Technical Report: DriveMaster Platform

Implementation of Microservices Architecture, Containerization, and CI/CD

Authors:

David Gerardo Diaz Gomez

Cristian Arturo Parra Gonzales

Daniel Mateo Montoya González

Supervisor:

Carlos Andrés Sierra

Universidad Distrital Francisco José de Caldas
Faculty of Engineering - Systems Engineering
Software Engineering Seminar

Date:

December 10, 2025

Contents

Abstract

This report documents the development and infrastructure modernization of the "DriveMaster" platform, a subscription-based management system (SaaS) for Colombian driving academies. The project integrates a dual-backend architecture (Java and Python) with an Angular frontend, ensuring modularity and scalability. The primary objective was to ensure the platform is fully deployable, testable, and maintainable through the adoption of modern software engineering practices, including containerization with Docker and orchestration with Docker Compose. Furthermore, automated acceptance testing was implemented, and a Continuous Integration and Continuous Delivery (CI/CD) workflow was established using GitHub Actions to automate validation and Docker image creation. The results demonstrate a significant improvement in system reliability and readiness for real-world deployment.

1 Introduction

Driving academy management in Colombia typically involves error-prone manual processes such as paper-based student registration, manual tracking of theoretical and practical hours, and manual certificate generation. To address this, the DriveMaster application was designed to centralize and automate student registration, instructor management, vehicle assignment, tracking of mandatory hours according to Colombian regulations (licenses A1, B1, C1, A2, B2, C2, A3, B3, and C3), and certificate generation.

This document presents the complete set of deliverables for the final project. The focus transcends functional development to target operational robustness. It addresses the need for a modular and portable environment that supports development and production workflows through component isolation. Additionally, it details the implementation of CI/CD pipelines for early defect detection, ensuring that only validated code is integrated into the main development flow.

2 Literature Review

The construction of modern distributed systems relies on three fundamental pillars guiding this project:

1. **Microservices Architecture and Modularity:** Separation of concerns is critical. This project uses a polyglot approach, employing Java (Spring Boot) for robust authentication and Python (FastAPI) for agile business logic.
2. **Containerization (Docker):** Current technical literature favors the use of containers to ensure parity between development and production environments. The use of lightweight base images (like Alpine) and multi-stage builds are standard practices for optimizing resources and security.
3. **Continuous Integration and Continuous Delivery (CI/CD):** The automation of testing and deployments through GitHub Actions enables early defect detection and maintains code quality at each integration.

3 Background

The "DriveMaster" system is composed of several interrelated subsystems: student management, instructor management, vehicle assignment, academic hours tracking, subscription management, and report generation.

The technical architecture is defined as follows:

- **Java Backend:** Manages authentication and users using Spring Boot and MySQL (Port 3307).
- **Python Backend:** Handles business logic (vehicles, academic courses, schedules) using FastAPI and MongoDB (Port 27017).
- **Frontend:** An Angular application served via Nginx, consuming REST APIs from both backends.
- **Communication:** All services communicate through a custom bridge network defined in Docker Compose.

4 Objectives

4.1 General Objective

To ensure the DriveMaster platform is fully deployable, testable, and maintainable through the adoption of modern software engineering practices.

4.2 Specific Objectives

- Containerize system components (Java, Python, Angular) to ensure consistency and portability.
- Orchestrate services and databases using Docker Compose, managing dependencies and networks.
- Implement automated acceptance tests to validate user requirements.
- Establish a CI/CD pipeline in GitHub Actions to automate unit testing, linting, and Docker image creation.
- Perform performance and stress testing using Apache JMeter to validate system capacity under load.

5 Scope

The project covers the complete Infrastructure as Code (IaC) configuration for the following components:

- **Services:** Java Backend, Python Backend, Angular Frontend.
- **Databases:** MySQL 8.0 and MongoDB 7.
- **Automation:** Multi-stage build scripts (Dockerfiles) and GitHub Actions workflows (ci.yml).
- **Testing:** Execution of unit tests in Java, Python, and performance testing with JMeter.

Exclusions: The system uses mocks for external services like real payment gateways, focusing on internal logic and architecture validation.

6 Assumptions

- **Docker Availability:** It is assumed the deployment environment has a compatible Docker runtime.
- **Connectivity:** Containers assume a stable internal network for service name resolution (e.g., mysql-drivermaster, mongo-fastapi) defined in docker-compose.yml.
- **Test Data:** It is assumed that initialization scripts are sufficient to populate the database for integration tests.
- **Version Compatibility:** It is assumed that versions of Java 17, Python 3.12, Node.js 20, and Angular are compatible with the development environment.

