UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Francisco José de Caldas District University

Department of Systems Engineering

WorkShop 3 - Technical Report of development of backend, frontend and test for SportGear Online

Juan Esteban Carrillo Garcia - 20212020147
Alejandro Sebastián González Torres - 20191020143

Miguel Angel Babativa Niño - 20191020069

Professor: Carlos Andrés Sierra Virgüez

A report submitted to expose the phases of design of a software product for the course of Software Engineer seminar

November 9, 2025

Declaration

We, Juan Esteban Carrillo Garcia, Alejandro Sebastián González Torres, Miguel Angel Babativa Niño, of the Department of Systems Engineering, Francisco José de Caldas District University, confirm that this is our own work and figures, tables, equations, code snippets, artworks, and illustrations in this report are original and have not been taken from any other person's work, except where the works of others have been explicitly acknowledged, quoted, and referenced. I understand that if failing to do so will be considered a case of plagiarism. Plagiarism is a form of academic misconduct and will be penalised accordingly.

I give consent to a copy of my report being shared with future students as an exemplar.

I give consent for my work to be made available more widely to members of UoR and public with interest in teaching, learning and research.

Juan Esteban Carrillo Garcia
Alejandro Sebastián González Torres
Miguel Angel Babativa Niño
November 9, 2025

Abstract

This report presents the implementation of the project's backends, developed in Java and Python, and their integration with the Web GUI. The main objective is to achieve a functional connection between the core components and to ensure code quality through unit testing and validation. The system follows a distributed full-stack architecture composed of two backends that separate authentication from business logic, both communicating through REST APIs. Authentication is managed by a Spring Boot service using JWT tokens, while the FastAPI backend handles the core business logic related to products and orders. Both services interact through RESTful interfaces and connect to MySQL and PostgreSQL databases for data persistence. On the other hand, the frontend is implemented in React, providing an intuitive user interface that consumes the exposed APIs for data and operations. Unit testing was conducted under Test-Driven Development (TDD) principles, using JUnit for service and controller validation in the Java backend and pytest for API and repository testing in the Python backend.

Keywords: REST API, TDD, REST API, Unit Testing, Code Quality

Contents

List of Abbreviations

e-commerce	Electronic Commerce

Chapter 1

Introduction

1.1 Background

After defining the conceptual and structural models of the **SportGear Online** project in previous workshops, this stage focuses on the practical implementation of its technical components. The project continues addressing the growing demand for online commerce platforms, where convenience and fast service have become key factors for modern users. At this point, the aim is to transform the system design into a functional e-commerce application that integrates two backends and a graphical web interface. This implementation phase validates that the proposed architecture can operate correctly in a real environment by ensuring communication, security, and data consistency across all components.

1.2 Problem Statement

Although the project design defined clear business and architectural goals, the challenge of this stage lies in achieving a stable and reliable implementation. Many e-commerce systems fail during this transition because of weak integration between modules or insufficient testing that leads to unstable performance. In a distributed environment, it is crucial to coordinate multiple backends and a web frontend through secure APIs and proper database management. Therefore, the problem addressed in this workshop is to implement and validate a dual-backend architecture—**Java Spring Boot** for authentication and **Python FastAPI** for business logic—integrated with a **React** frontend while maintaining software quality and robustness.

1.3 Aims and Objectives

1.3.1 General Aim

Implement and validate the complete **SportGear Online** system by connecting the Java and Python backends with the React web interface and ensuring quality through automated testing.

1.3.2 Specific Objectives

- Develop the authentication backend using **Spring Boot** with JWT-based security and a **MySQL** database.

- Build the business logic backend with **FastAPI** for managing products, orders, and payments using a **PostgreSQL** database.
- Integrate both backends through **REST APIs** and connect them to the **React** frontend.
- Apply **Test-Driven Development (TDD)** and create unit tests with **JUnit** and **pytest**.
- Evaluate system stability, modularity, and data integrity through testing results.

1.4 Solution Approach

The implementation was divided into two coordinated backends and one web frontend. The Java Spring Boot service manages user authentication, registration, and token generation with JWT, while the Python FastAPI service handles product, order, and payment operations. Both communicate through REST APIs and store information in relational databases, maintaining consistency and separation of concerns. The React interface consumes these APIs to provide a responsive experience for end users. Throughout development, the team applied TDD principles: tests were written before implementation to ensure that every feature behaved as expected, improving maintainability and reducing defects. This approach resulted in a functional and reliable integration of all system components.

1.5 Summary of Contributions and Achievements

This workshop contributes to the complete functional integration of the system's components, converting the conceptual design from previous stages into a working e-commerce platform. The main achievements include:

- Full implementation of a dual-backend architecture with successful communication through REST APIs.
- Secure authentication and session management using JWT in the Java backend.
- Stable data persistence using MySQL and PostgreSQL for separate modules.
- Integration of both backends with a React-based frontend that delivers a functional user experience.
- Application of TDD principles, achieving high unit test coverage and reliability in both environments.

1.6 Organization of the Report

This report is organized into seven chapters.

- **Chapter 1: Introduction** — Presents the background, problem statement, objectives, and general approach of the project implementation.
- **Chapter 2: Literature Review** — Provides an overview of the theoretical and technical foundations used, including frameworks such as Spring Boot, FastAPI, React, and testing methodologies like TDD.

- **Chapter 3: Methodology** — Describes the implementation strategy, architectural design, and development process supported with code snippets that illustrate good programming practices.
- **Chapter 4: Results** — Shows the main outcomes of the integration and testing phases, including test coverage metrics and performance results.
- **Chapter 5: Discussion** — Analyzes the results, interprets their significance, and discusses system performance, limitations, and improvements.
- **Chapter 6: Conclusion** — Summarizes the key findings and the overall impact of the project, highlighting the importance of the applied methodologies.
- **Chapter 7: Reflection** — Presents personal and technical reflections on the learning experience, teamwork, and skills developed during the project.

Chapter 2

Literature Review

2.1 Software Architecture and Full-Stack Development

Modern software applications rely on full-stack architectures that separate the presentation, logic, and data layers. This structure allows for modularity, scalability, and clearer responsibilities between frontend and backend development. According to Reaboi Reaboi (2024), full-stack development connects user interfaces with the underlying business logic and databases through well-defined APIs, enabling seamless data flow across components. Backend development, as described by GeeksforGeeks GeeksforGeeks (2024), is responsible for processing client requests, managing databases, and ensuring that all operations are executed securely and efficiently. This separation of concerns makes it easier to maintain and scale the system while ensuring consistent communication between different technologies.

2.2 Backend Frameworks and Integration Technologies

2.2.1 Spring Boot for Authentication

Spring Boot is a Java-based framework that simplifies the creation of RESTful backends and enterprise applications. It offers built-in support for dependency injection, security, and data persistence, making it an ideal choice for authentication services. Spring Boot integrates with **JSON Web Tokens (JWT)** to secure communication between the server and clients. According to the Auth0 manual Auth0 (2025), JWT provides a compact and self-contained method for securely transmitting information as a digitally signed JSON object. By using Spring Boot with JWT, authentication can be handled through token validation, ensuring safe and stateless communication between client and server. This approach is crucial in distributed architectures where each service must independently verify access permissions.

2.2.2 FastAPI for Business Logic

FastAPI is a Python framework designed for creating high-performance APIs with minimal code and strong data validation. Its asynchronous nature allows multiple requests to be handled simultaneously, improving scalability and response times. As explained by Suárez Suárez (2025), FastAPI supports modern design patterns such as dependency injection and model-based validation through Pydantic, encouraging clean and maintainable codebases. Its compatibility with OpenAPI also simplifies automatic documentation and testing of endpoints. For these reasons, FastAPI was selected to manage the business logic layer of the SportGear Online project, including product management, order processing, and database interactions.

2.3 Frontend Technologies

Frontend development is responsible for the user experience and interface design, providing an interactive environment for users to interact with backend systems. Modern frameworks like **React** enable the creation of dynamic and component-based user interfaces that can efficiently communicate with backend APIs. As stated by GeeksforGeeks GeeksforGeeks (2025*b*), front-end technologies such as HTML, CSS, JavaScript, and React allow developers to build responsive and intuitive interfaces that improve usability. In the context of SportGear Online, React serves as the bridge between the user and the system's data, consuming APIs to display products, handle user sessions, and manage orders in real time.

2.4 Software Testing and Quality Assurance

Testing and quality assurance (QA) are essential processes that ensure software reliability and maintainability. Testing focuses on verifying system functionality, while QA involves defining and enforcing standards throughout the development process GeeksforGeeks (2025*a*). Adopting a **Test-Driven Development (TDD)** approach promotes writing test cases before implementing the code, helping developers focus on requirements and detect issues early. Beck Beck (2003) defines TDD as a cyclic process of writing a failing test, implementing the minimum code to pass it, and refactoring. For the SportGear Online system, TDD was applied using **JUnit** for the Java backend and **pytest** for the Python backend, ensuring consistent testing practices across both environments. This strategy enhances code coverage and strengthens the system's long-term reliability.

2.5 Security and Data Management

Security and data management are critical aspects of distributed systems, especially for e-commerce applications where sensitive user information is stored and processed. As explained by Cohesity Cohesity (2025), effective data management involves encryption, controlled access, and integrity verification to protect data at every stage. In SportGear Online, authentication and authorization are enforced through JWT validation, while databases such as MySQL and PostgreSQL manage structured information for authentication and business processes, respectively. These combined measures ensure secure communication between services and reliable data persistence across all system layers.

2.6 Summary

The reviewed literature highlights the importance of combining modern frameworks, security standards, and testing methodologies to build reliable full-stack web applications. Spring Boot and FastAPI provide strong backends for authentication and business logic, while React delivers a flexible and responsive frontend. Security and testing practices based on JWT and TDD principles ensure code quality and data protection throughout the project lifecycle. Altogether, these tools and concepts form the technical foundation for the successful implementation of the SportGear Online platform.

Chapter 3

Methodology

3.1 System Architecture

The development of the **SportGear Online** platform followed a distributed full-stack design composed of two main backend services and a single web frontend. This architecture was designed to separate the system's responsibilities: one backend manages **authentication and user management** through **Spring Boot**, and the other handles the **business logic** for products and orders using **FastAPI**. The frontend, developed in **React**, acts as a client that communicates with both APIs to present information and actions to users.

Each backend runs independently and communicates using REST APIs over HTTP. Spring Boot connects to a **MySQL** database to store user credentials, while FastAPI uses a **PostgreSQL** database to persist product and order data. This structure improves modularity and scalability, allowing independent deployment and testing for each service.

