Secure Smart Parking Management System

Prof. Pallavi Sambhare[1], Prof. Sonali Guhe[2]
[1]*Department of DIC, G H Raisoni College of Engineering, Nagpur, India*
[2]*Dept. of Information Technology, G H Raisoni College of Engineering, Nagpur, India*

***Abstract*—This paper presents a Secure Smart Parking Management System implemented as a lightweight academic web prototype. The system uses Python Flask for request handling, SQLite for persistence, and HTML/CSS templates for the user interface. Its functional scope includes account registration, authentication, parking-slot display, booking creation, booking cancellation, administrator functions, and activity logging. The design treats foundational web-application security as a core requirement through password hashing, server-side validation, parameterized database queries, session protection, and role-based access checks. The project owner has confirmed that the application starts locally and that registration, login, and dashboard access work in the available prototype. This paper does not convert implementation evidence into unmeasured claims: booking, cancellation, administrative authorization, performance, and adversarial-security outcomes require separately recorded runtime tests. The work therefore contributes a feasible, security-oriented parking workflow for academic demonstration and establishes a reproducible evaluation plan for future extension.**

***Index Terms*—Smart Parking, Flask, SQLite, Web Application Security, Authentication, Password Hashing, Input Validation, Parameterized Queries, Role-Based Access Control, Activity Logging.**

I. INTRODUCTION

Parking management is a recurring operational problem in colleges, offices, residential complexes, and other shared facilities. Drivers need to know whether a slot can be used, while administrators need dependable records of users, allocations, and changes in availability. When the information is maintained by hand or through an unstructured register, the system can provide stale availability data, permit conflicting allocations, and make it difficult to identify the action that changed a record. A small web application can improve this workflow by providing one point of interaction for users and one controlled point of management for administrators.

The Secure Smart Parking Management System was developed as a local academic prototype rather than a production parking service. It manages sample slot records, maintains user accounts, and records bookings in a local database. The application separates normal-user tasks from administrator tasks. A normal user can register, authenticate, see the dashboard, attempt to book an available slot, cancel an active booking, and review recent booking records. An administrator can inspect the current slots, users, bookings, and selected activity logs and can change a slot state when doing so does not conflict with an active booking.

Security is important even in a compact prototype because the application stores account and booking information. Plain-text passwords, unsafe database construction, missing input validation, or unrestricted administration can compromise records and reduce the usefulness of the system. The project therefore incorporates security controls into its core request path instead of treating them as a later enhancement. The selected controls are intentionally lightweight: password hashing, validation on the server, parameterized SQLite operations, authenticated sessions, role checks, and selected activity records. The objective of the study is not to claim a complete smart-city deployment. It is to demonstrate how a manageable parking workflow can be combined with practical web-security controls within the time and infrastructure limits of a minor project. The paper describes the relevant background, requirements, architecture, data model, security design, implementation, and verification boundary. It also records which aspects of the current workflow have been observed and which must still be verified through genuine test evidence.

II. LITERATURE REVIEW

A. Security and Privacy in Smart Parking

Smart-parking research has progressed from simple vacancy detection to systems that consider privacy, secure communication, data integrity, and the constraints of Internet-of-Things devices. Chatzigiannakis, Vitaletti, and Pyrgelis [1] describe a privacy-preserving smart parking approach based on elliptic-curve security mechanisms. Their work illustrates that parking information can expose sensitive user and location details and that protected communication must be considered alongside operational convenience. The complexity of such cryptographic systems, however, makes them unsuitable as a direct implementation target for a small classroom prototype.

Al-Turjman and Malekloo [2] survey smart parking in IoT-enabled cities and show the diversity of sensing, communication, and decision-support architectures. Their survey provides an important context for the present work: an effective system may include sensors, cloud services, mobile applications, and data analysis, but a smaller prototype can isolate the fundamental workflow and still provide useful learning value. This study uses simulated slot records while retaining physical occupancy detection as a future direction.

