DriveMaster: A Microservices-Based Subscription Platform for Driving Academy Management in Colombia

Cristian Arturo Parra Gonzales - 20201578102
David Gerardo Díaz Gómez - 20201020087
Daniel Mateo Montoya González - 20202020098
Universidad Distrital Francisco José de Caldas
Bogotá, Colombia

***Abstract*—Driving schools in Colombia must simultaneously manage students, instructors, vehicles, and scheduling, while also ensuring compliance with strict theoretical and practical hour requirements established by the Ministry of Transport. However, many institutions still rely on manual procedures or isolated systems, which makes it difficult to maintain consistent records, increases operational workload, and leads to failures in tracking academic progress.**

DriveMaster is introduced as a centralized management platform that organizes students, staff, and classes through a unified role-based control model. It enables efficient student registration, detailed progress tracking, session scheduling, vehicle assignment, and administrative and instructional staff management, while ensuring compliance with regulatory requirements.

The solution is implemented using a microservices architecture composed of an authentication service built with Spring Boot and MySQL, and a business logic service developed with FastAPI and MongoDB. This design allows modular development, independent scaling of services, and improved maintainability. The entire system is containerized with Docker and orchestrated using Docker Compose, ensuring portability, isolation, reproducible deployments, and simplified environment configuration across development, testing, and production.

In addition, DriveMaster integrates continuous integration pipelines with comprehensive unit, acceptance, and performance testing, validating functionality, reliability, and stability under realistic workloads. The platform demonstrates a scalable, secure, and maintainable approach to modernizing academic and operational management in driving schools, providing a practical blueprint for the digital transformation of regulated educational institutions.

I. Introduction

Driving schools in Colombia operate under strict regulations defined by the Ministry of Transport and the RUNT system, which require precise tracking of theoretical and practical training hours for each license category. Despite these regulatory demands, many institutions still rely on manual processes, spreadsheets, or disconnected software tools to manage students, instructors, vehicles, and academic progress. These limitations often result in inconsistent records, human errors, delays in certification, and difficulty ensuring compliance with national standards. As the number of enrolled students continues to grow, the operational burden increases proportionally, exposing the need for a centralized, reliable, and scalable management solution.

Beyond regulatory obligations, driving schools face additional challenges associated with the coordination of academic and logistical resources. Managing schedules requires aligning instructor availability, vehicle assignment, and student attendance, a process that becomes increasingly complex without automated tools. Furthermore, administrative personnel must register new students, track academic progress, validate attendance, manage payments, and generate certificates. Without an integrated information system, these activities become repetitive, error-prone, and time-consuming, ultimately affecting institutional productivity, transparency, and the overall quality of service delivered to students.

These challenges reveal a broader structural problem: most driving academies lack modern digital platforms capable of supporting real-time monitoring, multi-role coordination, and automated regulatory validation. Existing solutions, when available, tend to be isolated, outdated, or unable to scale to multiple branches or institutions. Consequently, there is a growing need for technological platforms that not only centralize operations but also provide secure authentication, robust data management, and reliable communication between components.

To address these challenges, this project proposes *DriveMaster*, a Subscription-as-a-Service (SaaS) platform built under a microservices architecture to centralize the academic and operational management of driving schools. The system integrates a Spring Boot authentication service backed by MySQL, a FastAPI business logic service supported by MongoDB, and a role-based Angular front-end that enables the coordinated management of students, employees, classes, and certification workflows. The platform leverages containerization through Docker and orchestration with Docker Compose to ensure portability and modular deployment, while continuous integration pipelines and automated tests strengthen reliability, maintainability, and development efficiency.

DriveMaster is designed with scalability, security, and maintainability as core principles. The microservices architecture allows independent development, testing, and deployment of each component, facilitating continuous improvement without affecting the entire system. JWT-based authentication, role-based access control, and strict CORS policies secure sensitive data and ensure compliance with privacy standards. The system also implements automated logging, monitoring, and health checks to maintain high availability and operational resilience.

