JavaScript Fundamentals
Intermediate JS Project Analysis

Table of Contents

Ride-App

Implemented Concepts

Node.js

Object and Array De-structuring

- **Why:** To cleanly extract properties from objects (like `req.body` or `req.user`), making the code more readable and concise.
- **Where:** Extensively in controllers and services. For example, in `user-controller.js`, `const { email, password } = req.body;` is used to pull credentials from the request.

Rest and Spread Operators

- **Why:** To create copies of objects and merge new data into them, particularly when preparing data for database insertion or updates.
- **Where:** In `auth-service.js`, `...userData` is used to combine the original user data with the new `hashedPassword` before creating a new user.

ES6 class Syntax

- **Why:** To create a structured and reusable blueprint for repositories, organizing all related database methods into a single, cohesive unit.
- **Where:** In `src/repository/mysql/vehicles.repository.js` to define the `VehicleRepository` class.

ES6 Modules (export and import)

- **Why:** This is the fundamental mechanism for organizing the entire project. It allows the application to be broken into smaller, manageable files (modules) and enables sharing code (functions, classes, variables) between them.
- **Where: Everywhere**. `app.js` imports all the route files, route files import controllers, and controllers import services.

Node.js Architecture (Event Loop, Non-blocking I/O)

- **Why:** This is the core principle that allows the Node.js server to be highly efficient. All database queries are non-blocking, meaning the server can handle many user requests simultaneously without waiting.
- **Where:** This is an underlying concept for the entire application. Every time `await` is used on a database call (e.g., `await findUserByEmail(...)`), it leverages Node.js's non-blocking architecture.

Asynchronous Programming (async/await and Promises)

- **Why:** To manage operations that take time, like database queries. `async/await` makes this code much cleaner and easier to read. `Promise.all` is used to run multiple promises concurrently for a faster startup.
- **Where:** Almost every function that interacts with a database is `async`. In `app.js`, `Promise.all([connectDB(), connectMySQL()])` is used to connect to both databases at once.

Express.js Framework

- **Why:** To build the web server and define the RESTful API. Express simplifies routing, request/response handling, and middleware management.
- **Where:** It's the backbone of the application. `app.js` initializes the Express app, and `src/routes/` files define all API endpoints (`router.get`, `router.post`).

Middleware

- **Why:** To run code before a request is handled by its final route, perfect for tasks like authentication or logging.
- **Where:** `auth-middleware.js` is a critical piece of security that checks for a valid JWT. It's applied in `user-routes.js`, `rider-routes.js`, and `driver-controller.js`.

Password Hashing with bcrypt

- **Why:** For security. It is essential to **never** store passwords in plain text. `bcrypt` creates a secure, one-way hash of a user's password before it's stored.
- **Where:** In `auth-service.js` and `user-service.js`. `bcrypt.hash()` is used during registration, and `bcrypt.compare()` is used during login.

JWT Authentication

- **Why:** To create a stateless and secure way of authenticating users. After login, a JSON Web Token (JWT) is generated and sent to the user to be used in subsequent requests.
- **Where:** In `auth-service.js` to `jwt.sign()` a new token on login, and in `auth-middleware.js` to `jwt.verify()` the token on protected routes.

MySQL

Creating Databases and Tables

- **Why:** To define the rigid structure for core application data that has clear relationships, such as users and their vehicles.
- **Where:** The `src/entities/schema.sql` file contains the `CREATE TABLE` statements for the `users` and `vehicles` tables.

CRUD Operations

- **Why:** These are the four fundamental database operations: Create, Read, Update, and Delete. The repositories are built to perform these actions.
- **Where:** The `users.repository.js` and `vehicles.repository.js` files are dedicated to this (e.g., `createUser` is **C**reate, `findUserByEmail` is **R**ead).

Multiple Tables and JOINS (Relational Design)

- **Why:** To establish a relationship between different pieces of data. Linking tables with foreign keys avoids data duplication and maintains data integrity.
- **Where:** In `schema.sql`, the `vehicles` table has a `driver_id` column with a `FOREIGN KEY` constraint that references the `user_id` in the `users` table.

MongoDB

Connecting with Mongoose

- **Why:** Mongoose provides a straightforward, schema-based solution for modeling application data and interacting with MongoDB.
- **Where:** The connection logic is in `src/config/mongo.js`, and the Mongoose schema is defined in `src/entities/rides-schema.js`.

CRUD operations with Mongoose

- **Why:** To perform the fundamental Create, Read, Update, and Delete operations on the `rides` collection.
- **Where:** The `src/repository/mongodb/rides.repository.js` file is filled with Mongoose operations like `new Ride(ride).save()` (Create) and `Ride.findOne()` (Read).

Schema Design and Data Validation

- **Why:** To enforce a consistent structure on NoSQL documents, ensuring every `ride` has the required fields, correct data types, and valid status values.
- **Where:** In `src/entities/rides-schema.js`, the `rideSchema` is defined with types (`String`, `Number`), default values (`default: "pending"`)

Unimplemented Concepts

Node.js

extends and super (Class Inheritance)

- **Reason for Absence**: The project's structure is straightforward. The one class used, `VehicleRepository`, is self-contained and doesn't need to inherit functionality from a parent class, **making inheritance unnecessary**.

Static Methods, Getters, and Setters

- **Reason for Absence**: The application's logic is primarily functional and service-oriented. The `VehicleRepository` class uses standard instance methods, which is sufficient for its purpose.

Advanced Core Modules (fs, Events, Streams, http)

- **Reason for Absence**: Considerations for the future version.

Advanced Security (Helmet, CORS, Rate Limiting)

- **Reason for Absence**: Considerations for the future version.

Testing (Jest, Supertest)

- **Reason for Absence**: Considerations for the future version.

Performance and Optimization (Cluster, Redis)

- **Reason for Absence**: Considerations for the future version.