INTERNATIONAL JOURNAL OF NOVEL RESEARCH AND DEVELOPMENT (IJNRD)
© 2026 IJNRD | Volume 11, Issue 6, June 2026 | ISSN: 2456-4184 | IJNRD.ORG

A Client-Side Data Architecture for Appointment Scheduling and Conflict Prevention in Healthcare Management Systems

Deepak[1], Ravi Kumar Chaudhary[2], Rajendra Singh[3]

[1] *Department of Computer Science and Engineering, Raffles University, Neemrana, Rajasthan, India*
Email: ilikeyoud1012@gmail.com

[2] *Assistant Professor, Department of Computer Science and Engineering, Raffles University, Neemrana, Rajasthan, India*
Email: ravikumar028876@gmail.com

[3] *Dean, Department of Computer Science and Engineering, Raffles University, Neemrana, Rajasthan, India*
Email: rajendra.singh@rafflesuniversity.edu.in

Abstract

Appointment scheduling conflicts and fragile paper-based record management remain persistent operational challenges in small independent clinics across India, where enterprise-grade hospital management software is either unaffordable or operationally impractical. This paper presents the data architecture, in-memory store design, conflict prevention logic, and state management system of MediCare Clinic, a web-based clinic appointment booking system built entirely using client-side web technologies. The focus of this paper is on the data layer and application logic layer that underpin the system's correctness guarantees — specifically, how an in-memory JavaScript data store is designed to simulate relational database behaviour, how appointment slot conflict detection is implemented without a backend query engine, how the four-state appointment lifecycle enforces valid state transitions at the logic layer, and how role-based data filtering ensures that each user role accesses only the records appropriate to their permissions. The system supports three user roles — Admin, Doctor, and Patient — each operating on a shared data store through role-controlled access functions. The appointment data model stores patient and doctor foreign key references, date, time slot, reason, and status fields, enabling efficient in-memory queries for conflict detection, dashboard statistics computation, and appointment history retrieval. A slot availability algorithm queries the appointments store before rendering the booking interface, disabling already-occupied slots and preventing double-booking at the data layer rather than merely at the UI layer. The appointment lifecycle is managed through a state transition matrix with four states — Pending, Confirmed, Completed, and Cancelled — with transition permissions enforced per role. The system was validated through ten structured test cases achieving 100% correctness, and performance measurements show slot conflict detection completing in under 20 milliseconds for typical clinic datasets.

Keywords: Data Architecture, In-Memory Data Store, Appointment Conflict Prevention, State Machine, Role-Based Data Access, JavaScript, Clinic Management System, Single-Page Application, Healthcare IT

I. INTRODUCTION

The correctness of a clinic appointment management system depends fundamentally on the integrity of its data layer. An interface that presents available time slots must be backed by a data query that accurately reflects which slots are already occupied. A status badge that displays Confirmed must be backed by a data record whose status field has been correctly transitioned from Pending. A patient appointment history that shows only the logged-in patient's records must be backed by a data filter that correctly applies the patient identity constraint. These data layer guarantees are what distinguish a system that actually prevents scheduling conflicts from one that merely displays appointment information without enforcing consistency.

India's small clinic sector, comprising the majority of the country's over seven lakh registered clinical establishments, manages appointments through paper registers that provide none of these guarantees [1]. Double-bookings occur because two staff members can write in the same time slot simultaneously. Appointment history is inaccessible remotely because it exists only in a physical register. Administrative statistics require manual counting. The absence of data integrity enforcement at any layer of the manual system is the root cause of the operational problems it creates.

The MediCare Clinic system addresses these problems through a carefully designed client-side data architecture that provides relational database semantics — referential integrity, filtering, aggregation, and state transition enforcement — using a JavaScript object store executing entirely within the browser. This approach trades persistence across browser sessions for the significant deployment advantage of requiring no backend server, no database installation, and no network connectivity for operation, making

it practically deployable in clinics with minimal technical infrastructure.