B. Advanced Architectures and Their Trade-Offs

Edge and cloud approaches can improve responsiveness and support distributed services. Lee et al. [3] review secure and energy-efficient smart parking enabled by edge computing. Baza et al. [4] investigate privacy-preserving blockchain-assisted parking, and Yu [5] presents an edge-cloud-dew architecture intended to retain local capability during connectivity disruption. These studies motivate secure, resilient, and scalable designs; however, they introduce infrastructure, configuration, and evaluation requirements that exceed the scope of a local Flask and SQLite application.

RFID-based verification provides another way to associate vehicles or users with parking activity.

Dhetryansah et al. [6] discuss the development of smart parking using RFID sensors for online verification.

The present project does not include RFID hardware, ultrasonic sensors, or real-time sensor ingestion. Instead, it focuses on a controlled software workflow that may later integrate hardware events through a validated backend interface. This choice avoids reporting sensor accuracy or real-time detection capability when neither has been measured.

C. Research Positioning

The review identifies a gap between sophisticated research architectures and beginner-friendly implementations that explicitly demonstrate common web-application security practices. The present system occupies this gap. Its contribution is not a new detection algorithm, cryptographic protocol, or scalability benchmark. Its contribution is the integration of a small parking-management workflow with traceable security decisions: passwords are stored as hashes, inputs are validated before persistence, query values are parameterized, sensitive actions use access guards, and selected events can be reviewed through activity logs.

D. Design Implications for the Current Work

The studies reviewed above influence the project even where their full technologies are not implemented. Privacy-focused work motivates avoiding the storage of passwords in plain text and limiting logs to selected action details. IoT and edge work motivates a component-based backend that can later accept sensor events without rebuilding the account and booking modules. Blockchain and advanced cryptographic work highlight the value of integrity and traceability, which the prototype approaches at a smaller scale through controlled state changes and activity records. Thus, the literature is used to justify design priorities rather than to claim that the present system provides the same automation, scalability, or security guarantees as the cited systems.

III. PROBLEM STATEMENT AND OBJECTIVES

The problem addressed by this work is the absence of a simple, controlled mechanism for organizing parking-slot information and bookings while applying basic safeguards to user and system data. A manual or weakly protected process can make it difficult to establish current availability, prevent overlapping active bookings, distinguish normal users from administrators, or identify the records affected by important actions.

The system is designed to provide a compact answer to these workflow and security concerns.

1. Provide a browser-based interface through which users can register, authenticate, and view parking-slot information.
2. Maintain structured records for users, parking slots, bookings, and activity events using a local SQLite database.
3. Permit booking only when a requested slot exists, is marked available, and has no other active booking.
4. Restrict protected functions through login checks and reserve administration functions for the administrator role.
5. Apply password hashing, server-side validation, parameterized queries, session settings, and selected logging as foundational controls.
6. Define an evaluation procedure that records actual inputs, expected behaviour, observed behaviour, and evidence instead of relying on assumed pass results.

A. Functional and Quality Requirements

The functional requirements derive from the user roles and parking workflow. The system must permit a new user to create an account with valid input, permit an existing user to sign in with the correct credentials, display a coherent slot view to an authenticated user, and create a booking only when the requested slot is eligible. It must support cancellation of an active booking by its owner, restrict administration functions to the administrator role, and retain selected event records. The system must also provide controlled feedback for invalid requests so that a user knows whether an action was accepted, rejected, or requires a different step.

The quality requirements are intentionally modest but explicit. The prototype should be understandable to a student user, preserve essential records across application restarts, avoid storing passwords in readable form, and protect routes that should not be open to every visitor. It should use a simple local environment so that the working model can be demonstrated without a separate database server. Performance, fault

tolerance, high concurrency, and formal security assurance are not quality results of this study because they have not been measured under a defined workload or threat model.