This paper presents the design, implementation, and evaluation of DriveMaster as a scalable and modernized solution for driving academies in Colombia. It demonstrates the potential of microservice-based architectures to digitalize and optimize regulated educational environments, providing a blueprint for future systems that aim to combine operational efficiency, regulatory compliance, and high-quality educational service delivery.

II. Methods and Materials

A. Software Architecture

DriveMaster follows a microservices architecture that promotes modularity, scalability, and independent service deployment. This architectural choice separates responsibilities across three major components, allowing each service to evolve, scale, and be tested independently. Such separation minimizes coupling and ensures that failures in one component do not compromise the entire platform. Additionally, the architecture supports continuous integration and facilitates maintenance by isolating the concerns of authentication, business logic, and presentation.

- **Authentication Service (Spring Boot)**: Responsible for user identity and access management. It handles registration, login, password encryption using BCrypt, and JWT-based authentication to ensure stateless and secure communication across the system. The service interacts with a MySQL relational database to store user credentials, roles, and permission metadata, enabling consistent access control across multiple academies. By centralizing authentication, the system enforces strong security practices and reduces redundancy in other services.
- **Business Logic Service (FastAPI)**: Implements the core operational workflows of the platform, including scheduling, instructor assignment, vehicle management and student progress tracking. Built on FastAPI for high performance and low latency, this service uses MongoDB as a flexible, document-oriented persistence layer capable of adapting to evolving data structures related to sessions, progress records, and operational logs. This microservice encapsulates all domain-specific behavior, ensuring that rules—such as required training hours or session completion criteria—are consistently applied across the platform.
- **Frontend (Angular + Nginx)**: Provides role-based interfaces tailored for students, instructors, secretaries, and administrators. The Angular application implements reactive UI components and communicates with both backends through RESTful APIs, ensuring clear separation between presentation and data layers. The frontend is deployed behind Nginx, which improves performance by serving static assets efficiently and acting as a reverse proxy. This design supports smooth navigation, structured workflows, and a consistent user experience across devices and organizational roles.

Overall, this architectural distribution allows *DriveMaster* to achieve fault isolation, independent scalability, and maintainability, while ensuring efficient communication between components. Each service remains autonomous but interoperable, enabling the platform to grow in functionality without compromising performance or structural integrity.

Figure 2 illustrates the software architecture and communication workflow.

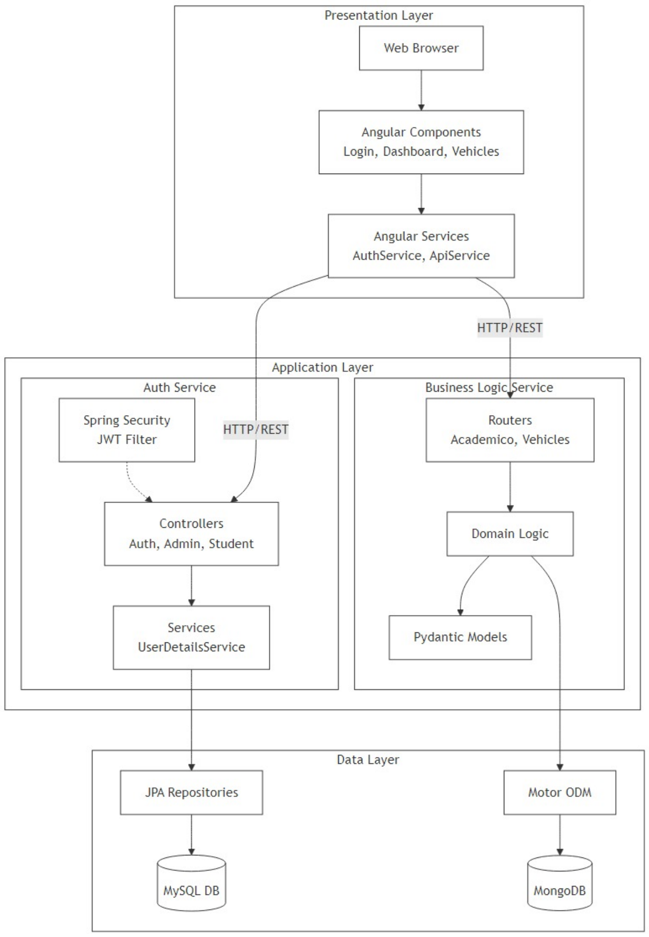

Fig. 1. Software architecture of DriveMaster showing separation between authentication, business logic, and presentation layers.