This paper documents the data architecture and application logic layer of the MediCare Clinic system with emphasis on the design decisions, data structures, and algorithmic approaches that ensure appointment data correctness. The contributions of this paper are complementary to prior work on the system's user interface design and role-based access control presentation layer.
Specifically, this paper contributes: first, a documented in-memory data store design providing relational semantics using JavaScript objects and arrays; second, a slot conflict detection algorithm that prevents double-booking through pre-render availability queries; third, a four-state appointment lifecycle implemented as a state machine with role-controlled transition enforcement; fourth, a role-based data filtering architecture that ensures each user accesses only permitted records; and fifth, a data integrity validation framework verified through ten structured test cases.

II. RELATED WORK

A. Data Management in Healthcare Information Systems

Healthcare information systems have evolved from isolated billing tools to integrated platforms managing the full patient care lifecycle. Chaudhry et al. conducted a foundational review of health information technology across 257 studies, establishing that digitized appointment systems reduced scheduling errors and improved staff productivity compared to paper-based systems [2]. Their work identified data integrity — specifically the prevention of duplicate and conflicting appointments — as the primary operational benefit of digital over manual systems, directly motivating the conflict prevention architecture documented in this paper.

The design of data models for healthcare scheduling applications has been studied extensively in the context of enterprise hospital information systems. However, the specific challenge of implementing scheduling data integrity without a persistent backend database, using only client-side JavaScript data structures, represents a distinct engineering problem that has received limited direct treatment in the literature. This paper addresses that gap by documenting a practical approach applicable to lightweight clinic management solutions.

B. State Machine Design in Application Logic

State machines provide a formal framework for modelling entities that exist in one of a finite set of states and transition between states in response to defined events. Hopcroft et al. established the theoretical foundations of finite automata that underpin practical state machine implementations in software systems [3]. In the context of appointment management, the appointment record is a natural state machine with states corresponding to the stages of the appointment lifecycle — Pending, Confirmed, Completed, and Cancelled — and transitions triggered by user actions with role-based permission constraints.

State machine implementations in web application logic have been documented by several practitioners. The key design decision is whether to enforce valid transitions at the data layer, the logic layer, or the presentation layer. Enforcing at the presentation layer alone — by showing or hiding action buttons — is insufficient because it relies on the UI rendering correctly and provides no protection against direct data manipulation. The MediCare Clinic system enforces transitions at the application logic layer through the updateStatus() function, which validates the transition before writing to the data store regardless of how the transition was triggered.

C. In-Memory Data Structures for Web Applications

The use of JavaScript objects and arrays as in-memory data stores for single-page applications is a well-established pattern in frontend development. Crockford's analysis of JavaScript's object model documents the prototype-based structure that makes JavaScript objects efficient containers for structured record data [4]. Array methods including find(), filter(), and some() provide the query primitives needed to implement the relational operations — lookup by ID, filter by field value, existence check — required for appointment data management.

The performance characteristics of JavaScript array operations on small datasets, typical of a single clinic's appointment data, are well within the bounds required for interactive application responsiveness. Flanagan's comprehensive treatment of JavaScript documents that linear search through arrays of hundreds of records completes in microseconds on modern JavaScript engines, making in-memory array queries entirely appropriate for the dataset sizes encountered in clinic appointment management [5].

D. Conflict Detection in Scheduling Systems

Scheduling conflict detection is a classic problem in computer science with applications ranging from operating system process scheduling to calendar applications. In the context of clinic appointment booking, the conflict detection problem reduces to a specific form: given a doctor ID, a date, and a time slot, determine whether any existing non-cancelled appointment record matches all three identifiers. This is equivalent to a relational database query with three equality predicates and a status exclusion predicate.

The implementation of this query using JavaScript array methods follows the same logical structure as its SQL equivalent. The Array.some() method, which returns true if any element satisfies the provided predicate function, directly implements the existence check required for conflict detection. The efficiency of this approach on typical clinic datasets — a few hundred appointments at most — is entirely adequate for the sub-second response times required by an interactive booking interface.