The goals above establish a boundary for the work. Online payment, SMS or email notifications, automatic reservation windows, multi-factor authentication, high-availability deployment, and large-scale analytics are outside the initial implementation. Their exclusion makes the project feasible but means that the prototype must not be described as a production parking platform.

IV. PROPOSED METHODOLOGY AND SYSTEM ARCHITECTURE

A. Development Method

The project follows an incremental development method. The first stage identifies the users, permitted actions, parking-slot states, and security requirements. The second stage defines the browser interface, Flask routes, database schema, and authorization logic. The third stage implements the account, dashboard, booking, cancellation, and administration modules. The final stage executes functional and security-oriented tests and documents only the outcomes actually observed in the running build.

B. Logical Request Flow

The request flow begins in the browser interface. A user submits a registration, login, booking, cancellation, or administration request. The Flask backend receives the request and evaluates the relevant validation, authentication, and authorization conditions. If the conditions are satisfied, the backend uses parameterized SQLite operations to read or update the required records. Selected successful actions are

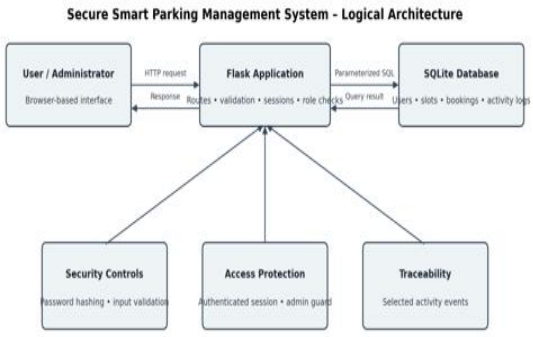


Fig. 1. Logical architecture of the secure smart parking management system.

Written To the Activity Log. The Browser is then redirected to the dashboard or administration view

with an outcome message. This design centralizes the rules that govern the parking workflow.

Step 1. Browser interface collects a valid action request and sends it to the corresponding Flask route.

Step 2. The backend checks request values and determines whether an authenticated session is required.

Step 3. For protected actions, the backend verifies the user role and the applicable parking or booking rule.

Step 4. SQLite persists or retrieves the relevant user, slot, booking, or log record through a parameterized operation.

Step 5. The system returns a controlled success, warning, or error message without exposing passwords or database-query details.

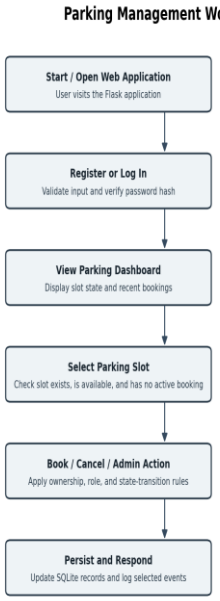


Fig. 2. User, booking, and administration workflow.

C. Data Model

The Users entity includes an identifier, a display name, a unique username, a password hash, a role, and a creation timestamp. Parking Slots includes an identifier, a unique slot code, and a state constrained to available, occupied, or maintenance. Bookings link a user to a slot and store an active, cancelled, or completed state with booking and end timestamps. Activity Logs stores selected user actions, a limited detail field, and a creation timestamp. Foreign-key enforcement is enabled when the application opens its SQLite connection.

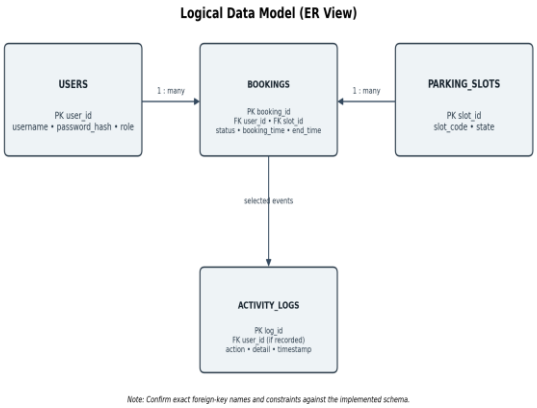


Fig. 3. Logical entity-relationship view of the application data.