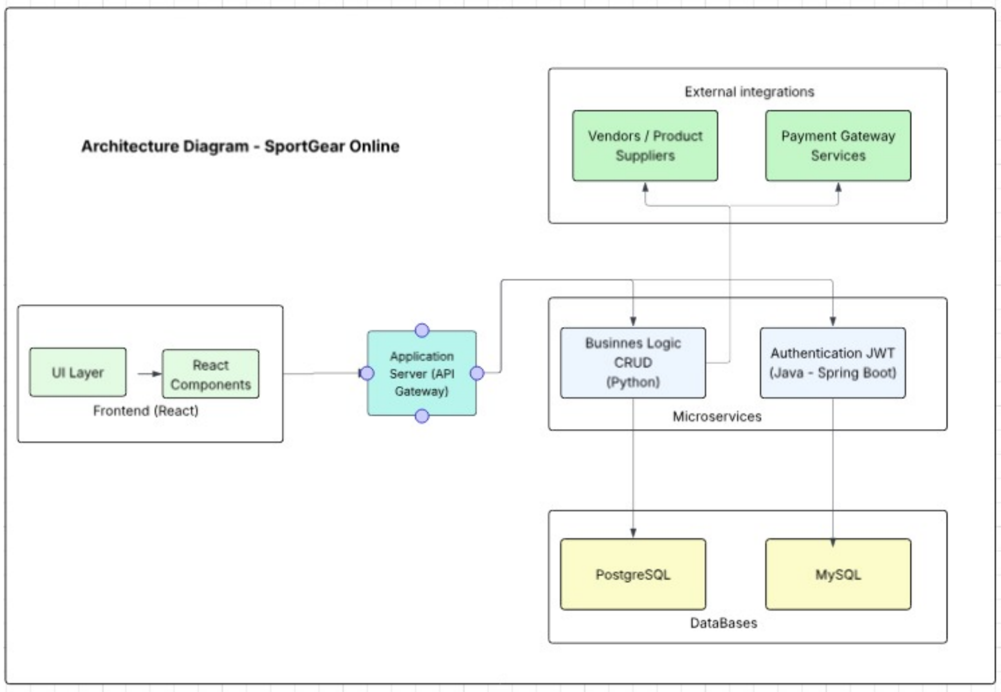

Architecture Diagram - SportGear Online
External integrations
Vendors / Product Suppliers
Payment Gateway Services
UI Layer
React Components
Frontend (React)
Application Server (API Gateway)
Businnes Logic CRUD (Python)
Authentication JWT (Java - Spring Boot)
Microservices
PostgreSQL
MySQL
DataBases

Figure 3.1: General architecture of the SportGear Online platform showing the two backends and the frontend integration.

3.2 Backend 1: Spring Boot Authentication Service

The authentication module was developed with **Spring Boot 3**, which provides the foundation for dependency injection, web controllers, and security configuration. The main goal of this backend is to register new users, manage authentication through JWT tokens, and validate each request received by the frontend or the second backend.

3.2.1 Structure and Components

The Java backend follows a clean layered structure:

- **Controller layer:** manages HTTP requests and exposes endpoints like `/auth/login` and `/auth/register`.
- **Repository layer:** handles database operations through JPA.
- **Security layer:** configures authentication filters and token validation.
- **Model layer:** defines entities such as `User`.

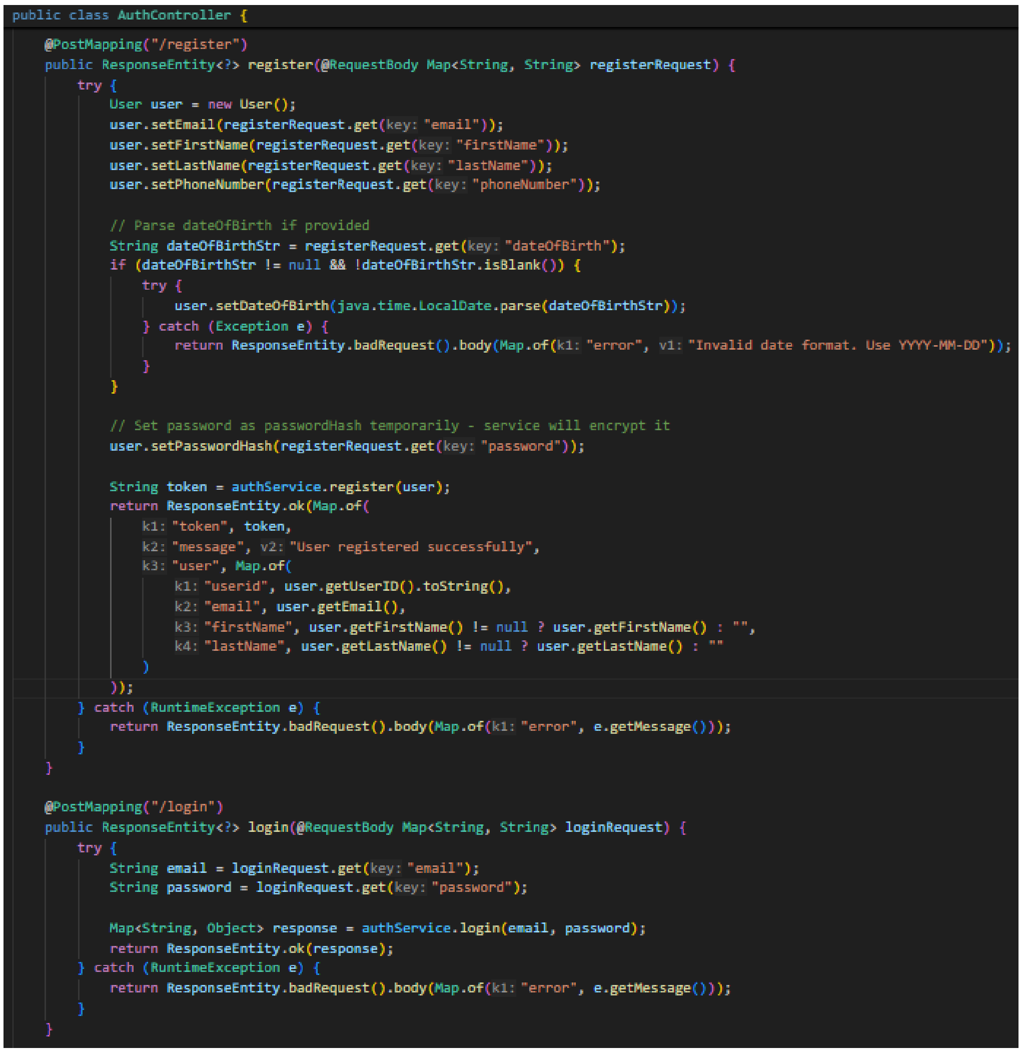
The project's package structure is similar to the following:

```
src/main/java/com/sportgear/auth/
 |-- controller/
 -----|-- AuthController.java
 |-- model/
 -----|-- User.java
 |-- repository/
 -----|-- UserRepository.java
 |-- security/
 -----|-- JwtUtils.java
 -----|-- SecurityConfig.java
 |-- SportgearAuthApplication.java
```

Listing 3.1: Spring Boot project structure

3.2.2 Authentication Controller

The controller exposes endpoints that allow the user to register and authenticate. Each request passes through Spring Security filters, and successful login responses include a JWT token that must be used for future API calls.

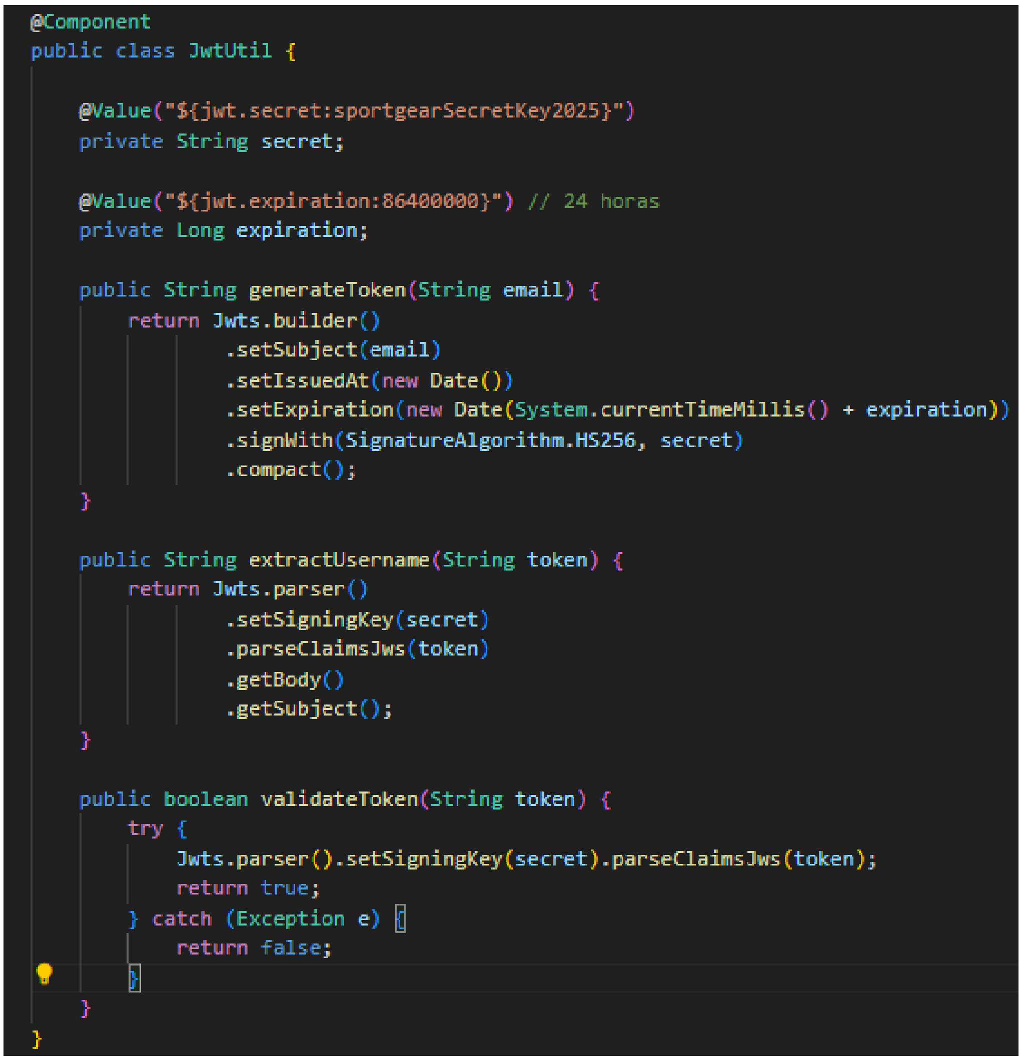


```java
public class AuthController {

    @PostMapping("/register")
    public ResponseEntity<?> register(@RequestBody Map<String, String> registerRequest) {
        try {
            User user = new User();
            user.setEmail(registerRequest.get(key: "email"));
            user.setFirstName(registerRequest.get(key: "firstName"));
            user.setLastName(registerRequest.get(key: "lastName"));
            user.setPhoneNumber(registerRequest.get(key: "phoneNumber"));

            // Parse dateOfBirth if provided
            String dateOfBirthStr = registerRequest.get(key: "dateOfBirth");
            if (dateOfBirthStr != null && !dateOfBirthStr.isBlank()) {
                try {
                    user.setDateOfBirth(java.time.LocalDate.parse(dateOfBirthStr));
                } catch (Exception e) {
                    return ResponseEntity.badRequest().body(Map.of(k1: "error", v1: "Invalid date format. Use YYYY-MM-DD"));
                }
            }

            // Set password as passwordHash temporarily - service will encrypt it
            user.setPasswordHash(registerRequest.get(key: "password"));

            String token = authService.register(user);
            return ResponseEntity.ok(Map.of(
                k1: "token", token,
                k2: "message", v2: "User registered successfully",
                k3: "user", Map.of(
                    k1: "userid", user.getUserID().toString(),
                    k2: "email", user.getEmail(),
                    k3: "firstName", user.getFirstName() != null ? user.getFirstName() : "",
                    k4: "lastName", user.getLastName() != null ? user.getLastName() : ""
                )
            ));
        } catch (RuntimeException e) {
            return ResponseEntity.badRequest().body(Map.of(k1: "error", e.getMessage()));
        }
    }

    @PostMapping("/login")
    public ResponseEntity<?> login(@RequestBody Map<String, String> loginRequest) {
        try {
            String email = loginRequest.get(key: "email");
            String password = loginRequest.get(key: "password");

            Map<String, Object> response = authService.login(email, password);
            return ResponseEntity.ok(response);
        } catch (RuntimeException e) {
            return ResponseEntity.badRequest().body(Map.of(k1: "error", e.getMessage()));
        }
    }
```

Figure 3.2: AuthController managing login and registration.

