CarHub: A Web-Based Car Rental and Booking System

Rushikesh Rameshwar Kadam[1], Faiz Munawar Tamboli[2], Rahul Sadashiv Khandekar[3], Sushant Gajanan Ghuge[4], Samadhan Yadav[5]
[1,2,3,4]*B. Tech Student, Department of Computer Science and Engineering, MIT College of Railway Engineering & Research, Barshi, India*
[5]*Professor, Department of Computer Science and Engineering, MIT College of Railway Engineering & Research, Barshi, India*

***Abstract*—This paper presents CarHub, a full-stack web application for browsing vehicles and coordinating rental bookings through customer, administrator, and car-provider interfaces. The implementation uses a React and Vite client, a Node.js and Express server, and a MySQL data store accessed through mysql2 and Sequelize. The reviewed workflows include vehicle discovery, account registration and email one-time-password login, protected booking pages, date-overlap checks, rental and transfer booking modes, administrative management, and Razorpay order creation with server-side signature verification. The system is examined through a source-based functional inventory rather than a live user study or performance benchmark. This distinction is important: implemented routes and controller logic demonstrate intended functionality, but do not by themselves establish usability, production reliability, or payment settlement. The paper describes the architecture, implementation choices, observed workflow coverage, and limitations, and proposes transactional booking safeguards, automated tests, accessibility evaluation, and measured deployment trials as future work.**

***Index Terms*—Accessibility, Booking Management, Car Rental, Payment Verification, Role-Based Access, Web Application.**

I. INTRODUCTION

Vehicle rental services coordinate several related activities: presenting available cars, communicating rental terms, collecting trip details, checking dates, and recording booking and payment status. When these activities are handled through disconnected channels, customers may find it difficult to compare options while operators must reconcile vehicle and customer records. A web application can bring these steps into a shared interface, provided that its booking rules and role boundaries are explicit.

CarHub is a web-based rental platform implemented as a single-page client and an API-backed server. Its customer-facing pages support account access, car discovery, vehicle details, booking review, and booking history. Separate protected areas are provided for administrators and car registrants. The codebase also contains event and tour workflows, which are adjacent services rather than the central subject of this paper.

The contribution of this paper is a source-grounded description of the application architecture and booking/payment workflows, together with a transparent account of what has and has not been evaluated. The evidence comes from inspection of the supplied frontend and backend implementation. No claim of measured speed, user satisfaction, or production-scale reliability is made. The remainder of the paper describes related technical foundations, system design, implementation, a code-level feature inventory, and limitations.

II. RELATED WORK

Modern web interfaces commonly separate reusable client-side components from server-side business rules. React documents a component-based approach for building user interfaces [1], while Vite provides the development and build tooling used by the CarHub frontend [2]. On the server, Node.js and Express provide the runtime and routing foundation for request handling [3], [4]. These technologies are enabling components; their use alone does not guarantee application quality or scalability.

Relational databases remain suitable for records with clear relationships, such as users, cars, and bookings. MySQL provides relational storage, and Sequelize

offers an object-relational mapping layer for JavaScript applications [5], [6]. CarHub uses both Sequelize models and mysql2-based query helpers, so the system combines model-oriented persistence with direct parameterized SQL in some workflows.

Reservation systems also need to coordinate scarce resources across time. In a car-rental context, a vehicle should not be assigned to overlapping active bookings. A pre-insert availability query is a useful business rule, but robust reservation systems must also consider concurrent requests and transaction isolation. Payment integrations add a separate trust boundary: the server must not treat a client-side success screen as proof of payment. Razorpay documents an order and signature-verification flow for integrations [8]. These practices inform the analysis below; they are not evidence that the application has passed an independent security or payment audit.

III. SYSTEM ARCHITECTURE

A. Architectural Layers

The application follows a client–API–database structure. The browser client is built with React, Vite, and React Router. It presents public pages and authenticated customer, administrator, and provider workspaces. The Express API groups routes and controllers by user, administrator, and car-registrant functions. MySQL stores application records. The server accesses the database through a mysql2 connection pool and Sequelize configuration. A payment controller communicates with Razorpay for order creation and verifies payment signatures using a server-side secret.

Web clients	Application API	Relational store
React + Vite	Node.js + Express	MySQL
Customer | Admin | Provider	Routes | Controllers | Services	mysql2 + Sequelize

Fig. 1. High-level CarHub architecture and principal application layers.

The web client sends requests to role-specific API routes. The server validates request data, applies booking or account rules, and reads or updates relational records. Payment operations follow a separate path from booking creation: a pending booking is associated with an order, and a subsequent verification request updates the booking only after the backend checks the returned signature. Fig. 1 is a logical view of the inspected code, not a deployment topology or network-security diagram.