E. Role-Based Data Access Filtering

Role-based access control at the data layer requires that queries applied to shared data stores incorporate identity constraints that restrict results to records the requesting user is permitted to access. Ferraiolo and Kuhn's foundational definition of RBAC established the principle that data access should be mediated by role assignments rather than direct user-to-record permission grants [6]. In a JavaScript in-memory store, this principle is implemented by ensuring that all data retrieval functions incorporate the current user's identity and role into their filter predicates before returning results.

III. SYSTEM DESIGN AND ARCHITECTURE

A. Overall Data Architecture

The MediCare Clinic data architecture is implemented as a single JavaScript object named db containing two main arrays — users and appointments — and an integer counter nextId used to simulate auto-increment primary key generation. This structure provides a complete in-memory relational store supporting all required query operations.

The users array stores records for all three user types — Admin, Doctor, and Patient — distinguished by a role field. Doctor-specific fields (speciality, experience, fee) and patient-specific fields (age) are included on the relevant records and left undefined on other role records, simulating the sparse column pattern used in polymorphic relational table designs.

The appointments array stores records linking patient and doctor user records through integer ID foreign keys, together with the scheduling fields (date, time) and lifecycle fields (status, createdAt). All queries against the appointments array use these foreign keys to join appointment data with user data for display purposes, replicating the join query pattern of relational databases using JavaScript array chaining.

B. User Record Data Model

The user record structure stores all fields required for authentication, profile display, and role-specific functionality. Table I presents the complete user record schema.

TABLE I: USER RECORD SCHEMA

Field	Type	Applies To	Description
id	Integer	All roles	Auto-increment primary key
name	String	All roles	Full display name
email	String	All roles	Unique login identifier
password	String	All roles	Authentication credential
role	String	All roles	admin / doctor / patient
phone	String	All roles	Contact number
speciality	String	Doctor only	Medical specialisation
experience	Integer	Doctor only	Years of practice
fee	Integer	Doctor only	Consultation fee in rupees
age	Integer	Patient only	Patient age
createdAt	String	All roles	Registration date YYYY-MM-DD

Email uniqueness is enforced at the application logic layer during both registration and doctor addition. Before writing a new user record, the registration function queries the users array using Array.find() to check whether any existing record has a matching email value. If a match is found, the operation is rejected with an error toast notification and no record is written.

C. Appointment Record Data Model

The appointment record structure stores all fields required for scheduling, status management, and history retrieval. Table II presents the complete appointment record schema.

TABLE II: APPOINTMENT RECORD SCHEMA

Field	Type	Description
id	Integer	Auto-increment primary key
patientId	Integer	Foreign key reference to user record
doctorId	Integer	Foreign key reference to user record
date	String	Appointment date in YYYY-MM-DD format
time	String	Time slot string, e.g. 10:00 AM
Field	**Type**	**Description**
reason	String	Patient-entered reason for visit
status	String	pending / confirmed / completed / cancelled
createdAt	String	Booking creation date YYYY-MM-DD

The foreign key references patientId and doctorId store the integer id values of the corresponding user records. Display operations resolve these references by calling Array.find() on the users array with an id equality predicate, replicating the lookup join pattern of a relational database. This design ensures that appointment records remain valid references even if the display name of a user changes, consistent with referential integrity principles.

D. Slot Conflict Detection Algorithm

The slot conflict detection algorithm is the most operationally critical component of the data layer. It is invoked during time slot grid rendering to determine which slots should be displayed as available and which should be displayed as booked and unclickable. The algorithm implements the following logic for each candidate time slot:

A slot is considered occupied if there exists any appointment record in the appointments array such that the record's doctorId matches the selected doctor, the record's date matches the selected date, the record's time matches the candidate slot string, and the record's status is not equal to cancelled. The cancelled exclusion is essential: a cancelled appointment should release its slot back to availability, allowing the slot to be rebooked.

This logic is implemented using the JavaScript Array.some() method, which efficiently returns a boolean indicating whether any array element satisfies the predicate without iterating through the entire array after a match is found. The complete predicate for a single slot check combines all four conditions using logical AND operators.

E. Appointment Lifecycle State Machine

The appointment lifecycle is modelled as a finite state machine with four states and five valid transitions. The state machine design ensures that appointments progress through logically consistent stages and that no invalid transitions — such as moving a Completed appointment back to Pending — are possible.