D. State and Booking Rules

The design uses two related states: the state of a parking slot and the state of a booking. A slot may be available, occupied, or under maintenance. A booking may be active, cancelled, or completed. These state values are constrained in the database schema so that unintended labels cannot be persisted through the normal application code. The dashboard counts a slot as available only when its slot state is available and no active booking is associated with it. This separation allows a booking history to be retained after a cancellation while avoiding the mistaken conclusion that a cancelled record continues to reserve the slot. Before an active booking is inserted, the booking route reads the requested slot and searches for an existing active booking. If the slot does not exist, is not available, or has an active booking, the request is redirected with a warning rather than changing data. Cancellation similarly checks the booking state and the relationship between the booking and the current user. Administrative changes use an additional rule: a slot must not be made occupied or under maintenance while an active booking remains open. These rules provide a transparent basis for functional tests.

V. SECURITY DESIGN

A. Credential Handling

The registration route accepts a name, a username, and a password. Before insertion, it checks name length, restricts usernames to letters, digits, and underscores, and requires a minimum password length. Passwords are processed using Werkzeug password-hashing

functions and stored in a password_hash field rather than in plain text. During login, the submitted password is evaluated against the stored hash; it is not recovered from the database or placed in the activity log.

B. Input Validation and Database Access

The backend performs validation in the server-side route handlers rather than assuming that client-side behaviour is trustworthy. The booking route verifies that a requested slot exists and checks for an active booking before inserting another record. The administrative route validates requested slot states and blocks an update that would conflict with an active booking. SQLite statements use placeholders for submitted values, reducing the risk that untrusted input changes query structure. These implementation choices need runtime testing with representative invalid and SQL-like input before they can be claimed as verified security-test outcomes.

C. Sessions, Authorization, and Logs

The application maintains the authenticated user identifier, name, username, and role in the session. Its session cookie configuration specifies HttpOnly and SameSite=Lax. A login-required guard redirects unauthenticated requests to the login page, and an admin-required guard rejects users whose role is not administrator. The logging function records selected login, logout, booking, cancellation, and slot-status events. The intended purpose of these records is traceability within the prototype, not a replacement for centralized security monitoring or tamper-resistant audit infrastructure.

D. Development Boundaries

The available source is configured for a local development demonstration. Before an external deployment, the default development secret must be replaced with a unique PARKING_SECRET_KEY, the seeded administrator credential must be changed, debug mode must be disabled, and traffic must be protected by HTTPS behind an appropriate production server. These actions are necessary deployment controls; they are not represented as completed by the present prototype evidence.

E. Security Scope and Threat Considerations

The selected controls address common risks relevant to a small database-backed web application: unintended disclosure of stored passwords, malformed registration data, attempts to access protected pages without a session, attempts by ordinary users to use administrator functions, and unsafe treatment of user input in database operations. The controls do not constitute a complete threat model. The prototype has not been subjected to source-code review by an independent security assessor, vulnerability scanning, rate-limit testing, session-hijacking testing, or penetration testing. It should therefore be described as security-conscious in design rather than security-certified in deployment.

Data minimization is also relevant. The current schema focuses on identifiers, credentials represented by hashes, slot codes, booking times, and limited activity detail. It does not require payment details, vehicle images, real-time location tracking, or sensor histories for the proposed workflow. If future versions collect those forms of data, the project will need explicit retention rules, access controls, consent considerations, and a review of applicable institutional and legal requirements.

VI. IMPLEMENTATION

A. Application Modules

The user-management module provides registration, authentication, and logout. The dashboard module queries all slots, computes available, occupied, and maintenance counts, and retrieves a user's recent bookings. The booking module creates one active booking only when the parking rule permits it. The cancellation module changes an active booking to cancelled and records the end time. The administrative module displays users, slots, bookings, and recent logs and permits authorised slot-state changes. The activity module records selected actions after a successful state change.

