res.status(400).json({
          success: false,
          message: 'Missing required fields: name, license_plate, capacity, company_id',
          data: null
        });
        return;
      }

      const result = await this.service.createVehicle(vehicleData);

      res.status(result.success ? 201 : 400).json(result);
    } catch (error) {
      console.error('Error in createVehicle:', error);
      res.status(500).json({
        success: false,
        message: 'Internal server error',
        data: null
      });
    }
  }
}
```

4. Define API Routes

File: src/routes/vehicleRoutes.ts

```
import { Router } from 'express';
import { VehicleController } from '@/controllers/VehicleController';

const router = Router();
const vehicleController = new VehicleController();
```

```
// Public routes
router.get('/vehicles', (req, res) => vehicleController.listVehicles(req, res));
router.get('/vehicles/:id', (req, res) => vehicleController.getVehicleDetails(req, res));

// Admin routes (authentication would be added in production)
router.post('/vehicles', (req, res) => vehicleController.createVehicle(req, res));

export default router;
```

5. Create Main Application File

File: src/index.ts

```
import express, { Application, Request, Response } from 'express';
import cors from 'cors';
import dotenv from 'dotenv';
import vehicleRoutes from '@/routes/vehicleRoutes';
import { testConnection } from '@/config/database';

// Load environment variables
dotenv.config();

const app: Application = express();
const PORT = process.env.PORT || 5001;
const SERVICE_NAME = process.env.SERVICE_NAME || 'Vehicle-Catalog-Service';

// Middleware
app.use(cors());
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

// Request logging middleware
app.use((req: Request, res: Response, next) => {
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.path}`);
  next();
});

// Health check endpoint
app.get('/health', (req: Request, res: Response) => {
  res.json({
    service: SERVICE_NAME,
    status: 'healthy',
    timestamp: new Date().toISOString(),
    version: process.env.SERVICE_VERSION || '1.0.0'
  });
});

// API routes
app.use('/api', vehicleRoutes);

// 404 handler
app.use((req: Request, res: Response) => {
  res.status(404).json({
    success: false,
    message: 'Endpoint not found',
    data: null
  });
});

// Error handler
app.use((err: Error, req: Request, res: Response, next: any) => {
  console.error('Unhandled error:', err);
  res.status(500).json({
    success: false,
    message: 'Internal server error',
    data: null
  });
});

// Start server
async function startServer() {
  try {
    // Test database connection
    const dbConnected = await testConnection();

    if (!dbConnected) {
      console.error(' Failed to connect to database. Exiting...');
      process.exit(1);
    }

    // Start listening
    app.listen(PORT, () => {
      console.log('='.repeat(50));
      console.log(` ${SERVICE_NAME} is running`);
      console.log(` Port: ${PORT}`);
      console.log(` Environment: ${process.env.NODE_ENV || 'development'}`);
      console.log(` Health Check: http://localhost:${PORT}/health`);
      console.log(` API Base URL: http://localhost:${PORT}/api`);
      console.log('='.repeat(50));
    });
  } catch (error) {
    console.error('❌ Failed to start server:', error);
    process.exit(1);
  }
}

startServer();
```

6. Update package.json Scripts

File: package.json

```
{
  "name": "vehicle-service",
  "version": "1.0.0",
  "description": "Vehicle/Catalog Microservice for ",
  "main": "dist/index.js",
  "scripts": {
    "dev": "nodemon --watch src --exec ts-node -r tsconfig-paths/register src/index.ts",
    "build": "tsc",
    "start": "node dist/index.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": ["microservice", "vehicle", "catalog"],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2",
    "dotenv": "^16.3.1",
    "cors": "^2.8.5",
    "mysql2": "^3.6.5",
    "knex": "^3.0.1"
  },
  "devDependencies": {
    "typescript": "^5.3.3",
    "@types/node": "^20.10.6",
    "@types/express": "^4.17.21",
    "@types/cors": "^2.8.17",
    "ts-node": "^10.9.2",
    "nodemon": "^3.0.2",
    "tsconfig-paths": "^4.2.0"
  }
}
```

Activity Practice 3: Isolation Testing

Goal

Verify that the service operates correctly and independently from the main application.

Step-by-Step Instructions

1. Start the Microservice

```
# Ensure you are in the vehicle-service directory
cd microservices\vehicle-service