3.2.3 JWT Utility

Token generation and validation are centralized in the `JwtUtils` class, which ensures secure and stateless authentication.

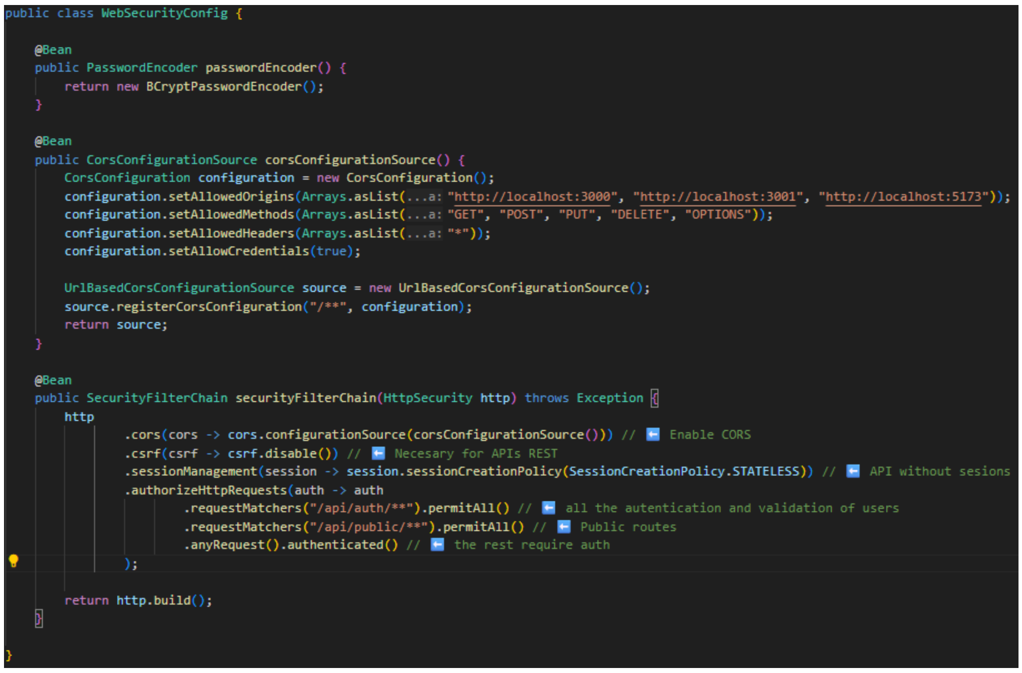


```java
@Component
public class JwtUtil {

    @Value("${jwt.secret:sportgearSecretKey2025}")
    private String secret;

    @Value("${jwt.expiration:86400000}") // 24 horas
    private Long expiration;

    public String generateToken(String email) {
        return Jwts.builder()
                .setSubject(email)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() + expiration))
                .signWith(SignatureAlgorithm.HS256, secret)
                .compact();
    }

    public String extractUsername(String token) {
        return Jwts.parser()
                .setSigningKey(secret)
                .parseClaimsJws(token)
                .getBody()
                .getSubject();
    }

    public boolean validateToken(String token) {
        try {
            Jwts.parser().setSigningKey(secret).parseClaimsJws(token);
            return true;
        } catch (Exception e) {
            return false;
        }
    }
}
```

Figure 3.3: JWT generation and validation.

3.2.4 Web Security Configuration

The class `WebSecurityConfig` defines access rules and CORS configuration, allowing the frontend to consume the authentication endpoints.



```java
public class WebSecurityConfig {

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration configuration = new CorsConfiguration();
        configuration.setAllowedOrigins(Arrays.asList(...a: "http://localhost:3000", "http://localhost:3001", "http://localhost:5173"));
        configuration.setAllowedMethods(Arrays.asList(...a: "GET", "POST", "PUT", "DELETE", "OPTIONS"));
        configuration.setAllowedHeaders(Arrays.asList(...a: "*"));
        configuration.setAllowCredentials(true);

        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", configuration);
        return source;
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
                .cors(cors -> cors.configurationSource(corsConfigurationSource())) // ⬅ Enable CORS
                .csrf(csrf -> csrf.disable()) // ⬅ Necesary for APIs REST
                .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS)) // ⬅ API without sesions
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers("/api/auth/**").permitAll() // ⬅ all the autentication and validation of users
                        .requestMatchers("/api/public/**").permitAll() // ⬅ Public routes
                        .anyRequest().authenticated() // ⬅ the rest require auth
                );

        return http.build();
    }

}
```

Figure 3.4: Spring Security configuration.

3.2.5 Backend 2: FastAPI Business Logic Service

The second backend was implemented using **FastAPI** in Python. This service manages the core business logic of the system, including operations related to products, orders, and payments. It exposes RESTful endpoints consumed by the frontend and, in some cases, by the authentication backend. FastAPI was chosen for its high performance, asynchronous capabilities, and automatic data validation using Pydantic models.

3.2.6 Project Organization

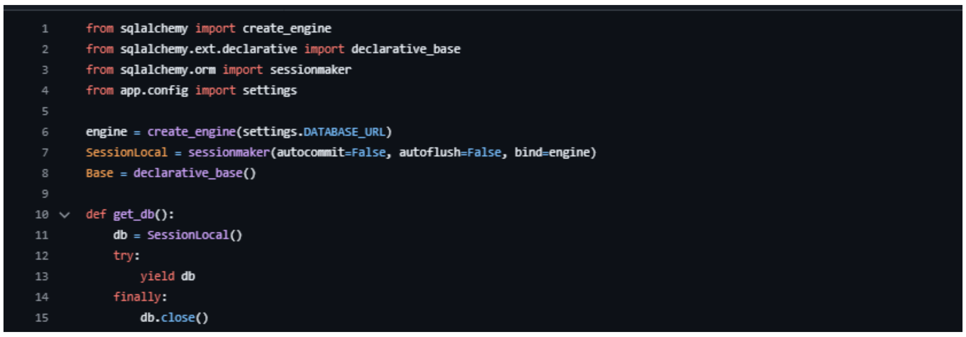
The project follows a modular structure that separates concerns between routes, models, schemas, and database configuration. This design simplifies maintenance and allows independent scaling of specific features.

```
app/
 |-- main.py
 |-- routes/
 -----|-- products.py
 -----|-- orders.py
 -----|-- payments.py
 |-- models/
 -----|-- product.py
 -----|-- order.py
 -----|-- payment.py
 |-- schemas/
 -----|-- product_schema.py
 -----|-- order_schema.py
 -----|-- payment_schema.py
 |-- database.py
```

Listing 3.2: FastAPI project structure

3.2.7 Database Connection

The connection to PostgreSQL is managed through SQLAlchemy. The file `database.py` defines the connection string and session management for the ORM. This configuration ensures that all CRUD operations on products and orders are handled efficiently.

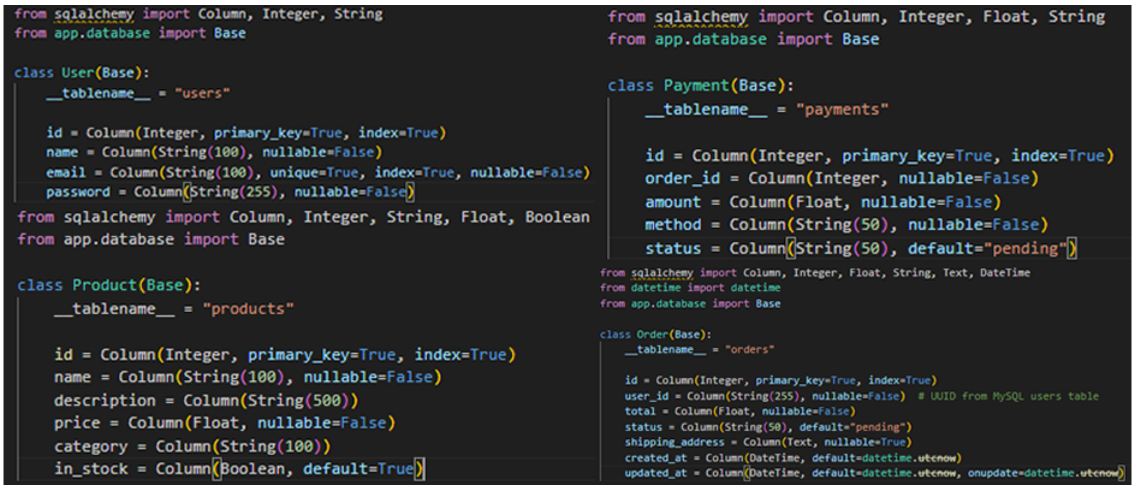


```python
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
from app.config import settings

engine = create_engine(settings.DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

Figure 3.5: Database connection configuration for the FastAPI backend using SQLAlchemy.

3.2.8 Data Models and Schemas

Each entity in the business domain (products, orders, and payments) has a model that maps to a PostgreSQL table and a corresponding Pydantic schema for request validation. This dual-layer approach guarantees that the data entering the API meets all defined constraints.