B. Technology Environment

Python provides the application language, and Flask provides routing, request handling, template rendering, sessions, redirects, and user messages. SQLite provides a lightweight embedded database that is appropriate for a small local demonstration because it avoids separate server deployment. HTML and CSS templates present the registration, login, dashboard, and administration pages. Werkzeug provides password-hashing functions. The prototype does not

depend on a sensor dataset, training dataset, or machine-learning framework because its current purpose is management and security workflow rather than occupancy prediction.

C. Parking States and Rules

At initialization, the application creates twelve slot codes from A-01 through A-12 when no slot data exists. A slot can be available, occupied, or under maintenance. The dashboard treats a slot as available only when its state is available and it has no active booking. This prevents a second active booking from being created for the same slot through the normal route. A cancellation is allowed when the booking is active
and belongs to the current user, or when the user is an administrator. An administrator must resolve an active booking before marking the relevant slot as occupied or under maintenance.

D. Feedback and Error Handling

The user interface is designed to return concise feedback after normal workflow events. Registration reports whether an account was created or whether a supplied field failed validation. Login reports invalid credentials without revealing whether a particular username exists. Booking requests report when a selected slot cannot be used, and cancellation reports whether a matching active booking was found. Administrative updates report invalid state values, missing slots, and conflicts caused by active bookings. These responses are intended to help the user correct a normal workflow error while avoiding disclosure of raw database statements, passwords, or debugging details.

During a local development demonstration, a developer must still handle errors responsibly. The use of debug mode is convenient while building the prototype, but it is not suitable for a public deployment because debug information can reveal implementation detail. This distinction is included in the paper so that the working local website is not mistaken for a hardened public service.

VII. VERIFICATION STATUS AND EVALUATION PLAN

A. Observed Prototype Workflow

Table I. Summary of Confirmed Prototype Evidence

Workflow item	Status	Evidence boundary
Local application startup	Observed	Project owner confirmed local startup.
User registration	Observed	Registration completed in the running prototype.
User login	Observed	Successful authentication reported.
Dashboard access	Observed	Dashboard access reported after login.
Booking / cancellation	Not yet verified	Runtime test evidence not supplied.
Admin access / security tests	Not yet verified	Runtime test evidence not supplied.
Performance / load	Not measured	No timing or workload measurements supplied.

The confirmed outcomes are limited to the four observed workflow items. The remaining functions are implementation paths or planned test cases, not verified experimental results. Accordingly, no accuracy, response-time, throughput, reliability, or security-effectiveness value is reported.

The confirmed workflow evidence available at the time of writing is limited to local startup, user registration, successful login, and dashboard access. These are recorded as observed actions because the project owner reported completing them in the running Flask application. The availability of source-code routes for booking, cancellation, activity logging, and administration shows that the corresponding implementation paths exist, but source inspection alone is not a substitute for a captured runtime result.

B. Required Functional Tests

1. Book an available slot and capture the slot, success message, and recent-booking record.
2. Attempt a second active booking for the same slot and record whether the application rejects it.
3. Cancel an active booking and confirm that its status changes and an end time is recorded.

4. Attempt to access administration as a normal user and record the actual access response.
5. Log in as the authorised administrator, update a slot state without an active booking, and confirm the resulting record and activity-log entry.
6. Submit invalid registration values and SQL-like text where appropriate, then record the observed validation behaviour without interpreting it as a security guarantee.

C. Measurement Discipline

Each test should record the date, build version, device and operating system, input, expected behaviour, observed behaviour, and screenshot or log reference. A pass percentage should be calculated only after all selected test cases are executed. Response-time, concurrency, load, usability, and penetration-testing claims require their own documented measurement methods and raw observations. The paper therefore reports no numerical performance, reliability, detection accuracy, or security-effectiveness result.