# Install dependencies (if not done already)
npm install

# Install tsconfig-paths for path aliases
npm install -D tsconfig-paths

# Start the service in development mode
npm run dev
```

Expected Output:

```
Database connection successful
==================================================
Vehicle-Catalog-Service is running
Port: 5001
Environment: development
Health Check: http://localhost:5001/health
API Base URL: http://localhost:5001/api
==================================================
```

2. Test Health Check Endpoint

Command:

```
curl http://localhost:5001/health
```

Expected Response (200 OK):

```
{
  "service": "Vehicle-Catalog-Service",
  "status": "healthy",
  "timestamp": "2026-01-10T04:57:42.123Z",
  "version": "1.0.0"
}
```

3. Test Vehicle Listing

Command:

```
curl http://localhost:5001/api/vehicles
```

Expected Response (200 OK):

```
{
  "success": true,
  "message": "Vehicles retrieved successfully",
  "data": [
    {
      "id": 1,
      "name": "Mercedes Sprinter Luxury",
      "description": "40-seat luxury bus with reclining seats",
      "license_plate": "29A-12345",
      "capacity": 40,
      "company_id": 1,
      "company_name": "Phuong Trang Express",
      "featured_image": "https://cloudinary.com/.../bus1.jpg",
      "created_at": "2026-01-05T08:30:00.000Z",
      "updated_at": "2026-01-05T08:30:00.000Z"
    },
    {
      "id": 2,
      "name": "Hyundai Universe Noble",
      "description": "45-seat standard bus",
      "license_plate": "51B-67890",
      "capacity": 45,
      "company_id": 2,
      "company_name": "Mai Linh Express",
      "featured_image": null,
      "created_at": "2026-01-06T10:15:00.000Z",
      "updated_at": "2026-01-06T10:15:00.000Z"
    }
  ],
  "pagination": {
    "page": 1,
    "limit": 10,
    "total": 2,
    "totalPages": 1
  }
}
```

4. Test Vehicle Details Lookup

Command:

```
curl http://localhost:5001/api/vehicles/1
```

Expected Response (200 OK):

```
{
  "success": true,
  "message": "Vehicle found",
  "data": {
    "id": 1,
    "name": "Mercedes Sprinter Luxury",
    "description": "40-seat luxury bus with reclining seats",
    "license_plate": "29A-12345",
    "capacity": 40,
    "company_id": 1,
    "company_name": "Phuong Trang Express",
    "featured_image": "https://cloudinary.com/.../bus1.jpg",
    "created_at": "2026-01-05T08:30:00.000Z",
    "updated_at": "2026-01-05T08:30:00.000Z"
  }
}
```

5. Test Search and Filtering

Test search by name:

```
curl "http://localhost:5001/api/vehicles?search=Mercedes"
```

Test filter by company:

```
curl "http://localhost:5001/api/vehicles?company_id=1"
```

Test filter by minimum capacity:

```
curl "http://localhost:5001/api/vehicles?min_capacity=40"
```

Test pagination:

```
curl "http://localhost:5001/api/vehicles?page=1&limit=5"
```

6. Test Error Handling

Test non-existent vehicle:

```
curl http://localhost:5001/api/vehicles/999
```

Expected Response (404 Not Found):

```
{
  "success": false,
  "message": "Vehicle not found",
  "data": null
}
```

Test invalid vehicle ID:

```
curl http://localhost:5001/api/vehicles/abc
```

Expected Response (400 Bad Request):

```
{
  "success": false,
  "message": "Invalid vehicle ID",
  "data": null
}
```

7. Test Create Vehicle (POST)

Using Postman or curl:

```
curl -X POST http://localhost:5001/api/vehicles `
  -H "Content-Type: application/json" `
  -d '{
    "name": "Thaco Universe TB120S",
    "description": "47-seat luxury sleeper bus",
    "license_plate": "60C-11111",
    "capacity": 47,
    "company_id": 1,
    "featured_image": "https://example.com/bus.jpg"
  }'
```

Expected Response (201 Created):

```
{
  "success": true,
  "message": "Vehicle created successfully",
  "data": {
    "id": 3,
    "name": "Thaco Universe TB120S",
    "description": "47-seat luxury sleeper bus",
    "license_plate": "60C-11111",
    "capacity": 47,
    "company_id": 1,
    "company_name": "Phuong Trang Express",
    "featured_image": "https://example.com/bus.jpg",
    "created_at": "2026-01-10T05:00:00.000Z",
    "updated_at": "2026-01-10T05:00:00.000Z"
  }
}
```

Test duplicate license plate:

