Parking Management System: Dual-Backend Web Architecture for Slot Tracking, Billing, and DevOps-Ready Delivery

Andrés Felipe Mateus Saavedra, Anderson Jefrey López Jiménez, Juan Esteban Oviedo Sandoval
Systems Engineering Project
Universidad Distrital Francisco José de Caldas (UDFJC)
Bogotá D.C., Colombia
Student codes: 20201020119, 20162020424, 20192020064
Emails: afmateuss@udistrital.edu.co, ajlopezj@udistrital.edu.co, jeoviedos@udistrital.edu.co

Abstract—Parking facilities require reliable control of vehicle entry/exit, space assignment, and fee calculation while providing operators with real-time indicators. We present a modular Parking Management System that separates authentication (Spring Boot with JWT) from domain logic (FastAPI with SQLAlchemy) and integrates a lightweight Web GUI for daily operation. The resulting platform supports slot-based sessions, automated billing with minute rounding, and an automation-ready delivery pipeline including unit tests, acceptance tests, containerization, and CI/CD evidence.

Index Terms—parking management, microservices, FastAPI, Spring Boot, JWT, Docker, CI/CD, software testing

I. Introduction

Parking management software is a representative transaction-oriented information system: it must register events (entries and exits), keep state consistent (occupied slots), and compute derived values (time and cost) in a way that is transparent for operators and auditable for customers. In real deployments, requirements often evolve from simple entry logs to more complete operational dashboards that aggregate occupancy, pricing, and activity, which increases the need for maintainable architecture and testable business rules.

From a software engineering perspective, this scenario is well suited to applying service separation and API-driven integration. A service-oriented approach reduces coupling compared with a monolith and makes it easier to evolve security and domain logic independently. The system exposes REST endpoints following common architectural principles [1] and uses JSON Web Tokens (JWT) to carry authentication context between requests [2]. The implementation relies on widely adopted frameworks and tools: Spring Boot for the authentication service [3], FastAPI for the core service [4], and PostgreSQL as the primary datastore for the core domain [5]. To improve reproducibility and automation, we containerize the components using Docker and Docker Compose [6], validate critical user stories with Cucumber acceptance tests [7], evaluate endpoint behavior under load using Apache JMeter [8], and integrate continuous testing and build checks with GitHub Actions [9]. This paper summarizes the final system and consolidates evidence previously documented in the Workshop 3 and 4 deliverables [10], [11].

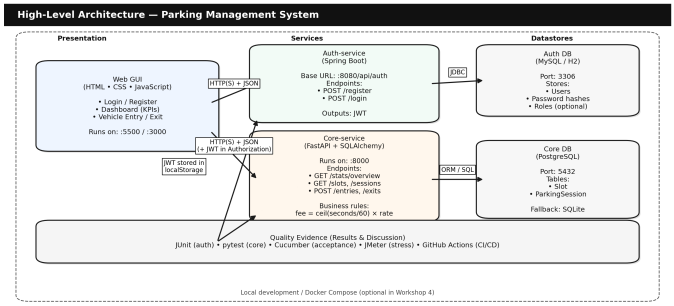

Fig. 1. High-level architecture with a Web GUI consuming separate authentication and core REST services.

TABLE I
Main components and responsibilities.

Layer	Tech	Responsibilities
Web GUI	HTML/JS	Operator views (dashboard and sessions), input validation/formatting, token storage, receipt modal.
Auth-service	Spring	User registration/login, JWT issuing, access control boundary.
Core-service	FastAPI	Slot assignment, session state, billing computation, statistics, persistence.
Database	SQL	Persist slots/sessions; maintain referential integrity and queryable history.

