Full Stack Home Assignment: "Smart Parking" System

0. Introduction & AI Policy

We value efficiency and pragmatism in engineering. As such, the use of AI coding assistants (e.g., ChatGPT, Cursor, GitHub Copilot, Claude, etc.) is **explicitly encouraged** to streamline your workflow and handle boilerplate code.

However, please proceed with caution: "It works" is not the final metric. Following your submission, we will hold a **live code review session** where you will be expected to walk us through your solution in detail. You must be fully capable of explaining the logic, architectural trade-offs, and control flow of every component. If you cannot explain a specific block of code or a design choice because it was auto-generated, it will be viewed as a gap in your understanding. Treat AI as a tool, but ensure you remain the architect.

1. Description

Implement a web application for managing a parking structure. Focus of the backend part of this assignment is on **concurrency**, **system architecture**, **real-time state management**, and **background processing**. Focus of the frontend part of the assignment is on enhancements of an existing legacy application.

Goal: Ensure that parking spots are reserved correctly without double-booking, and that the system state is consistent across multiple clients.

---

2. Backend Requirements

A. Database & Persistence

The system must use a relational database (PostgreSQL or MySQL) running in a Docker container.

- **Schema:** You are required to design the schema. At a minimum, you will need tables for:
    - `Users` (authentication)
    - `ParkingSpots`
        - `id`
        - `spot_number`

        - `floor_number` - *optional* if you choose to implement a multi-floor parking management
        - `type [Regular/Handicap]` - *optional* if you choose to add accessibility badges to the spots
    - `Reservations`
        - `id`
        - `user_id`
        - `spot_id`
        - `start_time`
        - `end_time`
        - `status [Booked, Completed]`
- **Migrations:** Do not use "auto-sync" or "synchronize: true" features of ORMs. Include files with migration scripts that initializes the database schema.
- **Bonus point**: Configure automatic migration scripts execution using a tool like Flyway, Liquibase, dbmate or similar.

B. Authentication (Pre-seeded)

Implement a standardized JWT (JSON Web Token) authentication flow.

- **No Registration and Reset Password Flows:** The system should boot with a **pre-seeded list of users** in the database.
- **Extensibility:** Design the User entity and Auth service so that OIDC (e.g., Google/Okta/GitHub) could be easily added in the future without rewriting the core logic.
- **Required Seed Users:**
    1. **Driver1:** `driver1@parking.com` / `password123`
    2. **Driver2:** `driver2@parking.com` / `password123`

C. Core API (REST)

Implement the following endpoints:

2. `POST /login`: Returns a JWT.
3. `GET /spots`: Returns a list of all parking spots.
4. `POST /reservations`:
    - Accepts `{ spot_id, start_time, end_time }`.
    - **Crucial:** This endpoint must handle **concurrency**. If two users try to reserve Spot #5 at the exact same millisecond, only one should succeed. The other should receive a user-friendly error.
2. `PUT /reservations/{id}/complete`: Marks a reservation as finished and frees up the spot immediately.

D. Background Processing (The "Stale" Checker)

Implement a background worker (or scheduled cron job) that runs independently of the user request flow.

- **Logic:** The worker should run periodically (e.g., every minute) to check for "stale" reservations.
- **Condition:** If a reservation is still marked "active" but the `end_time` has passed, the system should automatically mark the reservation as `completed` and free the parking spot so others can use it.
- **Logging:** The background job must log its actions e.g., *"Auto-released Spot #105 (Reservation ID 882)"*.

E. Real-Time Updates (WebSockets)

Implement a WebSocket server (using native WS, Socket.io, or similar).

- **Scenario:** Another user is viewing the booking page.
- **Requirement:** When a Driver reserves or releases a spot (via REST API) or when the Background Worker releases a spot, the other Driver's page must update the spot's color (Red/Green) **instantly** without them refreshing the page.