```
curl -X POST http://localhost:5001/api/vehicles `
  -H "Content-Type: application/json" `
  -d '{
    "name": "Test Bus",
    "license_plate": "60C-11111",
    "capacity": 40,
    "company_id": 1
  }'
```

Expected Response (400 Bad Request):

```
{
  "success": false,
  "message": "Vehicle with license plate 60C-11111 already exists",
  "data": null
}
```

Testing with Postman

Import Collection

Create a Postman collection with the following requests:

1. **Health Check**
   - Method: GET
   - URL: http://localhost:5001/health
2. **List All Vehicles**
   - Method: GET
   - URL: http://localhost:5001/api/vehicles
3. **Get Vehicle by ID**
   - Method: GET
   - URL: http://localhost:5001/api/vehicles/1
4. **Search Vehicles**
   - Method: GET
   - URL: http://localhost:5001/api/vehicles?search=Mercedes
5. **Filter by Company**
   - Method: GET
   - URL: http://localhost:5001/api/vehicles?company_id=1
6. **Paginated Results**
   - Method: GET
   - URL: http://localhost:5001/api/vehicles?page=1&limit=5
7. **Create Vehicle**
   - Method: POST
   - URL: http://localhost:5001/api/vehicles
   - Headers: Content-Type: application/json

- Body (raw JSON):
- {
-   "name": "New Bus",
-   "license_plate": "XX-XXXXX",
-   "capacity": 40,
-   "company_id": 1
- }

Microservice Independence Demonstration

Running Multiple Services Simultaneously

1. **Start the Vehicle Service** (Port 5001):

```
cd microservices\vehicle-service
npm run dev
```

2. **Start the Main Application** (Port 3000 or 8080):

```
cd ..\..\api
npm run start:dev
```

3. **Verify Independence:**
    - Vehicle Service: http://localhost:5001/api/vehicles
    - Main Application: http://localhost:8080/api/cars
    - Both should work independently without conflicts

Microservice Advantages Demonstrated

1. Independent Deployment

- The Vehicle Service can be deployed, updated, or restarted without affecting other services
- Each service has its own package.json and dependencies

2. Dedicated Port

- Runs on port 5001 (configurable via .env)
- No conflicts with other services

3. Data Ownership

- Owns the cars and bus_companies tables
- Other services access vehicle data only through this API

4. Technology Flexibility

- Could be rewritten in a different language (Python, Go, Java) without affecting other services
- Different teams can work on different services

5. Scalability

- Can be scaled independently based on demand
- If vehicle searches are popular, scale only this service

6. Fault Isolation

- If this service fails, other services (Booking, Payment) continue to work
- Failures are contained

Service Contract Compliance

This implementation fulfills the **Catalog Service API Contract** defined in Lab 4:

Endpoint	Method	Status	Contract Compliance
/api/vehicles	GET	Implemented	Returns paginated list with filtering
/api/vehicles/:id	GET	Implemented	Returns single vehicle with company info
/api/vehicles	POST	Implemented	Creates new vehicle (admin)

Response Format: All responses follow the standardized ApiResponse<T> format:

```
{
  success: boolean,
  message: string,
  data: T | null,
  pagination?: {...}
}
```

Architectural Alignment

This microservice implementation demonstrates key principles from Lab 4:

Decomposition by Business Capability - Catalog management is a distinct capability
Loose Coupling - Communicates only through REST API
High Cohesion - All vehicle-related logic is together
Service Ownership - Owns vehicle/catalog data exclusively
Independent Deployment - Runs separately on dedicated port
API Gateway Ready - Can be easily integrated via API Gateway in Lab 6

Project Structure Summary

```
vehicle-service/
├── src/
│   ├── config/
│   │   └── database.ts        # Database connection
│   ├── models/
│   │   └── Vehicle.ts          # TypeScript interfaces
│   ├── repositories/
│   │   └── VehicleRepository.ts  # Data access layer
│   ├── services/
│   │   └── VehicleService.ts     # Business logic layer
│   ├── controllers/
│   │   └── VehicleController.ts  # Request handling layer
│   ├── routes/
│   │   └── vehicleRoutes.ts      # API route definitions
│   └── index.ts                # Main application entry
├── .env                    # Environment variables
├── .env.example                # Example configuration
├── package.json                # Dependencies and scripts
├── tsconfig.json               # TypeScript configuration
└── README.md                     # Service documentation
```