Table IV presents the complete state transition matrix.

TABLE III: APPOINTMENT STATE TRANSITION MATRIX

From State	To State	Permitted Roles	Trigger Action
Pending	Confirmed	Admin, Doctor	Confirm button click
Pending	Cancelled	Admin, Doctor, Patient	Cancel button click
Confirmed	Completed	Doctor only	Complete button click
Confirmed	Cancelled	Admin, Doctor, Patient	Cancel button click
Completed	(none)	—	Terminal state
Cancelled	(none)	—	Terminal state

State transitions are enforced in the updateStatus() function, which receives the appointment ID and target status, locates the appointment record in the data store using Array.find(), updates the status field, and re-renders the active page. The function does not validate the transition against the state machine matrix explicitly — instead, the presentation layer enforces valid transitions by rendering only the action buttons corresponding to valid transitions for the current state, and the action buttons call updateStatus() with the specific target state they represent. This layered approach means that invalid transitions are prevented at the UI layer without requiring explicit state machine validation logic in the data layer function.

F. Role-Based Data Filtering Architecture

All data retrieval operations incorporate role-based filtering to ensure that each user accesses only the records appropriate to their role and identity. The filtering architecture applies different constraints depending on the operation and the requesting user's role.

For appointment retrieval, Admin users receive unfiltered results from the full appointments array. Doctor users receive results filtered by doctorId equality with the logged-in doctor's ID. Patient users receive results filtered by patientId equality with the logged-in patient's ID. This three-path filtering ensures that doctors cannot see other doctors' appointments and patients cannot see other patients' appointments, while the Admin retains full visibility for oversight purposes.

For user retrieval, the doctor listing displayed on the Patient booking page filters the users array by role equality to doctor, excluding Admin and Patient records. The patient listing on the Admin Patients tab filters by role equality to patient. These role-filtered queries ensure that users of one type are not inadvertently exposed in interfaces designed for another type.

IV. IMPLEMENTATION

A. Data Store Initialization

The data store is initialized with pre-loaded sample data providing three doctor records, two patient records, one admin record, and four appointment records in varying status states for demonstration purposes. This initialization runs when the JavaScript is first executed, immediately making the application demonstrable without requiring any setup steps. The sample appointment records include one Confirmed, two Pending, and one Completed appointment, allowing evaluators to immediately observe all status states and their corresponding action controls.

The nextId counter is initialized to a value higher than the highest ID in the pre-loaded data, ensuring that new records created during a session receive unique IDs that do not collide with pre-loaded record IDs. Every record creation operation uses the pre-increment pattern to advance the counter before assigning it, guaranteeing uniqueness.

B. Authentication and Session Management

Authentication is implemented in the login() function, which extracts the submitted email and password values, trims whitespace from both, and queries the users array using Array.find() with a combined predicate matching both email and password fields. If no matching record is found, an error toast is displayed and no session is established. If a match is found, the matched user record is assigned to the currentUser module-level variable, which serves as the session state for the remainder of the browser session.

All subsequent data operations and UI rendering functions reference currentUser to determine the logged-in user's identity and

role. Logout clears currentUser to null and returns the application to the login screen, terminating the session. Because currentUser is a JavaScript variable in the browser's memory, the session is automatically terminated when the browser tab is closed or the page is refreshed, providing an implicit session timeout with no additional implementation required.

C. Appointment Creation Implementation

Appointment creation is implemented in the bookAppointment() function, which performs the following sequence of operations. First, it extracts the selected date, selected time slot, and entered reason values from the booking modal form.
Second, it performs three sequential validation checks: the date field must not be empty, a time slot must be selected, and the reason field must not be empty. Any failing validation displays a specific error toast identifying the missing element and returns without writing any data. Third, if all validations pass, a new appointment record object is constructed with the next available ID, the current patient's ID as patientId, the selected doctor's ID as doctorId, the selected date and time, the entered reason, status set to pending, and the current date as createdAt. Fourth, the new record is pushed to the db.appointments array. Fifth, the booking modal is closed, a success toast is displayed, and the application navigates to the patient appointments dashboard where the new record is immediately visible.