B. User Roles and Functional Scope

The customer role can register, sign in, browse the car catalogue, inspect a car, submit a booking, and view personal bookings. The authentication implementation hashes registration passwords with bcryptjs, sends a one-time password during login, and issues a JSON Web Token after OTP verification. The user interface protects booking-related routes, while backend middleware verifies bearer tokens and limits the customer API to the user role. JWT is a compact claims format standardized in RFC 7519 [7]; correct token issuance and validation still depend on secure key management and deployment configuration.

The administrator workspace includes routes for dashboards, cars, categories, bookings, invoices, calendars, feedback, contacts, and tour operations. The car-registrant workspace includes its own login, dashboard, car and booking pages, profile, and tour functions. This separation indicates distinct workflows, although a complete authorization assessment would need to examine every endpoint and database operation independently.

C. Booking and Fare Workflow

The booking service loads the selected car, rejects missing, inactive, or unavailable vehicles, normalizes the booking mode and billing type, and checks for an existing booking whose date interval overlaps the requested interval. The service treats a transfer as a same-day booking and calculates a total from the configured rate, selected billing type, date range, and when billing per kilometre the supplied distance. A booking is inserted with payment method, payment status, and initial booking status. Cash bookings are marked booked by the service; online bookings remain pending until the payment workflow advances them.

This sequence gives the application a clear place to enforce business rules. However, the overlap check and subsequent insert are separate operations in the inspected service. Two near-simultaneous requests could therefore both observe availability before either insert. A transaction with an appropriate locking or uniqueness strategy, followed by concurrency tests, is needed before claiming race-free reservation behavior.

D. Technology Summary

Table I

Technologies Observed in the Implementation

Layer	Technology	Observed responsibility
Frontend	React, Vite, React Router	Page components, client routes, and role-based workspaces
Backend	Node.js, Express	API routing and booking, account, and payment logic
Persistence	MySQL, mysql2, Sequelize	Relational records and query/model access
Authentication	bcryptjs, JWT, email OTP	Password hashing, token issuance, and login verification
Payment	Razorpay SDK	Order creation and server-side signature verification

IV. IMPLEMENTATION

A. Client and Navigation

The frontend uses React components and React Router routes to organize customer, administrator, and car-registrant pages. Public routes include car browsing, car details, registration, login, contact, and related content. Booking and account pages are wrapped in client-side route guards that check locally stored tokens before rendering protected screens. These guards improve navigation flow but should be treated as a user-interface control only; server-side authorization remains essential.

The navigation component exposes links for cars, bookings, events, tours, about, and contact, as well as separate login choices for customers, administrators, and car registrants. The routing layer uses lazy-loaded page components and a loading fallback, which can defer page modules until navigation. No bundle-size or rendering benchmark was performed for this paper.

B. API and Data Access

The backend organizes routes and controllers by domain and role, with separate booking, payment, authentication, and management functions. Booking queries use parameter placeholders for user-supplied values in the inspected service. The database configuration creates a pooled mysql2 connection and a Sequelize instance. This dual access pattern supports both direct SQL and model operations, but it also requires consistent schema assumptions and error handling across both paths.

The domain model includes users, administrators, car registrants, cars, bookings, categories, cancellation requests, feedback, events, and tours. This scope extends the application beyond a simple public catalogue: operational records can be maintained by administrators and providers while customers retain access to their own booking workflows. Schema migrations and production data consistency should be documented and tested as deployment practices mature.

C. Authentication and Payment

During registration, the backend validates required fields and phone format, checks existing email and phone values, hashes the password, and persists the new user. Login verifies the supplied password and sends an email OTP; successful OTP verification issues a JWT containing the user identifier and role. The API middleware checks bearer tokens and applies role checks to protected user routes. A production review should additionally assess rate limits, OTP attempt controls, expiration handling, secret rotation, transport security, and account-recovery flows against an established verification baseline such as OWASP ASVS [9].

For online payments, the backend creates an order through the Razorpay SDK and associates its identifier with a pending booking. The verification handler computes an HMAC-SHA256 value from the order and payment identifiers using the server-side secret, then compares it with the supplied signature before marking the booking as paid/booked. The repository also contains a development-only simulated payment path when valid gateway credentials are absent; that path is explicitly not a real charge or proof of a live payment. A production deployment must use valid gateway credentials and independently verify amount, booking ownership, order association, replay behavior, and failure handling.

V. RESULTS AND DISCUSSION

A. Assessment Method

The evaluation reported here is a code-level functional inventory. The frontend route tree, authentication handlers, booking service, payment controller, database configuration, and package manifests were inspected to identify implemented responsibilities. This is a reproducible description of code structure,

not an executed integration test. The papC`er does not report a live database run, gateway transaction, user study, load test, accessibility audit, or measured response time.