II. Methods and Materials

A. System Decomposition

Figure 1 illustrates the high-level architecture. The Web GUI is a thin client implemented with HTML, CSS, and vanilla JavaScript that manages the operator workflow and communicates with two backends through HTTP fetch requests. The Auth-service provides /register and /login endpoints and returns a signed JWT. The Core-service encapsulates the parking domain, including slot state, active sessions, billing, and aggregated statistics exposed through dedicated endpoints. This decomposition enables independent scaling, clearer ownership, and simpler automated testing at multiple levels.

Table I summarizes the main components and their responsibilities.

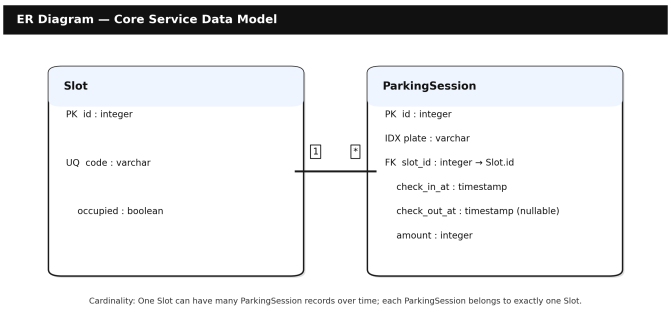

Fig. 2. Core data model (ER-style) centered on slots and parking sessions.

B. Configuration and Local Execution

For local development, each component runs on a dedicated port: the Core-service on :8000, the Auth-service on :8080, and the static Web GUI on a lightweight HTTP server (e.g., :5500). The browser stores the JWT in localStorage and attaches it as an Authorization: Bearer header in protected requests. On the backend side, configuration is environment-driven (database URL, secret keys, and allowed origins for CORS), allowing the same codebase to run in local mode, containers, or CI. This setup also supports a SQLite fallback for quick demos when a PostgreSQL instance is not available, while preserving the same API contracts.

C. Data Model and Plate Normalization

The core domain uses two primary entities: Slot and ParkingSession. A slot has an identifier, a human-readable code, and an occupancy flag; a parking session stores the plate, the assigned slot, timestamps for entry/exit, and the computed amount. Figure 2 summarizes these relationships.

Vehicle plates are normalized at the service boundary to reduce duplicates and simplify lookups. Internally, plates are stored uppercased without separators, while the GUI formats them for display as ABC-123. A shared regular expression validates the user input before requesting an entry or exit operation.

D. Core API and Business Rules

The Core-service acts as the source of truth for time and fee calculations. To keep the operator experience consistent across browsers, the GUI displays the values returned by the service, and only performs lightweight formatting (e.g., plate formatting and currency display).

Table II lists the core endpoints used by the GUI in the final iteration.

Billing is defined as a constant rate per minute:

$$\text{amount} = \text{minutes} \times \text{rate\_per\_minute}, \tag{1}$$

with minute rounding toward the next full minute:

$$\text{minutes} = \max\left(1, \left\lceil \frac{\Delta t}{60} \right\rceil\right), \tag{2}$$

TABLE II
Core-service endpoints used by the Web GUI.

Method	Endpoint	Purpose
GET	/stats/overview	KPIs (occupancy, active sessions, current rate).
GET	/slots	List slots with current occupancy and plate (if active).
GET	/sessions	Recent session history (ordering/limit).
POST	/entries	Register entry, assign a free slot, open session.
POST	/exits	Register exit, compute minutes/amount, free slot.