```python
from sqlalchemy import Column, Integer, String
from app.database import Base

class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String(100), nullable=False)
    email = Column(String(100), unique=True, index=True, nullable=False)
    password = Column(String(255), nullable=False)
```

```python
from sqlalchemy import Column, Integer, String, Float, Boolean
from app.database import Base

class Product(Base):
    __tablename__ = "products"

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String(100), nullable=False)
    description = Column(String(500))
    price = Column(Float, nullable=False)
    category = Column(String(100))
    in_stock = Column(Boolean, default=True)
```

```python
from sqlalchemy import Column, Integer, Float, String
from app.database import Base

class Payment(Base):
    __tablename__ = "payments"

    id = Column(Integer, primary_key=True, index=True)
    order_id = Column(Integer, nullable=False)
    amount = Column(Float, nullable=False)
    method = Column(String(50), nullable=False)
    status = Column(String(50), default="pending")
```

```python
from sqlalchemy import Column, Integer, Float, String, Text, DateTime
from datetime import datetime
from app.database import Base

class Order(Base):
    __tablename__ = "orders"

    id = Column(Integer, primary_key=True, index=True)
    user_id = Column(String(255), nullable=False)  # UUID from MySQL users table
    total = Column(Float, nullable=False)
    status = Column(String(50), default="pending")
    shipping_address = Column(Text, nullable=True)
    created_at = Column(DateTime, default=datetime.utcnow)
    updated_at = Column(DateTime, default=datetime.utcnow, onupdate=datetime.utcnow)
```

Figure 3.6: FastAPI model and schema definitions for product management.

3.2.9 CRUD Operations

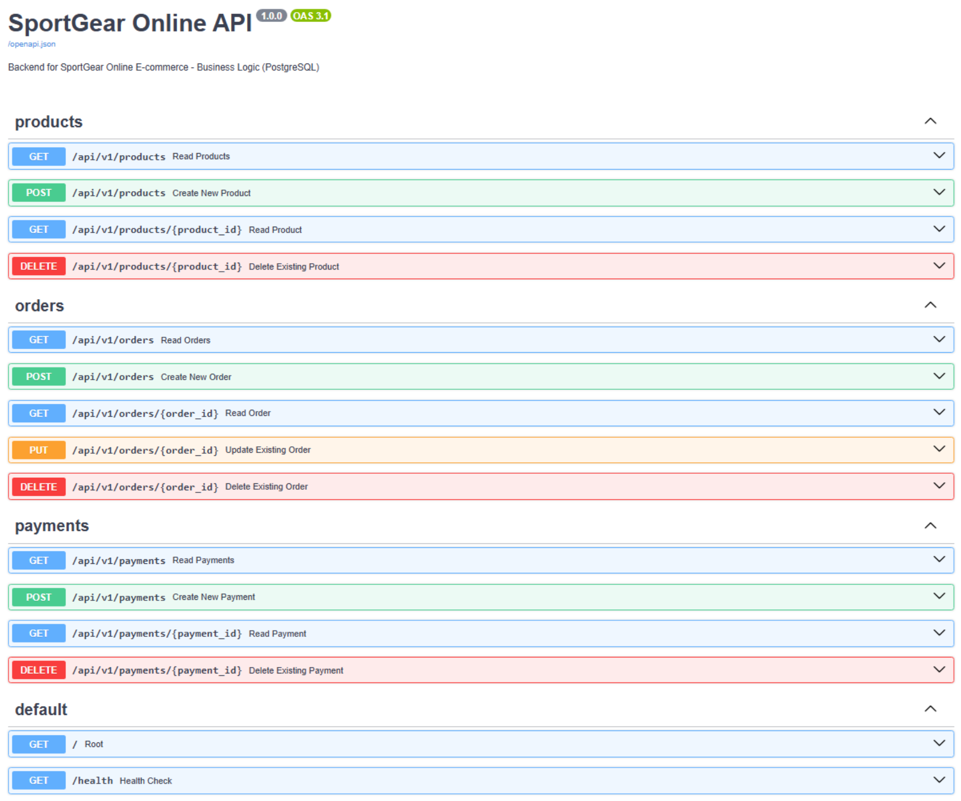
The CRUD functionality was implemented through dedicated route files in the `routes` directory. These handle creation, retrieval, update, and deletion for each entity. Each route uses dependency injection to obtain the active database session.

```python
@router.get("/products")
def get_products(db: Session = Depends(get_db)):
    return db.query(Product).all()

@router.post("/products")
```

```python
def create_product(product: ProductSchema, db: Session = Depends(get_db)):
    new_product = Product(**product.dict())
    db.add(new_product)
    db.commit()
    db.refresh(new_product)
    return new_product
```

Listing 3.3: CRUD endpoints for products in FastAPI

These endpoints are automatically documented by FastAPI and can be tested interactively through Swagger UI at `/docs`.



SportGear Online API 1.0.0 OAS 3.1
/openapi.json
Backend for SportGear Online E-commerce - Business Logic (PostgreSQL)

products
- GET `/api/v1/products` Read Products
- POST `/api/v1/products` Create New Product
- GET `/api/v1/products/{product_id}` Read Product
- DELETE `/api/v1/products/{product_id}` Delete Existing Product

orders
- GET `/api/v1/orders` Read Orders
- POST `/api/v1/orders` Create New Order
- GET `/api/v1/orders/{order_id}` Read Order
- PUT `/api/v1/orders/{order_id}` Update Existing Order
- DELETE `/api/v1/orders/{order_id}` Delete Existing Order

payments
- GET `/api/v1/payments` Read Payments
- POST `/api/v1/payments` Create New Payment
- GET `/api/v1/payments/{payment_id}` Read Payment
- DELETE `/api/v1/payments/{payment_id}` Delete Existing Payment

default
- GET `/` Root
- GET `/health` Health Check

Figure 3.7: Swagger UI automatically generated by FastAPI for endpoint testing.

—

3.3 REST API Endpoints

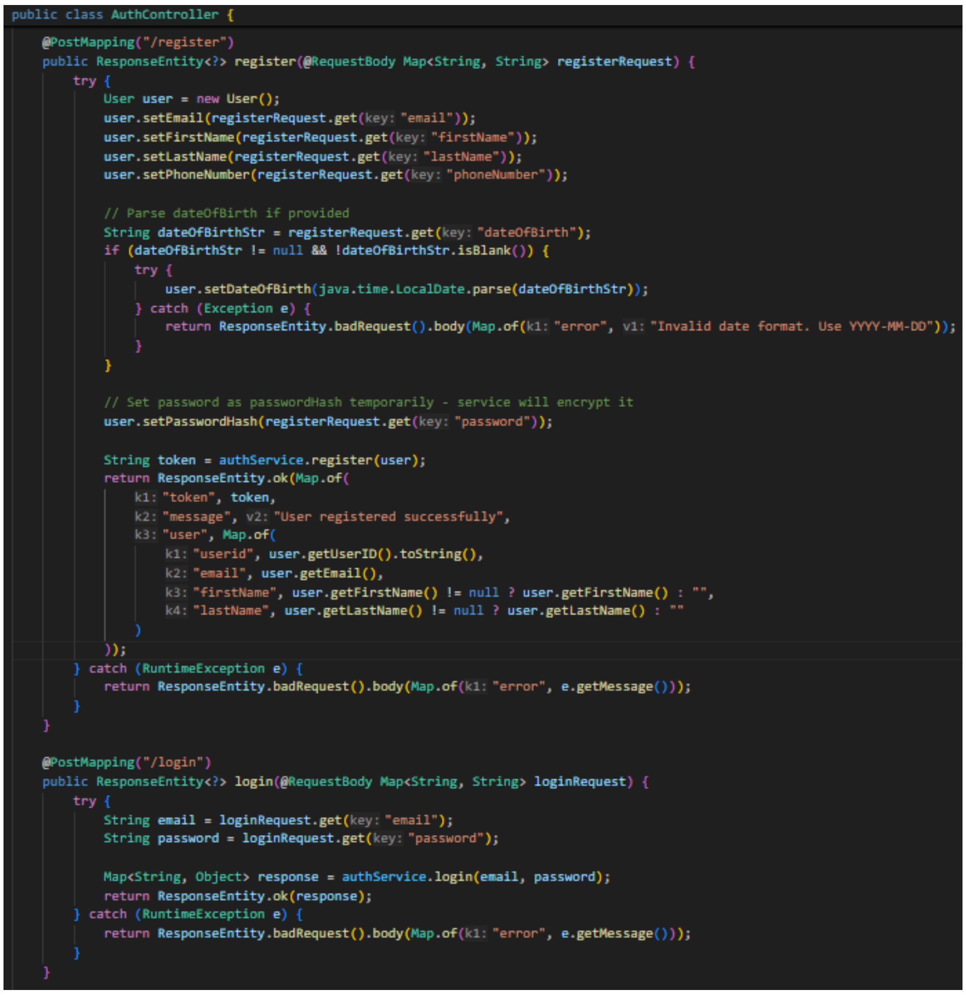
This section summarizes the main REST API endpoints implemented in both backends. Each API follows RESTful principles and communicates via JSON over HTTP.

3.3.1 Authentication API (Spring Boot)

The authentication backend exposes endpoints related to user management and session control.

- `POST /api/auth/register` – Registers a new user in MySQL.
- `POST /api/auth/login` – Authenticates users and returns a JWT token.

- `GET /api/auth/users/{id}` – Retrieves user data for authorized requests.



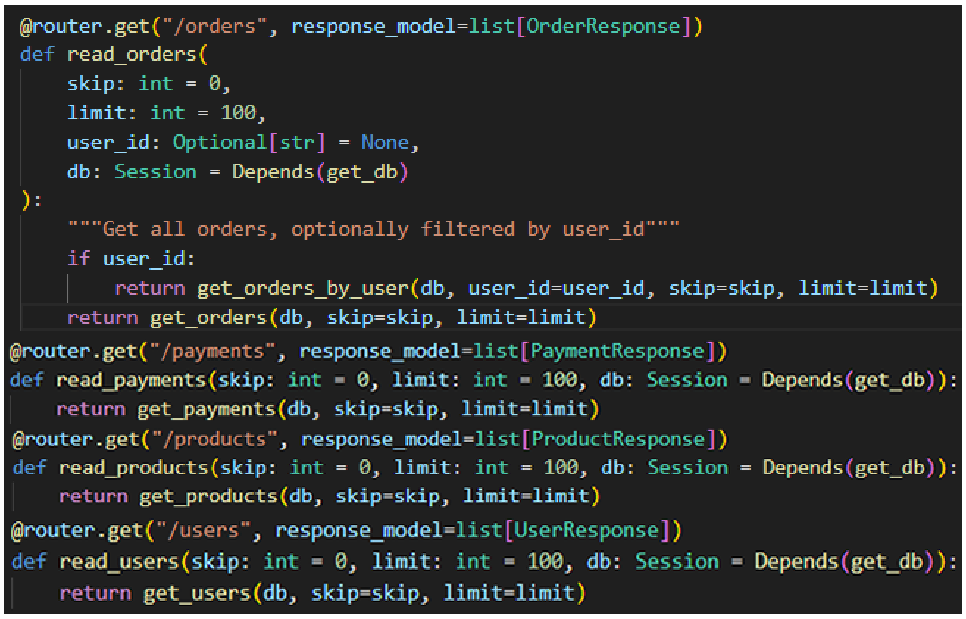
```java
public class AuthController {
    @PostMapping("/register")
    public ResponseEntity<?> register(@RequestBody Map<String, String> registerRequest) {
        try {
            User user = new User();
            user.setEmail(registerRequest.get(key: "email"));
            user.setFirstName(registerRequest.get(key: "firstName"));
            user.setLastName(registerRequest.get(key: "lastName"));
            user.setPhoneNumber(registerRequest.get(key: "phoneNumber"));

            // Parse dateOfBirth if provided
            String dateOfBirthStr = registerRequest.get(key: "dateOfBirth");
            if (dateOfBirthStr != null && !dateOfBirthStr.isBlank()) {
                try {
                    user.setDateOfBirth(java.time.LocalDate.parse(dateOfBirthStr));
                } catch (Exception e) {
                    return ResponseEntity.badRequest().body(Map.of(k1: "error", v1: "Invalid date format. Use YYYY-MM-DD"));
                }
            }