B. Observed Workflow Coverage

Table II

Source-Level Workflow Inventory

Workflow	Evidence in supplied source	Interpretation
Vehicle discovery	Customer car-list and car-detail routes	Catalogue browsing is represented in the client and API.
Role separation	Customer, admin, and car-registrant routes	Distinct interfaces and protected route groups are present.
Booking	Availability query, date-overlap condition, fare calculation, insert	Core booking rules are implemented; concurrency is not proven.
Authentication	Password hashing, email OTP, JWT middleware	Several account controls are present; independent security testing is required.
Online payment	Order creation and HMAC signature check	Payment integration logic exists; no live settlement was tested.

The source review shows that the project covers more than car discovery: booking state, customer authentication, administrative operations, provider functions, and payment-related endpoints are represented. The strongest design feature is the separation of user-facing roles and domain-specific backend modules. The main operational concern visible from the booking service is the time gap between availability checking and insertion. Table II therefore distinguishes source evidence from conclusions that would require runtime validation.

C. Interpretation

The codebase provides a credible foundation for a multi-role rental application, but feature presence should not be equated with demonstrated effectiveness. For example, a protected frontend route does not establish backend authorization, a successful local payment simulation does not establish gateway settlement, and an overlap query does not establish correctness under concurrent load. These distinctions motivate the validation plan and limitations described next.

VI. LIMITATIONS AND FUTURE WORK

First, the reported assessment is based on source inspection. Automated unit and integration test results, deployment configuration, production logs, and user feedback were not supplied as evidence for this paper. The system should be exercised with seeded test data, invalid input, boundary dates, cancellation states, failed payments, and concurrent booking attempts. Performance should be measured under stated workload and hardware conditions before making scalability claims.

Second, the booking availability check should be made atomic with the booking write or protected through a database-level concurrency strategy. Tests should verify boundary semantics for inclusive dates, transfer bookings, and each billing mode. The payment workflow should be validated against gateway test credentials and should bind order amount, booking identity, and payment result before updating state. The development simulation must remain isolated from production configuration.

Third, the persistence layer uses both direct mysql2 queries and Sequelize. A documented schema/migration strategy and a consistent transaction boundary would reduce drift between access paths. Security work should include endpoint-by-endpoint authorization tests, secret handling review, abuse controls for OTP flows, and a formal verification checklist [9]. The interface also needs keyboard, contrast, screen-reader, and responsive-layout evaluation against WCAG 2.2 [10].

Future work should prioritize automated tests for booking and payment state transitions, a transactional availability design, observable error handling, accessibility testing, and user-centered evaluation. A controlled pilot could then measure task completion, booking error rate, and response time. These outcomes should be reported with the test setup and sample size rather than inferred from the implementation alone.

VII. CONCLUSION

CarHub is a full-stack web application that brings car discovery, booking, role-specific management, and payment-related workflows into one platform. Its React/Vite client, Express API, MySQL persistence, user and provider areas, overlap checks, OTP/JWT authentication, and Razorpay integration are visible in

the supplied source. The implementation review establishes feature coverage at the code level, but not measured usability, reliability, or production readiness. Transaction-safe booking, systematic testing, security review, accessibility evaluation, and live payment validation are the necessary next steps toward a dependable rental service.

ACKNOWLEDGMENT

The authors express their gratitude to their project guide, the Head of the Department, and the faculty members of the Department of Computer Science and Engineering, MIT College of Railway Engineering & Research, Barshi, for their guidance and support. They also acknowledge the cooperation of their team members, friends, and family during the project work.

REFERENCES

[1] Meta Open Source, "Quick start," *React Documentation*, [Online]. Available: https://react.dev/learn. [Accessed: Nov. 9, 2026].

[2] Vite Team, "Getting started," *Vite Guide*, [Online]. Available: https://vite.dev/guide/. [Accessed: Nov. 9, 2026].

[3] OpenJS Foundation, "Node.js documentation," [Online]. Available: https://nodejs.org/docs/. [Accessed: Nov. 9, 2026].

[4] Express.js, "Express—Node.js web application framework," [Online]. Available: https://expressjs.com/. [Accessed: Nov. 9, 2026].

[5] Oracle, "MySQL 8.4 reference manual," [Online]. Available: https://dev.mysql.com/doc/refman/8.4/en/. [Accessed: Nov. 9, 2026].

[6] Sequelize, "Sequelize v6 documentation," [Online]. Available: https://sequelize.org/docs/v6/. [Accessed: Nov. 9, 2026].

[7] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," IETF, RFC 7519, May 2015. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7519.

[8] Razorpay, "Server integration: Node.js," [Online]. Available: https://razorpay.com/docs/payments/server-integration/nodejs/integration-steps/. [Accessed: Nov. 9, 2026].

[9] OWASP Foundation, "Application security verification standard," [Online]. Available: https://owasp.org/www-project-application-security-verification-standard/. [Accessed: Nov. 9, 2026].

[10] World Wide Web Consortium, "Web content accessibility guidelines (WCAG) 2.2," W3C Recommendation, Oct. 5, 2023. [Online]. Available: https://www.w3.org/TR/WCAG22/