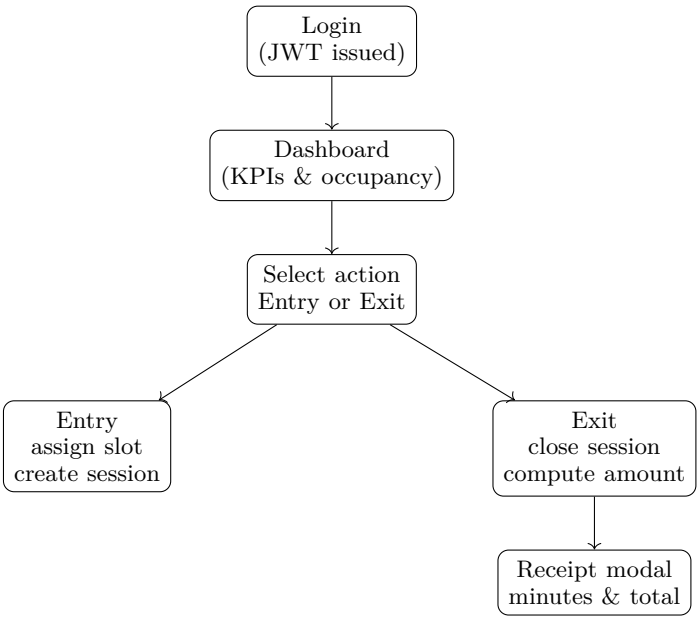

Fig. 3. Operator workflow (UI-level) for the main entry/exit actions.

where Δt is the difference between the exit and entry timestamps in seconds. This rule avoids undercharging for partial minutes and enforces a minimum of one minute.

A practical implementation detail is time handling. During development, the system faced issues caused by mixing timezone-aware and timezone-naive datetimes when persisting timestamps without explicit timezone metadata. To eliminate mismatches in local execution, the final configuration treats stored timestamps consistently as local time within the core service and avoids force-setting timezones on persisted values.

E. Operator Workflow Visualization

To document the human workflow, Fig. 3 models the main steps an operator follows. This diagram complements the service-level view in Fig. 1 by emphasizing the sequence of actions and checks performed at the UI layer.

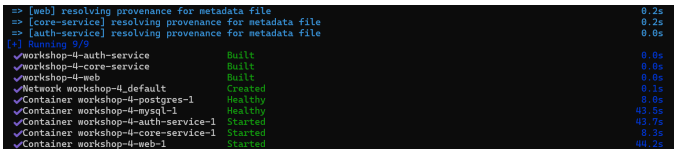

Fig. 4. Excerpt of the Docker Compose configuration orchestrating the web GUI and backends.

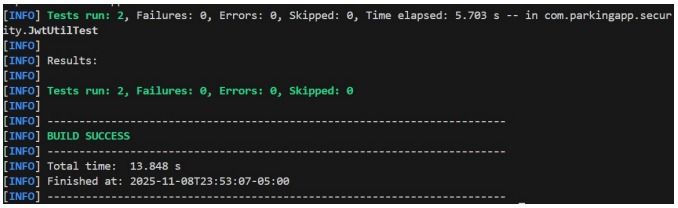

Fig. 5. JUnit execution evidence for the authentication service.

F. Containerization and Automation Tooling

The system is packaged to support repeatable local execution and automated verification. Containerization encapsulates dependencies (Java runtime, Python environment, and database drivers), while orchestration coordinates service startup and networking. Figure 4 shows an excerpt of the Compose configuration used to orchestrate the services.

III. Results & Discussion

The project was validated through unit testing, acceptance testing, stress testing, and continuous integration runs. Instead of listing full logs, we present evidence figures and interpret the observed behavior in terms of functional correctness and operational readiness.

A. Unit Testing Evidence

The authentication service includes unit tests executed with JUnit, and the core service includes unit tests executed with pytest. Figures 5 and 6 show representative passing test outputs captured during the workshop execution.

B. Acceptance Testing with Cucumber

Cucumber scenarios were used to formalize expected behavior for core user stories. In particular, the login flow and protected resource access validate that JWT handling is correct and that the system enforces an authentication boundary. Figure 9 shows a feature excerpt used to define the acceptance criteria in a readable, behavior-driven format.