D. Evidence Record Format

A practical evidence record can use the following sequence for each test: identify the test case and precondition; record the account role and initial slot or booking state; enter the exact input or action; state the expected behaviour derived from the requirements; capture the observed screen or database-facing outcome; and mark the test as passed, failed, or inconclusive only after reviewing that evidence. The record should also state the tester and date. This approach prevents a screenshot from being detached from its test condition and prevents an expected behaviour from being presented as an observed result. For security-oriented checks, the same discipline is important. A route guard may be present in source code, yet its runtime behaviour must still be observed for an unauthenticated or non-administrator request. A parameterized query may be used in the implementation, yet a selected validation case must be recorded before saying that the application safely handled that input. The goal is not to make broad security claims from a small demonstration; it is to document precisely what the prototype was asked to do and how it responded.

VIII. LIMITATIONS AND FUTURE SCOPE

The current system is a local software prototype with simulated slot states. It does not automatically identify whether a vehicle is present, connect to physical sensors, offer payment handling, send messages, provide a mobile application, or operate as a deployed service. SQLite is practical for a small prototype but is not intended as a claim of large-scale concurrent deployment. In addition, the documented evidence does not include a completed booking/cancellation test set, timing study, load study, or external security assessment.

Future development can integrate ultrasonic sensors and a Raspberry Pi for occupancy detection, add RFID or QR-based vehicle verification, provide notifications, use HTTPS and a production database, introduce stronger administrator safeguards, and add automated functional tests. Historical bookings could later support data analysis, provided that data protection, consent, access control, and measurement methods are designed explicitly. These extensions should be evaluated with genuine experimental data rather than assumed values.

IX. CONCLUSION

The Secure Smart Parking Management System demonstrates a feasible academic approach to combining a simple parking workflow with basic web-application security practices. Its implemented architecture uses a browser interface, Flask application logic, and SQLite persistence to support account management, parking-slot information, bookings, cancellation, administration, and activity records. The design incorporates password hashing, validation, parameterized queries, sessions, and role checks. The confirmed local workflow currently covers startup, registration, login, and dashboard access. Remaining booking, administrative, and security-related checks have been documented as a reproducible test plan rather than claimed as results. This evidence-based boundary keeps the study suitable for academic submission and creates a clear foundation for future implementation and evaluation.

REFERENCES

[1] Chatzigiannakis, A. Vitaletti, and A. Pyrgelis, "A privacy-preserving smart parking system using an IoT elliptic curve-based security platform," Computer Communications, vols. 89–90, pp. 165–177, 2016, doi: 10.1016/j.comcom.2016.03.014.

[2] Al-Turjman and A. Malekloo, "Smart parking in IoT-enabled cities: A survey," Sustainable Cities and Society, vol. 49, Art. no. 101608, 2019, doi: 10.1016/j.scs.2019.101608.

[3] P. Lee, F. T. J. Leng, R. A. A. Habeeb, M. A. Amanullah, and M. H. U. Rehman, "Edge computing-enabled secure and energy-efficient smart parking: A review," Microprocessors and Microsystems, vol. 93, Art. no. 104612, 2022, doi: 10.1016/j.micpro.2022.104612.

[4] M. Baza et al., "Privacy-preserving blockchain-assisted private-parking scheme with efficient matching," Computers and Electrical Engineering, vol. 103, Art. no. 108340, 2022, doi: 10.1016/j.compeleceng.2022.108340.

[5] Y.-C. Yu, "Smart parking system based on edge-cloud-dew computing architecture," Electronics, vol. 12, no. 13, Art. no. 2801, 2023, doi: 10.3390/electronics12132801.

[6] Dhetryansah et al., "Development of smart parking systems using RFID sensors for online verification in different areas," IOTA, vol. 3, no. 2, pp. 139–147, 2023, doi: 10.31763/iota.v3i2.587.

[7] The diagrams summarize the architecture and data relationships described in the paper. They should be verified against the final source code and database schema before publication.