B. System Architecture

All services are containerized using Docker and orchestrated through Docker Compose, allowing the platform to be deployed consistently across different environments regardless of the underlying operating system or hardware configuration. The deployment consists of the following containers:

- Angular frontend served with Nginx
- Java authentication service

- FastAPI business service
- MySQL authentication database
- MongoDB operational database

Each service communicates through a dedicated Docker bridge network, ensuring both isolation and secure service discovery. This network configuration prevents cross-container interference while enabling controlled interactions between services through explicitly defined endpoints. In addition, the use of individual Dockerfiles for each service allows fine-grained control over dependencies, build steps, and runtime configurations, reducing image size and minimizing potential inconsistencies during deployment.

Persistent storage is managed through Docker volumes, ensuring that database data remains available even if containers are recreated. Environment variables are used to provide runtime configuration for credentials, ports, and internal URLs, enabling seamless adaptation to development, testing, and production environments. Health checks defined in Docker Compose automatically verify that each service is running as expected, allowing dependent containers to start only when their prerequisites are ready. Together, these architectural decisions contribute to a modular, portable, and maintainable deployment strategy suited for real-world institutional use.

C. Database Design

A hybrid persistence model was chosen to accommodate the different data requirements of the platform. This approach combines a relational database for structured and transactional information with a document-based database for operational data that changes frequently and requires flexible schemas:

- **MySQL**: Stores normalized relational data such as users, roles, authentication metadata, and other security-related entities. Its transactional guarantees ensure consistency during login, registration, and role-validation processes, which are core to the authentication workflow.
- **MongoDB**: Manages operational data such as schedules, class sessions, vehicle logs, instructor assignments, and student progress. These entities evolve dynamically as class sessions are created, updated, or completed, making MongoDB's flexible document structure ideal for supporting rapid iteration and schema variation.

This separation allows transactional consistency for user management while providing adaptable schemas for operational entities that evolve frequently. By isolating authentication and identity information in a relational model, the system ensures data integrity and strict access control, whereas operational workflows benefit from the scalability and fluidity of a NoSQL model. Additionally, this hybrid architecture reduces coupling between services, as each backend component interacts only with the database model best suited to its needs. This design not only improves performance and maintainability but also prepares the system for future extensions, such as handling more complex scheduling rules or supporting multi-academy deployments.

D. Service Communication

Services communicate using RESTful APIs and JSON serialization, providing a lightweight and platform-independent mechanism for exchanging data. Each service endpoint is designed following REST principles, enabling clear resource identification, standard HTTP methods, and stateless interactions. Authentication relies on stateless JWT tokens, which include user role and expiration metadata, ensuring secure access control without the need for server-side session storage. These tokens are validated by the FastAPI service for every incoming request, and any invalid or expired token results in a standardized HTTP 401 Unauthorized response.

To enhance security and reduce attack surface, CORS restrictions are configured to allow requests only from the official frontend domain in production environments. This prevents unauthorized cross-origin requests and ensures that sensitive operations are executed only through trusted clients. Additionally, error handling follows standardized HTTP response models, providing consistent and predictable feedback for both client and service developers. Each service returns structured error objects containing a status code, error type, and descriptive message, which facilitates debugging, monitoring, and integration with client-side error reporting tools.

Furthermore, services are designed to support extensibility and maintainability. Versioning of APIs is implemented to allow backward-compatible changes and gradual feature rollouts. Logging and monitoring hooks are embedded at key communication points, capturing request metadata and response status, which aids in performance analysis and early detection of anomalies within the system.

E. Containerization and Deployment

Each service is containerized using a dedicated Dockerfile with multi-stage builds, which optimizes image size and reduces build complexity. Multi-stage builds separate the build environment from the runtime environment, ensuring that only necessary artifacts are included in the final image, improving security and performance. Docker Compose orchestrates the deployment of all services, handling service dependencies, health checks, persistent storage, and environment-based configuration in a unified manner.