C. Stress Testing with JMeter

JMeter was used to exercise critical endpoints under moderate concurrent load. The plan targets authentication and core operations (e.g., login and resource listing/registration), and collects response-time percentiles, throughput, and error rate. In the workshop configuration,

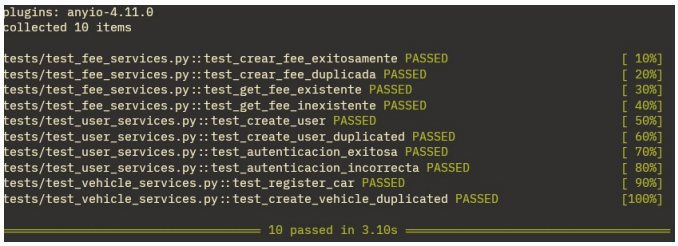

Fig. 6. pytest execution evidence for the core service.

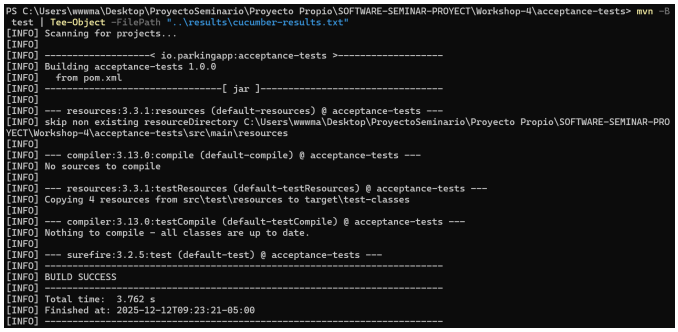

Fig. 7. Cucumber feature excerpt used to validate key user stories via REST calls.

a typical plan used 50 virtual users, 30 seconds of ramp-up, and a 3–5 minute execution window, which is sufficient to reveal slow endpoints and database bottlenecks in an academic environment.

Qualitatively, the system remains stable under moderate load, with most requests completing within a few hundred milliseconds and only occasional outliers. This behavior is consistent with a lightweight service composition where network latency and database access dominate. If the system is extended for production, future work should include stronger workload modeling (e.g., daily peaks), longer endurance tests, and profiling of slow queries.

D. CI/CD Execution Evidence

The CI workflow automates repository checkout, environment setup for Java and Python, test execution, and Docker image builds. Figure 10 shows evidence of a successful GitHub Actions run.

E. Functional Case Study: Entry–Exit Billing Consistency

A key correctness criterion is that the exit operation returns coherent values for timestamps, minutes, and total amount. In practice, the user triggers an entry for a valid plate, the system assigns a free slot and stores check_in_at, and later the exit request closes the session and returns minutes and amount computed by Eqs. (1)–(2). During development, inconsistencies were observed when mixing timezone-aware and timezone-naive timestamps; once the time representation was made consistent, the displayed receipt values matched the persisted values and operator expectations. This reinforces the

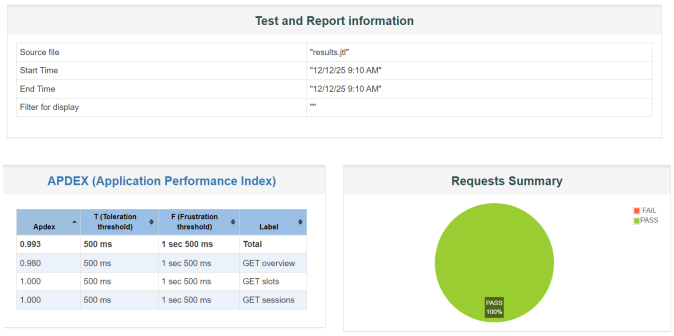

Fig. 8. Evidence of stress test results with JMeter, Test and Report information

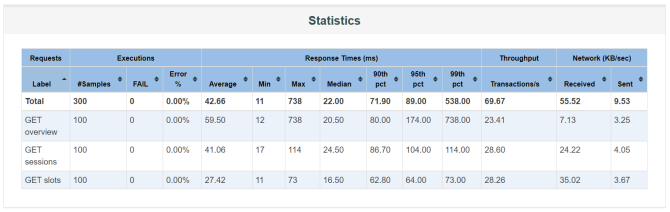

Fig. 9. Evidence of stress test results with JMeter, Statistics

decision to keep the backend as the single source of truth for billing.