            // Set password as passwordHash temporarily - service will encrypt it
            user.setPasswordHash(registerRequest.get(key: "password"));

            String token = authService.register(user);
            return ResponseEntity.ok(Map.of(
                k1: "token", token,
                k2: "message", v2: "User registered successfully",
                k3: "user", Map.of(
                    k1: "userid", user.getUserID().toString(),
                    k2: "email", user.getEmail(),
                    k3: "firstName", user.getFirstName() != null ? user.getFirstName() : "",
                    k4: "lastName", user.getLastName() != null ? user.getLastName() : ""
                )
            ));
        } catch (RuntimeException e) {
            return ResponseEntity.badRequest().body(Map.of(k1: "error", e.getMessage()));
        }
    }

    @PostMapping("/login")
    public ResponseEntity<?> login(@RequestBody Map<String, String> loginRequest) {
        try {
            String email = loginRequest.get(key: "email");
            String password = loginRequest.get(key: "password");

            Map<String, Object> response = authService.login(email, password);
            return ResponseEntity.ok(response);
        } catch (RuntimeException e) {
            return ResponseEntity.badRequest().body(Map.of(k1: "error", e.getMessage()));
        }
    }
```

Figure 3.8: Spring Boot authentication endpoints used for user registration and login.

3.3.2 Business Logic API (FastAPI)

The FastAPI backend handles operations related to products, orders, payments and users. The endpoints interact with the PostgreSQL database and are secured via JWT tokens provided by the Java authentication backend.

- `GET /orders` – Retrieves all orders, optionally filtered by user ID.
- `GET /payments` – Retrieves all registered payments.
- `GET /products` – Lists all available products.
- `GET /users` – Lists registered users.



```python
@router.get("/orders", response_model=list[OrderResponse])
def read_orders(
    skip: int = 0,
    limit: int = 100,
    user_id: Optional[str] = None,
    db: Session = Depends(get_db)
):
    """Get all orders, optionally filtered by user_id"""
    if user_id:
        return get_orders_by_user(db, user_id=user_id, skip=skip, limit=limit)
    return get_orders(db, skip=skip, limit=limit)
@router.get("/payments", response_model=list[PaymentResponse])
def read_payments(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    return get_payments(db, skip=skip, limit=limit)
@router.get("/products", response_model=list[ProductResponse])
def read_products(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    return get_products(db, skip=skip, limit=limit)
@router.get("/users", response_model=list[UserResponse])
def read_users(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    return get_users(db, skip=skip, limit=limit)
```

Figure 3.9: FastAPI examples of routes exposing CRUD operations for products, orders, payments and users.

—

3.4 Testing Methodology

Testing was applied throughout the development phase following a **Test-Driven Development (TDD)** approach. Both backends include unit and integration tests to ensure reliability and correctness of their core features.

3.4.1 Java Backend Testing

The Spring Boot backend uses **JUnit 5** for testing authentication, token validation, and database persistence. The following snippet shows a representative test for the login endpoint.

```java
@SpringBootTest
@AutoConfigureMockMvc
class AuthControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void testLoginShouldReturnJwtToken() throws Exception {
        mockMvc.perform(post("/api/auth/login")
                .contentType(MediaType.APPLICATION_JSON)
                .content("{\"username\":\"admin\",\"password\":\"1234\"}")
        )
                .andExpect(status().isOk());
    }
}
```

Listing 3.4: JUnit test for authentication endpoint

3.4.2 Python Backend Testing

For the FastAPI backend, tests were implemented using **pytest** and the built-in `TestClient`. These tests verify CRUD operations and validate responses from the REST endpoints.

```
from fastapi.testclient import TestClient
from app.main import app

client = TestClient(app)

def test_get_products():
    response = client.get("/api/v1/products")
    assert response.status_code == 200
    assert isinstance(response.json(), list)
```

Listing 3.5: pytest test for product endpoint

All tests were executed successfully, confirming proper communication between layers and ensuring that both APIs behave as expected.

3.5 Summary

This chapter described the implementation process of the **SportGear Online** platform, detailing both backend architectures and their REST APIs. The Java backend securely manages authentication using JWT and MySQL, while the Python backend handles product and order management with PostgreSQL and FastAPI. Testing confirmed the stability and correct behavior of endpoints before frontend integration. The next chapter presents the system results, GUI evidence, and full API interaction.

Chapter 4

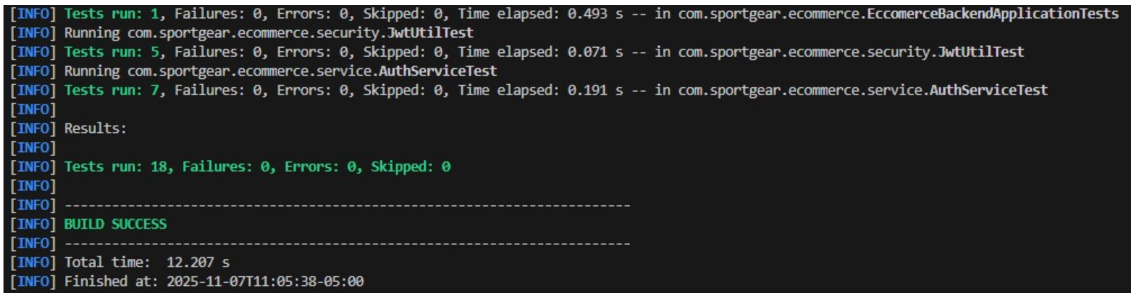
Results

4.1 Unit Test Execution Results

This section presents the actual execution of unit and integration tests for both backend services. Each test validates a critical functionality within its respective backend and confirms the correct behavior of the APIs before integration with the frontend.

4.1.1 Java Backend

The Java authentication service was tested using JUnit 5 and MockMvc. Tests included validation of user registration, login, JWT generation, and repository persistence using an H2 embedded database.

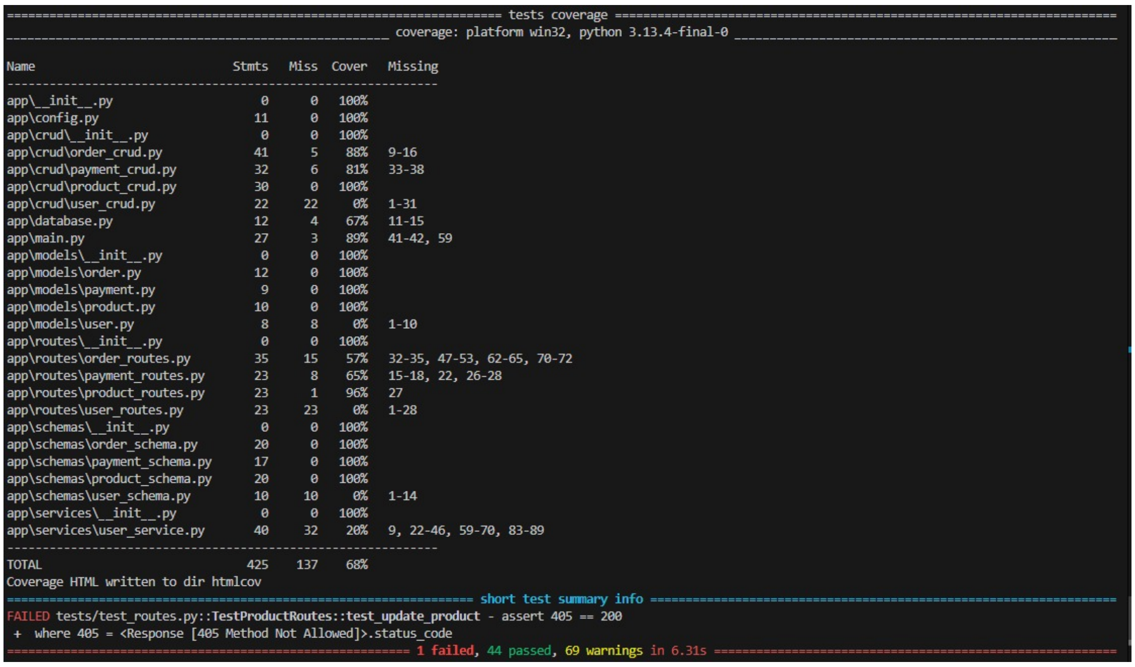


```
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.493 s -- in com.sportgear.ecommerce.EccomerceBackendApplicationTests
[INFO] Running com.sportgear.ecommerce.security.JwtUtilTest
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.071 s -- in com.sportgear.ecommerce.security.JwtUtilTest
[INFO] Running com.sportgear.ecommerce.service.AuthServiceTest
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.191 s -- in com.sportgear.ecommerce.service.AuthServiceTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 18, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  12.207 s
[INFO] Finished at: 2025-11-07T11:05:38-05:00
```

Figure 4.1: Complete test suite execution results for the Java backend using JUnit 5. All tests passed successfully, confirming authentication and JWT flow correctness.

4.1.2 Python Backend

The Python backend, implemented with FastAPI, was tested using the `pytest` framework and FastAPI's `TestClient`. The test suite verified CRUD operations on products, orders, and payments, as well as authentication validation through mocked tokens.



```
=================================================================== tests coverage ====================================================================
___________________________________________________ coverage: platform win32, python 3.13.4-final-0 ____________________________________________________