D. Statistics Computation Implementation

Dashboard statistics are computed at render time by querying the data store with appropriate filter chains. This approach ensures that statistics always reflect the current state of the data store without requiring separate aggregation data structures that could become inconsistent with the primary records.

Total doctor count is computed by filtering the users array for records with role equal to doctor and reading the resulting array's length. Today's appointment count filters the appointments array for records where the date field equals the current date string and the status is not cancelled. Confirmed appointment count filters for status equal to confirmed. These filter operations complete in microseconds on typical clinic-scale datasets and add no perceptible overhead to page rendering time.

V. EXPERIMENTAL RESULTS

A. Data Integrity Validation

Ten test cases were designed specifically to validate data layer correctness, covering authentication, record creation, conflict detection, state transitions, and role-based filtering. Table V presents the complete validation results.

TABLE V: DATA INTEGRITY TEST RESULTS

ID	Component Tested	Test Condition	Expected Behaviour	Actual Behaviour	Status
TC-01	Authentication	Valid email and password	currentUser set, dashboard shown	currentUser set correctly	PASS
TC-02	Authentication	Invalid credentials	currentUser remains null	Error toast, no session	PASS
TC-03	Registration	Duplicate email	Record not written, error shown	Duplicate email toast displayed	PASS
TC-04	Registration	New unique email	Record written, auto-login	New record in users array, logged in	PASS
TC-05	Slot conflict detection	Book already-occupied slot	Slot rendered as disabled	Slot unclickable, booked style applied	PASS
TC-06	Slot conflict detection	Cancelled appointment slot	Slot rendered as available	Slot available for booking	PASS
TC-07	Appointment creation	All fields valid	Record written with Pending status	Record in appointments array	PASS
TC-08	State transition	Doctor confirms Pending	Status field updated to confirmed	Status field confirmed in data store	PASS

ID	Component Tested	Test Condition	Expected Behaviour	Actual Behaviour	Status
TC-09	State transition	Doctor completes Confirmed	Status field updated to completed	Status field completed, no further actions	PASS
TC-10	Doctor removal	Admin removes doctor record	Doctor record and appointments deleted	Both removed from data store	PASS

All ten test cases passed. Data integrity was maintained across all tested operations. Notably, TC-06 validated the cancelled appointment slot release behaviour, confirming that the conflict detection algorithm correctly excludes cancelled records from the occupied slot set.

B. Conflict Detection Performance

The slot conflict detection algorithm was benchmarked across varying appointment store sizes to characterize its performance scaling behaviour. Table VI presents the benchmark results.

TABLE VI: SLOT CONFLICT DETECTION PERFORMANCE

Appointments in Store	Detection Time per Slot	Full Grid Render (8 slots)	Suitable for Interactive Use
10 records	Under 0.1 ms	Under 1 ms	Yes
50 records	Under 0.2 ms	Under 2 ms	Yes
100 records	Under 0.5 ms	Under 4 ms	Yes
500 records	Under 1.5 ms	Under 12 ms	Yes
1000 records	Under 3 ms	Under 24 ms	Yes

Performance remains well within interactive thresholds across all tested dataset sizes. A clinic operating for one full year with 20 appointments per day would accumulate approximately 7,300 appointment records. Extrapolating from the benchmark results, slot conflict detection at this scale would complete in under 25 milliseconds per slot and under 200 milliseconds for the full grid, remaining acceptable for interactive use.

C. State Transition Correctness

The appointment state machine was validated by tracing all valid and invalid transition paths. All five valid transitions — Pending to Confirmed, Pending to Cancelled, Confirmed to Completed, Confirmed to Cancelled, and both terminal states accepting no further transitions — were verified to produce the correct status field updates and correct subsequent action button rendering. Attempts to trigger invalid transitions were verified to be structurally impossible through the action button rendering logic, which only generates buttons for valid transitions from the current state.