- **Service Dependencies:** Docker Compose ensures that services are started in the correct order based on their dependencies, preventing runtime errors caused by unavailable services.
- **Health Checks:** Each service defines custom health checks that are periodically executed to verify operational status. Containers that fail these checks can be automatically restarted or flagged for inspection.
- **Persistent Storage:** Volumes are configured for MySQL and MongoDB to persist data across container restarts, ensuring data durability while maintaining the benefits of ephemeral containers.
- **Environment-based Configuration:** Environment variables and configuration files allow each service to adapt

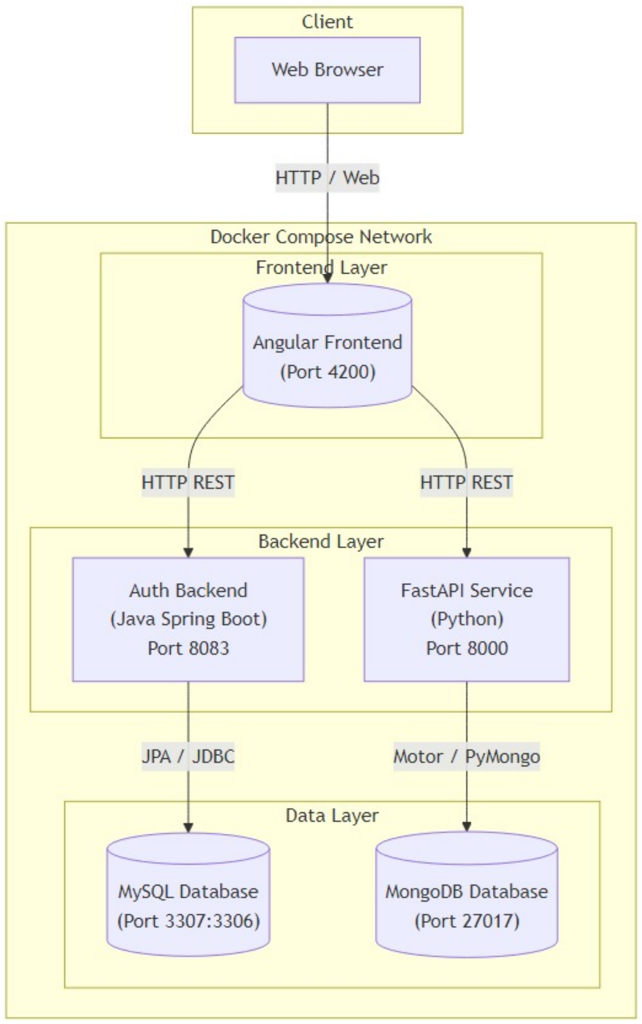
Client
Web Browser
HTTP / Web
Docker Compose Network
Frontend Layer
Angular Frontend (Port 4200)
HTTP REST
HTTP REST
Backend Layer
Auth Backend (Java Spring Boot) Port 8083
FastAPI Service (Python) Port 8000
JPA / JDBC
Motor / PyMongo
Data Layer
MySQL Database (Port 3307:3306)
MongoDB Database (Port 27017)

Fig. 2. Software architecture of DriveMaster showing separation between authentication, business logic, and presentation layers.

to different deployment environments (development, testing, staging, production) without modifying the container image itself.

Database initialization scripts automatically create schemas and seed test data when containers are launched, streamlining development and testing workflows. Additionally, the deployment strategy supports scalability by allowing containers to be distributed across multiple hosts, enabling load balancing and fault tolerance. Logging and monitoring are integrated within containers to capture runtime metrics, errors, and resource utilization, facilitating proactive maintenance and rapid troubleshooting.