Name                          Stmts   Miss  Cover   Missing
-----------------------------------------------------------
app\__init__.py                   0      0   100%
app\config.py                    11      0   100%
app\crud\__init__.py              0      0   100%
app\crud\order_crud.py           41      5    88%   9-16
app\crud\payment_crud.py         32      6    81%   33-38
app\crud\product_crud.py         30      0   100%
app\crud\user_crud.py            22     22     0%   1-31
app\database.py                  12      4    67%   11-15
app\main.py                      27      3    89%   41-42, 59
app\models\__init__.py            0      0   100%
app\models\order.py              12      0   100%
app\models\payment.py             9      0   100%
app\models\product.py            10      0   100%
app\models\user.py                8      8     0%   1-10
app\routes\__init__.py            0      0   100%
app\routes\order_routes.py       35     15    57%   32-35, 47-53, 62-65, 70-72
app\routes\payment_routes.py     23      8    65%   15-18, 22, 26-28
app\routes\product_routes.py     23      1    96%   27
app\routes\user_routes.py        23     23     0%   1-28
app\schemas\__init__.py           0      0   100%
app\schemas\order_schema.py      20      0   100%
app\schemas\payment_schema.py    17      0   100%
app\schemas\product_schema.py    20      0   100%
app\schemas\user_schema.py       10     10     0%   1-14
app\services\__init__.py          0      0   100%
app\services\user_service.py     40     32    20%   9, 22-46, 59-70, 83-89
-----------------------------------------------------------
TOTAL                           425    137    68%
Coverage HTML written to dir htmlcov
=============================================================== short test summary info ===============================================================
FAILED tests/test_routes.py::TestProductRoutes::test_update_product - assert 405 == 200
 +  where 405 = <Response [405 Method Not Allowed]>.status_code
=================================================== 1 failed, 44 passed, 69 warnings in 6.31s ====================================================
```

Figure 4.2: Unit test execution results for the Python backend using pytest. The tests validated the correctness of CRUD operations, schema validation, and route availability.

—

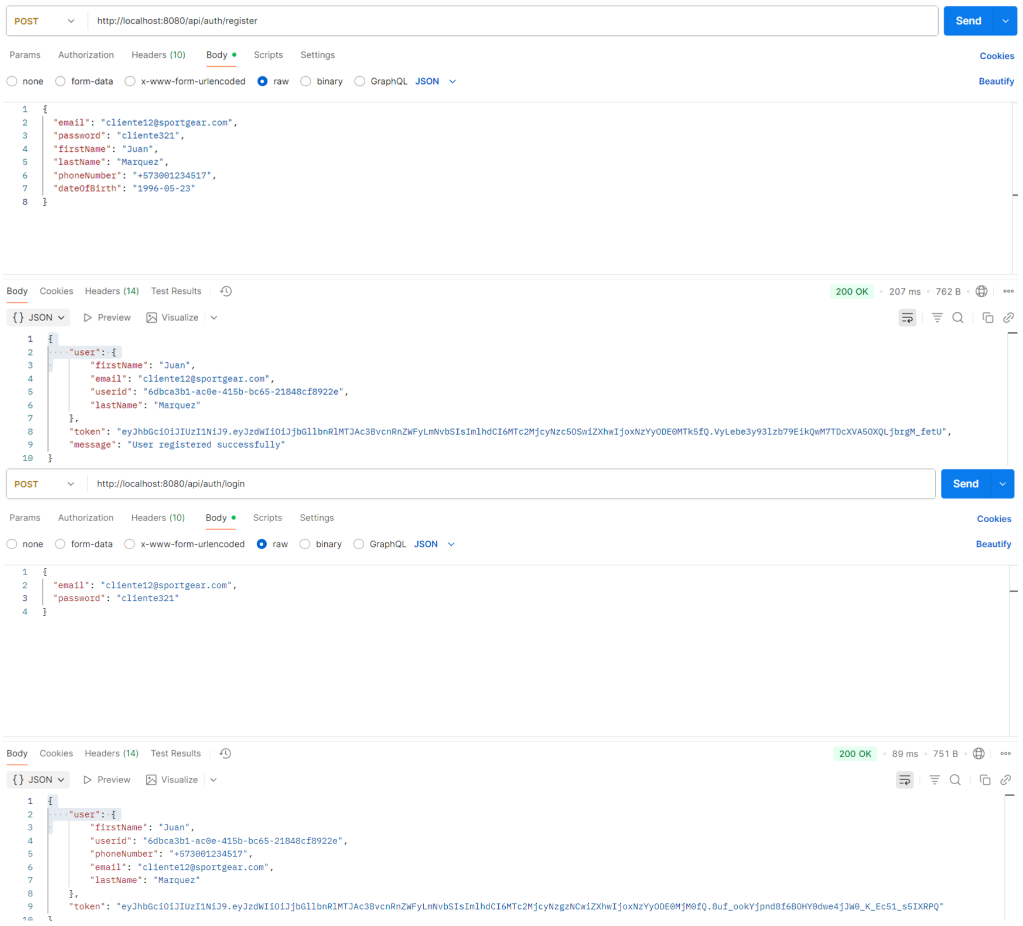
4.2 API Testing with Postman

To complement automated tests, both backend APIs were validated using **Postman**. These manual tests ensure the endpoints correctly handle HTTP requests and responses, including authentication headers and payload formats.

4.2.1 Java Authentication API (Spring Boot)

The following requests were executed against the authentication backend hosted at `http://localhost:8080/ap`

- `POST /register` – creates a new user.
- `POST /login` – authenticates an existing user and returns a JWT token.


POST http://localhost:8080/api/auth/register

Body: raw JSON

```
{
  "email": "cliente12@sportgear.com",
  "password": "cliente321",
  "firstName": "Juan",
  "lastName": "Marquez",
  "phoneNumber": "+573001234517",
  "dateOfBirth": "1996-05-23"
}
```

200 OK · 207 ms · 762 B

```
{
    "user": {
        "firstName": "Juan",
        "email": "cliente12@sportgear.com",
        "userid": "6dbca3b1-ac0e-415b-bc65-21848cf8922e",
        "lastName": "Marquez"
    },
    "token": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJjbGllbnRlMTJAc3BvcnRnZWFyLmNvbSIsImlhdCI6MTc2MjcyNzc5OSwiZXhwIjoxNzYyODE0MTk5fQ.VyLebe3y93lzb79EikQwM7TDcXVA5OXQLjbrgM_fetU",
    "message": "User registered successfully"
}
```

POST http://localhost:8080/api/auth/login

Body: raw JSON

```
{
  "email": "cliente12@sportgear.com",
  "password": "cliente321"
}
```

200 OK · 89 ms · 751 B

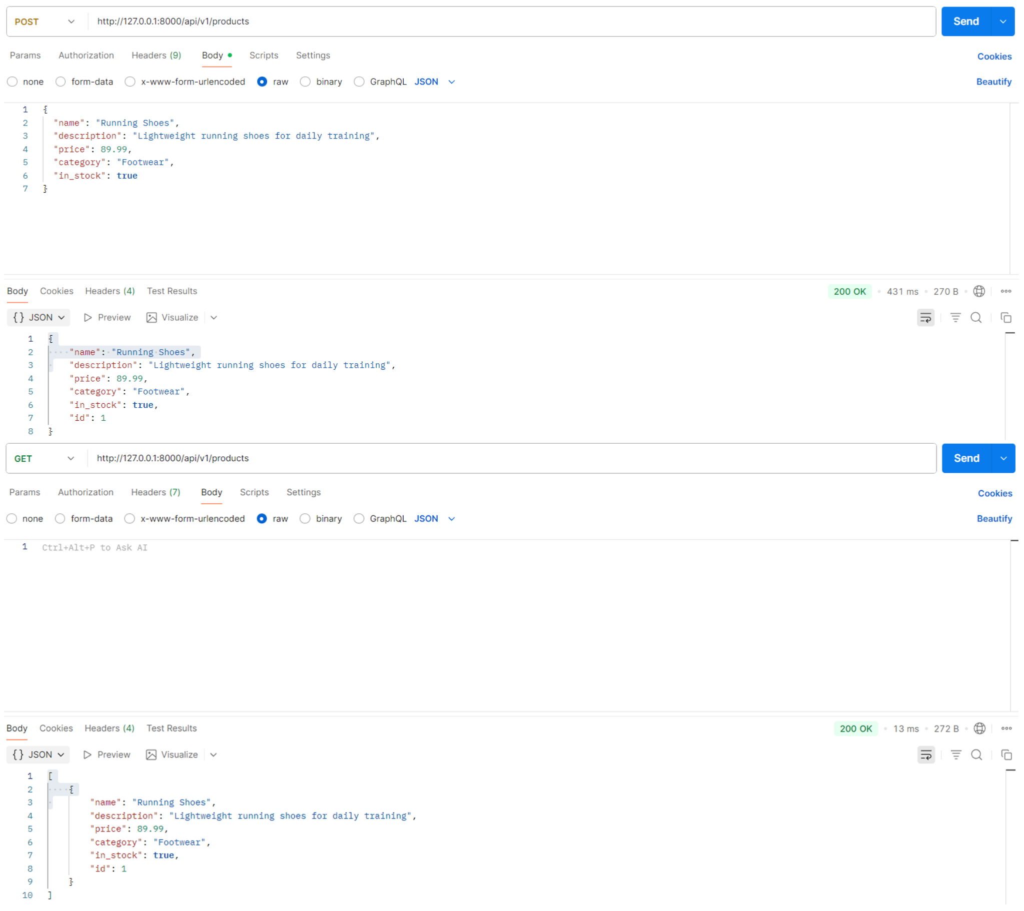
```
{
    "user": {
        "firstName": "Juan",
        "userid": "6dbca3b1-ac0e-415b-bc65-21848cf8922e",
        "phoneNumber": "+573001234517",
        "email": "cliente12@sportgear.com",
        "lastName": "Marquez"
    },
    "token": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJjbGllbnRlMTJAc3BvcnRnZWFyLmNvbSIsImlhdCI6MTc2MjcyNzgzNCwiZXhwIjoxNzYyODE0MjM0fQ.8uf_ookYjpnd8f6BOHY0dwe4jJW0_K_Ec51_s5IXRPQ"
```

Figure 4.3: Postman request validating Java backend authentication endpoints for user registration and login. The response includes a valid JWT token.

4.2.2 Python Business Logic API (FastAPI)

The business logic API was tested at `http://localhost:8000/api/v1` using JWT authentication obtained from the Java backend. Endpoints for products were selected as representative examples of CRUD operations.

- `GET /products` – retrieves all existing products.
- `POST /products` – creates a new product with JSON payload.



http://127.0.0.1:8000/api/v1/products

```
{
  "name": "Running Shoes",
  "description": "Lightweight running shoes for daily training",
  "price": 89.99,
  "category": "Footwear",
  "in_stock": true
}
```

200 OK • 431 ms • 270 B

```
{
    "name": "Running Shoes",
    "description": "Lightweight running shoes for daily training",
    "price": 89.99,
    "category": "Footwear",
    "in_stock": true,
    "id": 1
}
```

http://127.0.0.1:8000/api/v1/products