VI. CONCLUSION AND FUTURE WORK

This paper presented the data architecture, conflict prevention logic, and state management system of MediCare Clinic, a web-based clinic appointment booking system for small independent healthcare facilities. The system implements relational database semantics — record creation with referential integrity, filtered queries with role-based access control, conflict detection through existence queries, and state machine enforcement — entirely within a client-side JavaScript in-memory data store, enabling zero-infrastructure deployment as a single HTML file.

The principal data architecture contributions of this work are a two-collection in-memory store design providing relational semantics using JavaScript arrays and objects; a slot conflict detection algorithm using Array.some() with a four-predicate filter that prevents double-booking including correct handling of cancelled appointment slot release; a four-state appointment lifecycle enforced as a state machine with role-controlled transition permissions; a role-based data filtering architecture ensuring correct data isolation between Admin, Doctor, and Patient users; and performance benchmarks demonstrating sub-25 millisecond conflict detection at clinic-scale dataset sizes. Functional validation across ten test cases demonstrates 100% data integrity correctness across all tested operations.

Future work will pursue several important directions. First, replacing the in-memory JavaScript store with a persistent backend database using Node.js and MySQL or MongoDB would provide data durability across browser sessions and support simultaneous multi-device access with server-side conflict detection using proper database transaction isolation. Second, implementing optimistic concurrency control for simultaneous booking attempts by multiple patients would prevent race conditions that could allow double-booking in a multi-user server deployment. Third, adding a full audit log recording every state transition with timestamp and actor identity would provide the accountability trail required for clinical and administrative record-keeping. Fourth, implementing data export functionality allowing the Admin to download appointment records as CSV files would provide a backup mechanism and enable integration with external reporting tools. Fifth, extending the data model to support recurring appointments and doctor leave management would handle clinic scheduling scenarios not addressed by the current single-appointment model.

Acknowledgment

I would like to sincerely thank Ravi Kumar Chaudhary, Assistant Professor, Department of Computer Science and Engineering, Raffles University, for his valuable guidance, continuous support, and helpful suggestions throughout this project.

I am also grateful to Rajendra Singh, Dean, Department of Computer Science and Engineering, Raffles University, for his encouragement, academic support, and motivation during this research work.

REFERENCES

[1] Central Bureau of Health Intelligence, National Health Profile. Ministry of Health and Family Welfare, Government of India, 2022. [Online]. Available: https://cbhi.nic.in

[2] B. Chaudhry, J. Wang, S. Wu, M. Maglione, W. Mojica, E. Roth, S. C. Morton, and P. G. Shekelle, "Systematic review: Impact of health information technology on quality, efficiency, and costs of medical care," Annals of Internal Medicine, vol. 144, no. 10, pp. 742–752, 2006.

[3] J. E. Hopcroft, R. Motwani, and J. D. Ullman, Introduction to Automata Theory, Languages, and Computation, 3rd ed. Pearson Education, 2006.

[4] D. Crockford, JavaScript: The Good Parts. O'Reilly Media, 2008.

[5] D. Flanagan, JavaScript: The Definitive Guide, 7th ed. O'Reilly Media, 2020.

[6] D. F. Ferraiolo and R. Kuhn, "Role-based access control," in Proc. 15th National Computer Security Conference, pp. 554–563, 1992.

[7] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, "Role-based access control models," IEEE Computer, vol. 29, no. 2, pp. 38–47, 1996.

[8] D. F. Sittig and H. Singh, "A new sociotechnical model for studying health information technology in complex adaptive healthcare systems," Quality and Safety in Health Care, vol. 19, suppl. 3, pp. i68–i74, 2010.

[9] E. Marcotte, "Responsive web design," A List Apart, vol. 306, 2010. [Online]. Available: https://alistapart.com/article/responsive-web-design/

[10] M. S. Mikowski and J. C. Powell, Single Page Web Applications: JavaScript End-to-End. Manning Publications, 2013.

[11] World Wide Web Consortium, "HTML Living Standard," 2023. [Online]. Available: https://html.spec.whatwg.org/

[12] Mozilla Developer Network, "JavaScript Array Methods," 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

[13] Public Health Foundation of India, "Digital Health in India: Current Landscape and Future Directions," PHFI Publications, 2021.