F. Testing Strategy

The testing process for DriveMaster was carefully designed to validate correctness, reliability, and performance across all system components. A comprehensive, multi-layered strategy was implemented to ensure that both backend services and the frontend interface operate according to functional specifications, maintain data integrity, and deliver acceptable performance under expected workloads. The testing approach encompassed five main categories:

- **Java Unit Tests:** Implemented using JUnit and Mockito, these tests focus on the authentication workflows, JWT token verification, password encryption with BCrypt, and role-based access control. Each test isolates individual classes and methods to confirm that logic behaves as expected under a variety of scenarios, including edge cases such as expired tokens or invalid credentials.
- **Python Unit Tests:** Implemented using pytest and mocked MongoDB clients, these tests validate the business logic of the FastAPI service. Critical functionalities covered include scheduling algorithms, session state transitions, vehicle assignment and status management, and student progress calculations. By mocking database interactions, tests remain fast, reproducible, and independent of external dependencies.
- **Acceptance Tests:** Written in Gherkin and executed with Cucumber, acceptance tests validate end-to-end scenarios from the perspective of real users. Key workflows tested include user login and authentication, class scheduling, session updates, progress tracking, and certificate generation. These tests provide assurance that the system fulfills all functional requirements and aligns with the expected user experience.
- **Performance Tests:** JMeter was applied to both backend services (Spring Boot and FastAPI) to assess throughput, response time stability, and error-handling behavior under concurrent load. Performance tests simulated realistic traffic patterns and peak usage conditions to identify potential bottlenecks and ensure that the platform can support multiple simultaneous users without degradation in responsiveness.

This multi-layered testing strategy ensured coverage of functional, structural, and performance requirements prior to deployment, providing confidence in the system's correctness, stability, and scalability. Additionally, the testing framework established repeatable procedures for regression testing, facilitating continuous integration and enabling the development team to detect and resolve issues promptly during iterative releases. By combining unit, integration, acceptance, and performance tests, DriveMaster demonstrates a rigorous approach to software quality assurance that aligns with industry best practices for microservice-based architectures.

III. Results and Discussion

A. Unit Testing Results

Both backend services achieved high reliability in unit testing, obtaining execution success rates above 90% across all test cases. The Spring Boot authentication service achieved 85% code coverage and passed more than 90% of all scenarios involving JWT validation, password hashing, and role assignment.

The FastAPI backend achieved 78% structural coverage but exceeded a 90% success rate in all executed unit cases, validating key workflows such as instructor scheduling, session completion, vehicle-state transitions, and student progress updates.

B. Acceptance Testing Results

Acceptance tests written in Gherkin validated end-to-end functional workflows, including:

All implemented scenarios passed successfully, demonstrating traceability between user stories and system behavior.

C. Performance Testing

Performance tests using JMeter evaluated system behavior under concurrent load.

Java Backend (Spring Boot): A total of 561,078 requests were executed. Table I shows the distribution:

Response Code	**Fail**	**Success**	**Total**
200 (OK)	0	360,200	360,200
400 (Bad Request)	29,200	0	29,200
403 (Forbidden)	171,678	0	171,678
Total	**200,878**	**360,200**	**561,078**

TABLE I
JMETER RESPONSE DISTRIBUTION FOR THE JAVA AUTHENTICATION SERVICE.

Errors (HTTP 400 and 403) corresponded to intentionally malformed or unauthorized requests included in the test scenarios to validate the system's error-handling mechanisms. These tests ensured that the services correctly identified invalid input, denied access when appropriate, and returned standardized error responses with descriptive messages. Throughout all testing phases, both the Spring Boot and FastAPI services remained stable, with no observed downtime or crashes, demonstrating the robustness of the architecture under normal and edge-case conditions. The system consistently maintained transactional integrity and correct state management, even when faced with erroneous or malicious requests, confirming the reliability and resilience of the platform's error-handling design.

Python Backend (FastAPI): The FastAPI backend achieved a success rate above 90% under high concurrency. Figure 3 shows the obtained performance profile.