```
Ctrl+Alt+P to Ask AI
```

200 OK • 13 ms • 272 B

```
[
    {
        "name": "Running Shoes",
        "description": "Lightweight running shoes for daily training",
        "price": 89.99,
        "category": "Footwear",
        "in_stock": true,
        "id": 1
    }
]
```

Figure 4.4: Postman request examples for FastAPI business logic endpoints. Successful GET and POST requests confirm product data persistence and response validation.

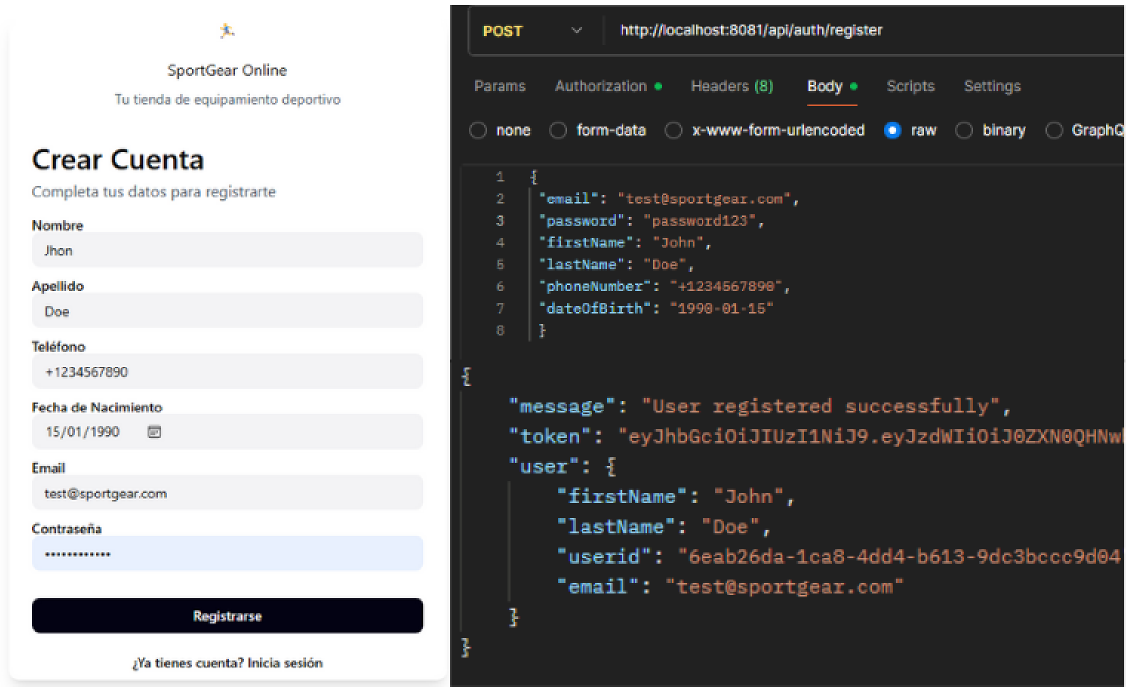
—

4.3 Evidence of Web GUI Integration

This section presents visual evidence of the complete integration between the React frontend and both backend services (Java for authentication and Python for business logic). Each screenshot demonstrates a fully functional user interface component communicating with the respective REST APIs.

Register

The user registration interface allows new users to create accounts by providing their personal information. The React form validates input fields client-side before sending a POST request to the Java authentication backend at `/api/auth/register`. Upon successful registration, the system automatically generates a JWT token and logs the user in.


SportGear Online

Tu tienda de equipamiento deportivo

Crear Cuenta

Completa tus datos para registrarte

Nombre	Jhon
Apellido	Doe
Teléfono	+1234567890
Fecha de Nacimiento	15/01/1990
Email	test@sportgear.com
Contraseña	••••••••••••

POST http://localhost:8081/api/auth/register

Body: raw

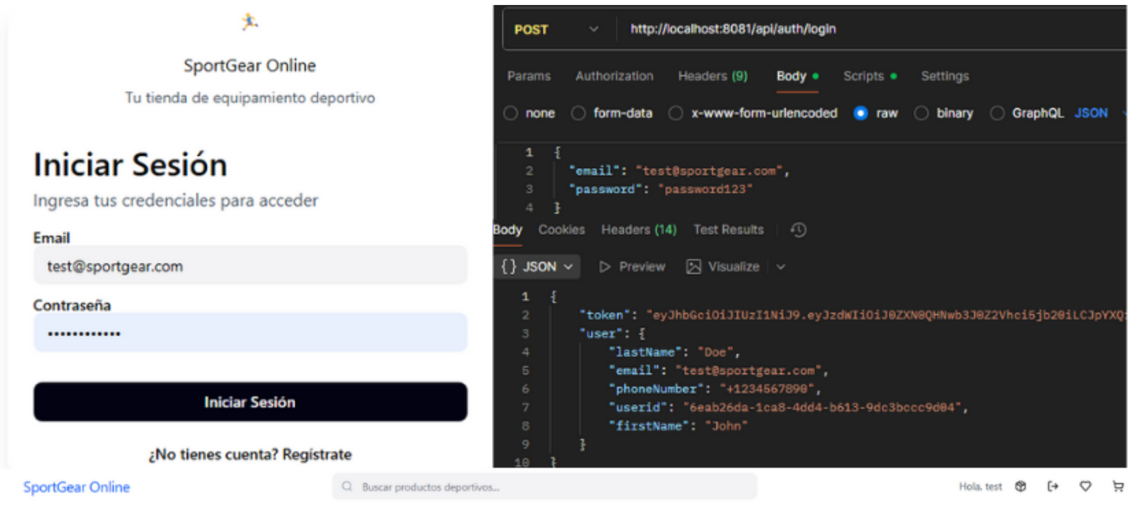
```
{
 "email": "test@sportgear.com",
 "password": "password123",
 "firstName": "John",
 "lastName": "Doe",
 "phoneNumber": "+1234567890",
 "dateOfBirth": "1990-01-15"
}
```

```
{
    "message": "User registered successfully",
    "token": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ0ZXN0QHNw
    "user": {
        "firstName": "John",
        "lastName": "Doe",
        "userid": "6eab26da-1ca8-4dd4-b613-9dc3bccc9d04
        "email": "test@sportgear.com"
    }
}
```

Figure 4.5: User registration interface showing the registration form with email, password, and personal information fields. This component communicates with the Java backend authentication service at `POST /api/auth/register`, successfully creating new user accounts and receiving JWT tokens for immediate session initialization.

Login

The authentication interface provides secure access to registered users. This component handles credential validation, token storage in browser's local storage, and automatic redirection to the main application view. The JWT token received is used for all subsequent authenticated API requests.


SportGear Online

Tu tienda de equipamiento deportivo

Iniciar Sesión

Ingresa tus credenciales para acceder

Email: test@sportgear.com

Contraseña: ••••••••••••

¿No tienes cuenta? Regístrate

```
POST http://localhost:8081/api/auth/login
```

```
{
  "email": "test@sportgear.com",
  "password": "password123"
}
```

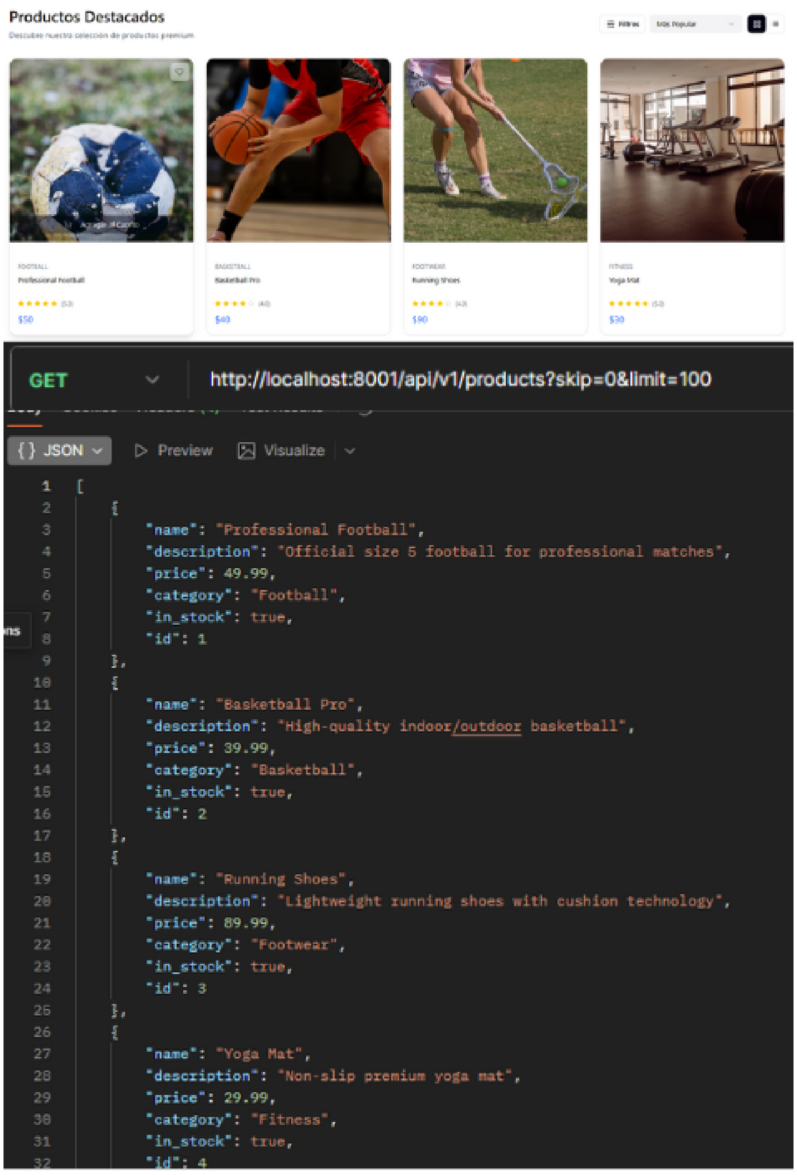
Response Body (JSON):

```
{
    "token": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ0ZXN0QHNwb3J0Z2Vhci5jb20iLCJpYXQ
    "user": {
        "lastName": "Doe",
        "email": "test@sportgear.com",
        "phoneNumber": "+1234567890",
        "userid": "6eab26da-1ca8-4dd4-b613-9dc3bccc9d04",
        "firstName": "John"
    }
}
```

Figure 4.6: Authentication interface displaying the login form where users enter their credentials. Upon submission, the frontend sends a `POST` request to the Java backend (`/api/auth/login`), receives a JWT token, and stores it in local storage for subsequent authenticated requests to both backend services.

Products

The product catalog displays all available sporting goods items in a responsive grid layout. Users can browse products, view detailed information including pricing and availability, and add items to their shopping cart. The component implements pagination and real-time stock status updates.


Productos Destacados

Descubre nuestra colección de productos premium

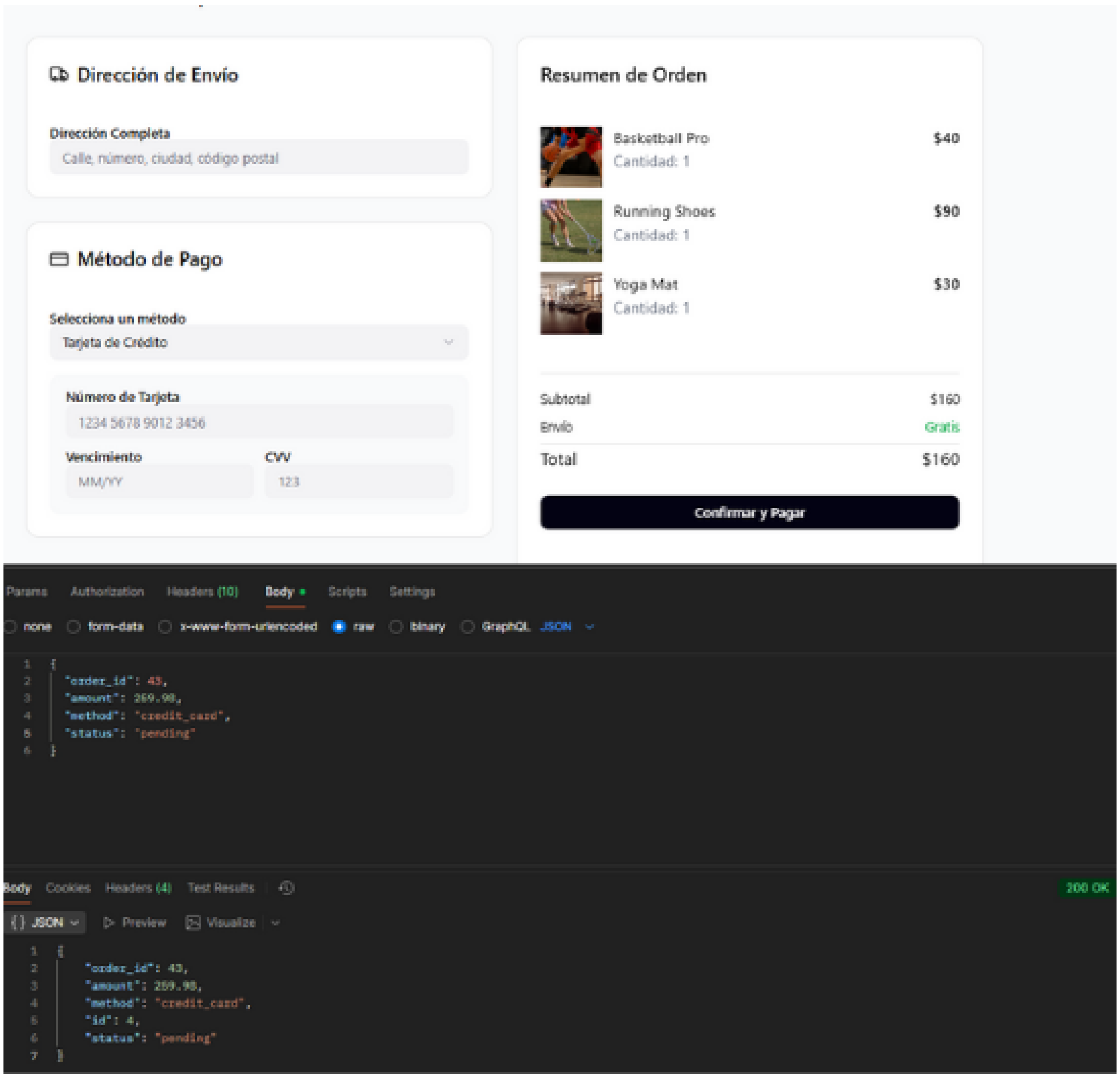
```
GET http://localhost:8001/api/v1/products?skip=0&limit=100
```