---

3. Frontend Requirements

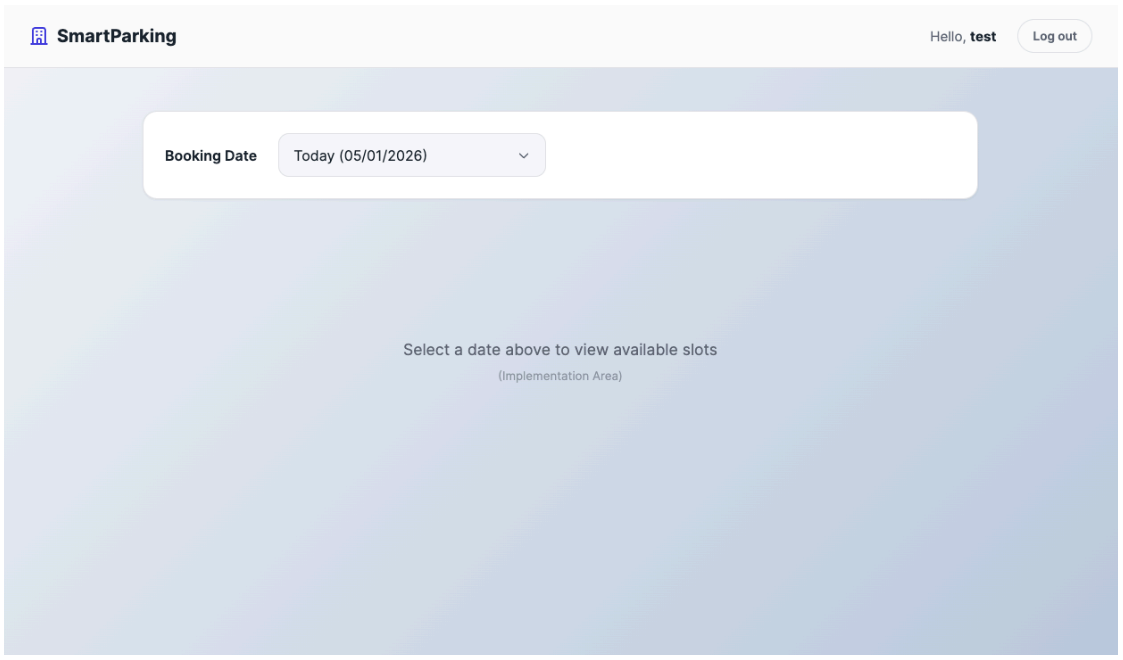
A. What You Have

You are provided with a working frontend app containing:

- **Login Page**: Handles user entry (Creds: `test / test`)
- **Slots Page**: A protected route with a date picker and a logout button
- **Architecture**: A custom Vanilla JS router and component system

🚫 **Constraint**:
You should **not rewrite** the existing Vanilla JS app (Router, AuthService, etc.).
Your goal is to **integrate your code into it**.

Please download the app using the following link: fullstack-smart-parking.zip.



B. The Task

You need to implement the **Parking Slots View** using **Vue** or **React**.
This component should mount inside the `#parking-slots-view` container on the Slots Page.

C. Vue/React Integration

- Embed a Vue/React application into the existing Vanilla JS page
- The Vue/React app must react to **Date Dropdown** changes (which is controlled by the Vanilla JS app)

D. The Logic (Parking Slots)

- Display a grid of parking availability
- **Capacity**: There are **5 Parking Spots**
- **Schedule**: Each spot has **3 Time Slots per day** (e.g., `08:00–12:00`, `12:00–16:00`, `16:00–20:00`)

States

- 🟢 **Available**: User can click to book
- 🔴 **Booked**: Visual indicator (grayed out/red), non-clickable

E. Real-Time Updates (WebSockets) ⚡

The application must be **collaborative**.

Scenario:
User A and User B are both looking at **"Today"**.

- If User A books **Spot 1** at **08:00**
- User B's screen must update **instantly** to show that slot as unavailable

Requirements

- Implement a **WebSocket connection** to listen for booking updates from the server
- Handle **race conditions**
  (What happens if User A and User B click the exact same slot at the exact same time?)

F. Login

Replace the mock `test/test` authentication with real requests to your backend.

---

4. Technical Constraints & Deliverables

A. Docker Compose

The application must be runnable via a single command: `docker-compose up`. This environment should spin up:

1. The Database.
2. The Backend API (and Background Worker).
3. (Optional) The Frontend, if served via container.

B. Documentation (README.md)

- **Setup:** Clear instructions on how to start the app.
- **Architecture Decisions:** Briefly explain:
    - How you handled the race condition (Database locking? Optimistic concurrency?).
    - How you organized the WebSocket vs. REST logic.
    - Assumptions made about the domain.

C. Repository Link

- **Public Access:** Provide a link to a public Git repository (GitHub, GitLab, or Bitbucket) containing all source code, configuration files, and documentation.
- **Commit History:** We value seeing the evolution of the project. Please commit incrementally with meaningful messages (e.g., "Add WebSocket event for spot updates") rather than pushing a single "Initial Commit" at the end.

- **Clean Repository:** Ensure that `node_modules`, build artifacts, and sensitive local environment files (like `.env` containing real keys) are properly excluded via `.gitignore`.

---

5. Bonus (Optional)

- **Analytics Endpoint:** `GET /stats` returning peak occupancy hours.
- **Testing:** Integration tests specifically targeting the race condition (simulating parallel requests).

---

6. Evaluation Criteria

We will evaluate the assignment based on:

1. **Correctness:** Does the concurrency handling work? (No double bookings allowed).
2. **Architecture:** Separation of concerns (Database layer, API layer, Background Worker).
3. **Code Quality:** TypeScript/Language best practices, clarity, and modularity.
4. **Simplicity:** Avoiding over-engineering while meeting the requirements.
5. **API Design:** Correct usage of HTTP methods and status codes
6. **Error Handling:** Consistent and informative error responses
7. **Documentation:** Completeness and clarity of README
8. **Bonus Features:** Implementation of any optional tasks

---

7. Deadline & Planning (Self-Managed)

We value ownership and reliability as much as coding skill. Instead of a fixed deadline, you are responsible for defining your own timeline.

1. Estimation & Commitment Review the requirements thoroughly. Once you have scoped the work, **reply to our email within 24 hours** with a committed delivery date and time (e.g., *"I will submit the repository by Wednesday at 17:00"*).

- *Note:* We are looking for a realistic estimate, not necessarily the fastest one. A reliable delivery is better than a rushed, missed deadline.

2. Execution & Accountability Your submission will be evaluated against your committed deadline. To demonstrate how you managed the timeline:

- **Submission Time:** Ensure the repository link is sent before your self-imposed deadline.
- **Retrospective:** In your `README.md`, include a short **"Planned vs. Actual"** section. Briefly list your initial time estimates for the major components (DB, Auth, WebSocket, etc.) and compare them to the actual time spent. If there were delays, explain how you adjusted your scope or plan to meet the final deadline.