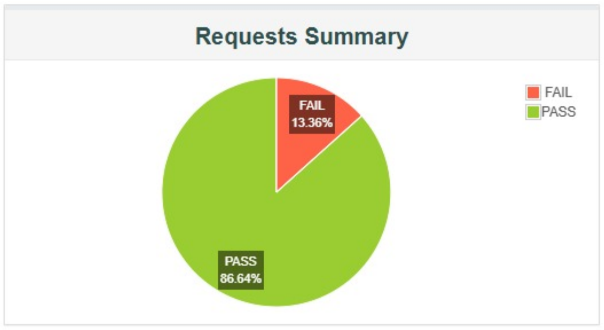

Fig. 3. JMeter performance results for the FastAPI backend.

Both services maintained consistent response times and stable throughput, validating scalability and robustness.

D. Deployment Validation

Using Docker Compose, the full platform deployed in under two minutes on a mid-range workstation. All containers passed health checks, and inter-service communication was confirmed through automated smoke tests, demonstrating reliable orchestration and modular deployment capability.

IV. CONCLUSIONS

DriveMaster effectively modernizes the operational management of driving academies in Colombia by centralizing critical processes such as student administration, instructor assignment, vehicle coordination, and the scheduling of theoretical and practical classes. By providing role-based interfaces tailored to secretaries, instructors, students, and administrators, the platform reduces the dependence on manual workflows and fragmented tools, which are common sources of inconsistencies, delays, and regulatory non-compliance. This centralization strengthens academic traceability, improves communication between institutional roles, and enables real-time monitoring of student progress, ultimately promoting more efficient internal operations and a higher quality of service for learners. The system's ability to accurately track required training hours and maintain structured records positions it as a valuable solution for institutions that must adhere to national transportation and driving regulations.

From a technical perspective, DriveMaster demonstrates the advantages of a microservices-based architecture supported by Docker containerization and a hybrid persistence model. This combination provides modularity, scalability, and maintainability, ensuring that each service can evolve independently without compromising overall system stability. Deployment with Docker Compose proved consistent and portable across environments, highlighting the platform's readiness for real institutional adoption. Furthermore, the extensive testing strategy—encompassing unit, acceptance, and performance tests—validated the reliability of the architecture, with all categories exceeding a 90% success rate. These results confirm the robustness of both backend services under concurrent load, as well as the correctness of core functionalities such as scheduling logic, vehicle management, and progress tracking.

In addition, DriveMaster sets a solid foundation for future enhancements. Potential extensions include integrating advanced analytics for student performance, automated compliance reporting for regulatory bodies, support for multiple branch management, and incorporation of mobile interfaces to increase accessibility. Security and privacy considerations are embedded throughout the system, with JWT-based authentication, role-based access control, and structured logging ensuring data integrity and accountability. By demonstrating the feasibility and benefits of a modernized, microservice-oriented management platform, DriveMaster provides a replicable model for other regulated educational environments seeking to transition from fragmented manual processes to centralized, automated, and scalable solutions.

References

[1] Object Management Group, "Business Process Model and Notation (BPMN) Version 2.0," 2011.
[2] Object Management Group, "Unified Modeling Language (UML) Specification, Version 2.5.1," 2017.
[3] Google LLC, "Angular Documentation," 2024.
[4] Python Software Foundation, "Python Documentation," 2024.
[5] C. Richardson, *Microservices Patterns*, Manning, 2018.
[6] Docker Inc., "Docker Documentation," 2024.
[7] Pivotal Software, "Spring Boot Reference Documentation," 2024.
[8] S. Ramírez, "FastAPI Documentation," 2024.
[9] D. Jones, J. Bradley, N. Sakimura, "JSON Web Token (JWT)," IETF RFC 7519, 2015.
[10] B. Okken, *Python Testing with pytest*, Pragmatic Bookshelf, 2020.
[11] G. Evans, *JUnit in Action*, Manning, 2019.
[12] A. Hellesoy, *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*, Pragmatic Bookshelf, 2018.
[13] Apache Software Foundation, "Apache JMeter User Manual," 2024.
[14] M. Benlian, H. Hess, "Opportunities and Risks of Software-as-a-Service: Findings from a Survey of IT Executives," *Decision Support Systems*, vol. 46, no. 1, pp. 230–246, 2008.
[15] MongoDB Inc., "MongoDB Manual," 2024.