F. Summary of Quality Activities

Table III consolidates the main validation activities and their goal.

IV. Conclusions

This paper presented a Parking Management System implemented as a modular, dual-backend web application with a thin Web GUI. By separating authentication from core domain logic, the solution improves maintainability and enables independent evolution of security and business rules. The core service provides slot-based session tracking and deterministic billing with minute rounding, while the interface offers a fast operator workflow and an integrated receipt view.

Beyond functional requirements, the project incorporates repeatable delivery practices through containerization and automated testing. These practices provide a foundation for future extensions such as configurable pricing policies, richer analytics and reporting, role-based access control, and production-grade deployment. A key limitation is that current performance evaluation is scoped to moderate loads and a limited set of scenarios; therefore, broader workload modeling and additional negative-path tests remain important future work.

Acknowledgments

The authors thank Eng. Carlos Andrés Sierra, M.Sc., for guidance during the Software Engineering Seminar course.

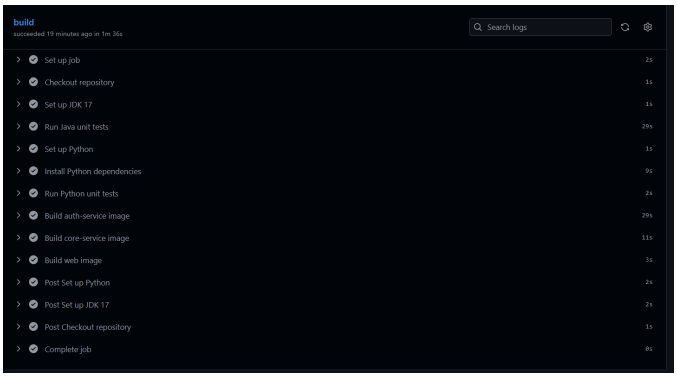

Fig. 10. Evidence of a successful CI workflow run validating builds, tests, and Docker images.

TABLE III
Quality assurance activities and evidence.

Artifact	Scope	Goal and Evidence
JUnit tests	Auth-service	Validate authentication logic; passing run captured (Fig. 5).
pytest tests	Core-service	Validate session and billing logic; passing run captured (Fig. 6).
Cucumber	End-to-end	Validate user stories and JWT boundary; feature excerpt shown (Fig. 9).
JMeter	Performance	Exercise REST endpoints under load; configuration and interpretation summarized.
GitHub Actions	CI/CD	Automate build/test/image checks; successful run shown (Fig. 10).

References

[1] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[2] M. Jones, J. Bradley, and N. Sakimura, "Json web token (jwt)," RFC 7519, 2015.

[3] "Spring boot reference documentation," VMware, 2025, accessed 2025-12-11.

[4] "Fastapi documentation," FastAPI, 2025, accessed 2025-12-11.

[5] "Postgresql documentation," PostgreSQL Global Development Group, 2025, accessed 2025-12-11.

[6] "Docker documentation," Docker, 2025, accessed 2025-12-11.

[7] "Cucumber documentation," Cucumber, 2025, accessed 2025-12-11.

[8] "Apache jmeter user manual," Apache Software Foundation, 2025, accessed 2025-12-11.

[9] "Github actions documentation," GitHub, 2025, accessed 2025-12-11.

[10] A. F. Mateus Saavedra, A. J. López Jiménez, and J. E. Oviedo Sandoval, "Workshop 3 documentation: Backend implementation, testing, and web integration," Course deliverable, Software Engineering Seminar, Universidad Distrital Francisco José de Caldas, 2025.

[11] ——, "Workshop 4 documentation: Containerization, acceptance testing, api stress testing, and ci/cd integration," Course deliverable, Software Engineering Seminar, Universidad Distrital Francisco José de Caldas, 2025.