7 Limitations

- **Persistence in Tests:** Java unit tests use mocks (Mockito) to simulate repositories and services, which differs from the production environment where MySQL is used, although it improves execution speed and test isolation.
- **Python Database Simulation:** Python tests use custom mocks (FakeCollection) instead of MongoDB to ensure isolation and speed, which might hide specific database engine discrepancies.
- **Secret Security:** Although environment variables are used, the example docker-compose.yml file shows passwords in plain text (e.g., root), which should be managed via Docker Secrets in a real production environment.
- **Horizontal Scalability:** The current configuration does not include load balancers or orchestration with Kubernetes, limiting automatic horizontal scalability.

8 Methodology

The technical methodology was divided into four main phases:

8.1 Strategic Containerization

Multi-stage builds were used in Dockerfiles to reduce the final image size:

8.1.1 Java Backend:

- **Build Stage:** Uses `maven:3.9.8-eclipse-temurin-17`. Compiles code and downloads dependencies.
- **Runtime Stage:** Uses `eclipse-temurin:17-jdk`. Contains only the JRE and the compiled JAR file.

8.1.2 Python Backend:

- Uses `python:3.12-slim` and installs dependencies from `requirements.txt` (FastAPI, Uvicorn, Pydantic, Motor).

8.1.3 Angular Frontend:

- **Build Stage:** Uses `node:20-alpine` to generate the production build.
- **Runtime Stage:** Uses `nginx:alpine` to efficiently serve static files.

8.2 Service Orchestration

docker-compose.yml was configured to coordinate execution:

- **Dependencies:** Backend services depend on their respective databases through `depends_on`.
- **Volumes:** Persistent volumes (`mysql_data`, `mongo_data`) were defined to ensure data durability across container restarts.
- **Networks:** Services communicate through internal DNS names (mysql-drivermaster, mongo-fastapi, auth-backend, fastapi-service).

8.3 Automation (CI/CD)

A GitHub Actions workflow was implemented with the following sequential stages:

1. **Test Java Backend:** Sets up Java 17, runs Maven tests, and uploads test reports.
2. **Test Frontend:** Sets up Node.js 20, runs Angular tests with ChromeHeadless, and generates coverage reports.
3. **Test Python Backend:** Sets up Python 3.12, installs dependencies, and runs smoke tests.
4. **Build Docker Images:** If tests pass, builds Docker images using layer caching to optimize build times.

8.4 Performance Testing

Performance testing was implemented using Apache JMeter:

- **Configuration:** JMeter scripts for load testing on critical endpoints.
- **Metrics:** Response time, throughput, and error rate under different loads.
- **Reports:** Generation of HTML reports with performance analysis.

9 Results

The implementation of the methodology yielded the following verifiable technical results:

- **Java Unit Test Execution:** Multiple test suites were successfully executed using JUnit 5 and Mockito, covering controllers (5 suites), services (5 suites), and utilities (2 suites).
- **Python Unit Test Execution:** Multiple test modules were successfully completed, covering academic models, vehicle models, endpoints, and CORS.
- **Continuous Integration:** The GitHub Actions pipeline correctly executed the Checkout, Set up JDK/Python/Node, Run tests, and Build Docker Images stages, ensuring code integrity.
- **Successful Containerization:** All services were successfully containerized and can be deployed with a single command: `docker-compose up`.
- **Performance Testing:** Multiple load scenarios were executed with JMeter, generating detailed system performance reports.

10 Discussion

The adoption of containerization in DriveMaster provided immediate benefits in terms of isolation and consistency. By packaging each service with its own dependencies (Java 17 with Spring Boot for authentication, Python 3.12 with FastAPI for business logic, and Angular 20 with Node.js for the frontend), common environment conflicts that typically affect monolithic developments were eliminated. This separation is particularly valuable in the context of driving academies, where different modules (user management, vehicles, academic courses) can evolve independently.

The implemented microservices architecture demonstrates a clear separation of responsibilities: the Java backend exclusively manages authentication and authorization through JWT, while the Python backend handles business logic related to vehicles and academic management. This division allows that, for example, if the vehicle management service requires maintenance, the authentication system continues operating, allowing users to continue accessing their profiles and basic data.

The hybrid testing strategy proved effective for agile development of DriveMaster. Java unit tests use mocks with Spring Security Test, while Python tests employ custom mocks (FakeCollection) to simulate MongoDB without needing a real database. Although this approach does not exactly replicate the production environment (real MySQL and MongoDB), the significant gain in CI/CD pipeline execution speed justifies this decision, enabling rapid iterations during development.

Orchestration through Docker Compose demonstrates a decoupled architecture where a service failure (e.g., the Python vehicle management service) does not necessarily stop the Java authentication service, complying with microservices resilience principles. This characteristic is crucial for a SaaS platform like DriveMaster, where multiple academies depend on the system and partial availability is preferable to a total outage.

Performance testing with JMeter on critical endpoints of both backends revealed improvement points in query optimization and concurrent connection handling, particularly in scenarios where multiple students attempt to schedule classes simultaneously or when secretaries manage bulk registrations. These metrics provide a solid foundation for future optimizations and horizontal scaling of the system.

11 Conclusion

The project successfully achieved the transition from a developing application to a fully orchestrated and automated platform. The implementation of Docker and Docker Compose solved portability challenges, allowing the entire tech stack to be deployed with a single command. The integration of automated tests (unit and performance) within a robust CI/CD pipeline ensures that software quality remains high, reducing the risk of regressions and facilitating continuous collaboration among developers.

The DriveMaster platform is now ready for production deployment, with a scalable, maintainable, and testable architecture that meets modern software engineering standards.

12 References

1. Docker Inc. (2024). *Docker Documentation: Dockerfiles and Multi-stage builds.* Retrieved from https://docs.docker.com/
stage/

2. Newman, S. (2021). *Building Microservices: Designing Fine-Grained Systems* (2nd ed.). O'Reilly Media.

3. Humble, J., & Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley Professional.

4. GitHub. (2024). *GitHub Actions Documentation.* Retrieved from https://docs.github.com/en/actions

5. Fowler, M. (2014). *Microservices.* Retrieved from https://martinfowler.com/articles/microservices.html

13 Appendices

13.1 Appendix A: Docker Compose Configuration (Extract)

The following snippet shows the critical orchestration configuration. This extract was selected because it demonstrates the key concepts discussed in Section 8.2 (Service Orchestration): service dependencies (`depends_on`), database configuration, and the relationship between services and their data stores. The complete configuration includes additional services (MongoDB, Python backend, Angular frontend) and volume definitions, but this minimal example illustrates the orchestration pattern without overwhelming the reader with the full 60+ line configuration file.

Listing 1: Docker Compose Configuration - Extract

```
services:
  mysql-drivermaster:
    image: mysql:8.0
    environment:
      MYSQL_DATABASE: auth_db
  auth-backend:
    build: ./java-backend/auth
    depends_on:
      - mysql-drivermaster
```

13.2 Appendix B: Repository Structure

The source code organization clearly separates backend and frontend domains to facilitate independent deployment:

- `Workshop 1/` - Project definition and Business Model Canvas
- `Workshop 2/` - Architecture and design diagrams
- `Workshop 3/` - Implementation with Java backend, Python backend, Angular frontend
- `Workshop 4/` - Performance testing with JMeter and CI/CD

13.3 Appendix C: CI/CD Workflow Configuration

The GitHub Actions workflow includes the following main stages. This extract was selected to demonstrate the workflow structure discussed in Section 8.3 (Automation CI/CD): trigger events (`on: push, pull_request`), job definitions, and the dependency chain between jobs (`needs:`). The complete workflow includes additional jobs (frontend-tests, python-smoke) and detailed step configurations, but this extract captures the essential workflow pattern and job dependencies that enable the automated build process.

Listing 2: GitHub Actions CI/CD Workflow - Extract

```
name: CI
on:
  push:
  pull_request:
jobs:
  java-tests:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/setup-java@v4
        with:
          java-version: '17'
      - name: Run Maven tests
        run: cd "Final␣Project/Workshop␣3/java-backend/auth" && mvn clean test
  docker-build:
    needs: [java-tests, frontend-tests, python-smoke]
    steps:
      - name: Build Docker images
        uses: docker/build-push-action@v5
```

Acknowledgements

The authors express their sincere gratitude to Engineer Carlos Andrés Sierra, M.Sc., for his valuable academic guidance, constructive feedback, and constant support during the development of this project. His experience and knowledge in software engineering were fundamental for the successful understanding and implementation of microservices architecture, containerization, and continuous integration practices, which constitute the foundation of this work.

Glossary

CI/CD (Continuous Integration/Continuous Delivery) Practice of automating the integration of code changes and deployment to production.

Docker Platform for developing, shipping, and running applications inside containers.

Gherkin Human-readable language used to describe requirements and test scenarios in Cucumber.

JWT (JSON Web Token) Standard for creating access tokens that allow secure transmission of information between parties.

Multi-stage Build Docker strategy to use multiple FROM statements in a Dockerfile, allowing copying artifacts from one stage to another to reduce final image size.

Microservices Software architecture that structures an application as a collection of loosely coupled services.

SaaS (Software as a Service) Software distribution model where applications are hosted by a provider and made available to customers over the Internet.

Orchestration Automated coordination of multiple services or containers to work together.

List of Figures and Tables