```
[
    {
        "name": "Professional Football",
        "description": "Official size 5 football for professional matches",
        "price": 49.99,
        "category": "Football",
        "in_stock": true,
        "id": 1
    },
    {
        "name": "Basketball Pro",
        "description": "High-quality indoor/outdoor basketball",
        "price": 39.99,
        "category": "Basketball",
        "in_stock": true,
        "id": 2
    },
    {
        "name": "Running Shoes",
        "description": "Lightweight running shoes with cushion technology",
        "price": 89.99,
        "category": "Footwear",
        "in_stock": true,
        "id": 3
    },
    {
        "name": "Yoga Mat",
        "description": "Non-slip premium yoga mat",
        "price": 29.99,
        "category": "Fitness",
        "in_stock": true,
        "id": 4
```

Figure 4.7: Product catalog view showing the grid layout of available sporting goods items. This component fetches data from the Python FastAPI backend via `GET /api/v1/products`, displaying product information including name, description, price, category, and stock availability with real-time data synchronization.

Payment

The payment processing interface enables users to complete their purchases securely. The form collects payment method details, validates transaction amounts against order totals, and communicates with the business logic backend. The system supports multiple payment methods including credit cards, debit cards, and digital wallets.



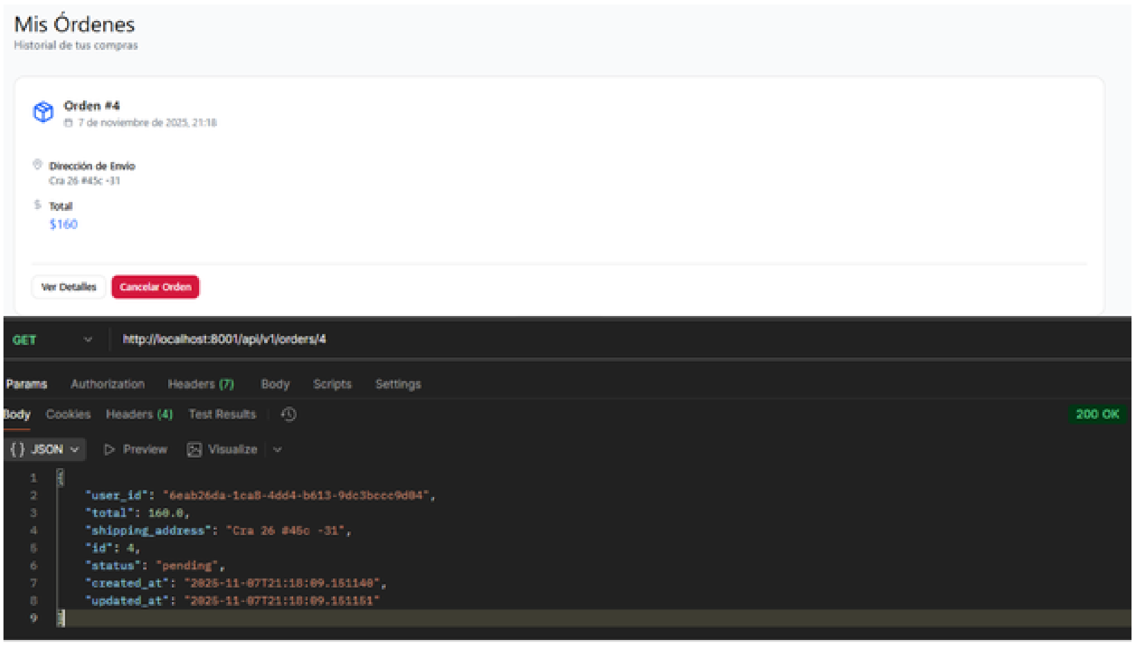
```
{
  "order_id": 43,
  "amount": 259.98,
  "method": "credit_card",
  "status": "pending"
}
```

```
{
    "order_id": 43,
    "amount": 259.98,
    "method": "credit_card",
    "id": 4,
    "status": "pending"
}
```

Figure 4.8: Payment processing interface allowing users to complete transactions for their orders. The form collects payment method details and communicates with the Python backend at `POST /api/v1/payments`, creating payment records with proper order association, amount validation, and status tracking (pending, completed, failed).

Order

The order management interface provides users with comprehensive visibility into their purchase history. Users can view order details, track shipping status, and access order-related information such as total amounts and delivery addresses. The interface updates in real-time as order statuses change.



```
{
    "user_id": "6eab26da-1ca8-4dd4-b613-9dc3bccc9d84",
    "total": 160.0,
    "shipping_address": "Cra 26 #45c -31",
    "id": 4,
    "status": "pending",
    "created_at": "2025-11-07T21:18:09.151140",
    "updated_at": "2025-11-07T21:18:09.151151"
}
```

Figure 4.9: Order management interface displaying the user's purchase history and current orders. This view retrieves data from `GET /api/v1/orders` endpoint of the Python backend, showing order details including total amount, shipping address, order status (pending, processing, completed, cancelled), and creation timestamps with proper authentication token validation.

4.4 Summary

This chapter presented the testing and validation results of the **SportGear Online** platform. Automated unit tests confirmed the functional integrity of both backends, while Postman and GUI evidence verified end-to-end communication between services and the frontend. The results demonstrate that the distributed architecture successfully supports authentication, product management, and payment workflows across all system layers.

Chapter 5

Discussion and Analysis

5.1 Introduction

This chapter discusses the main results and findings obtained during the implementation of the **SportGear Online** platform in the third workshop. The analysis focuses on how the system design was transformed into a working distributed application composed of two backends and a frontend. It also evaluates the integration between technologies, testing practices, and architectural decisions that ensured functionality, scalability, and maintainability.

5.2 Integration of Business and Technical Perspectives

The development of the SportGear Online platform maintained a strong alignment between the original business model and its technical implementation. The modular structure designed in previous workshops was successfully reflected in code: the **Java Spring Boot** backend manages authentication and user control, while the **Python FastAPI** backend implements product and sales logic. This separation of concerns made it possible to preserve business rules independently of the authentication process, ensuring flexibility for future extensions.

From a business perspective, this implementation guarantees that the platform's core value proposition — a reliable and user-friendly online store — is maintained through a technically consistent and secure architecture.

5.3 Usability and User Experience

The frontend, developed in **React**, connects seamlessly with both backends through REST APIs. Its design follows a clean and minimal interface that allows users to easily register, log in, browse products, and manage purchases.

User experience considerations guided the structure of API calls, component hierarchy, and state management. This approach not only improves usability but also demonstrates the feasibility of integrating a multi-service backend with a modern frontend framework.

Although this stage focused on system integration rather than design optimization, the results show that the interface can be easily extended with additional features such as product search, filters, and responsive design.

5.4 Object-Oriented Structure and System Consistency

The use of object-oriented principles in both backends ensured a consistent and maintainable system structure. In **Spring Boot**, the model-view-controller (MVC) architecture clearly separates business logic, data access, and presentation layers. Meanwhile, in **FastAPI**, dependency injection and asynchronous routes were applied to achieve modularity and high performance.

Each component communicates through defined contracts (DTOs and schemas), which reduces coupling and increases maintainability. This design follows best practices in modern distributed architectures, promoting scalability and clarity in code evolution.

5.5 Architectural Design and Scalability

The distributed architecture implemented in this workshop proved to be effective for modular deployment. Both backends expose APIs that can be independently scaled or replaced without affecting other components.

The system supports asynchronous operations, concurrent requests, and secure access via JWT tokens. These features demonstrate that the architecture can support a growing number of users and transactions without compromising reliability.

This flexibility prepares the platform for future integration with containerization and orchestration tools, which will be developed in the next workshop.

5.6 Process Efficiency and Testing Practices

Process efficiency was ensured through the application of **Test-Driven Development (TDD)**. Unit tests were implemented using *JUnit* in the Java backend and *pytest* in the Python backend.

These tests validated user authentication, product registration, and purchase workflows, guaranteeing the correct behavior of the main services. This continuous testing process helped detect and fix logical errors early, increasing the overall reliability of the system.

In addition, automated test reports provided valuable metrics on code coverage and stability, confirming that the functional design from previous workshops was effectively implemented.

5.7 Evaluation of Methodology

The combination of tools and methodologies used in Workshop 3 proved to be highly effective for achieving the project's objectives. Each component of the system contributed to a complete and functional architecture:

- **Spring Boot:** Managed secure user authentication and role handling.
- **FastAPI:** Implemented product management and sales logic with high performance.
- **React:** Provided an interactive frontend connected to both backends.
- **JWT:** Ensured secure and stateless communication between services.
- **TDD:** Maintained continuous validation of business requirements.

This integration of frameworks and methodologies demonstrated how a theoretical design could be transformed into a functional, reliable system.

5.8 Limitations

Despite the successful implementation, some limitations were identified:

- The system has not yet been deployed in a production environment; it currently runs locally on separate servers.
- Testing focused mainly on functionality; performance and stress testing will be conducted in future stages.
- The frontend, although functional, can still be improved in design and accessibility.
- Continuous integration and automated deployment have not yet been implemented.

These aspects will be addressed in the next workshop, where containerization and CI/CD pipelines will be incorporated.

5.9 Summary

The analysis confirms that Workshop 3 successfully transformed the SportGear Online project from conceptual design into a working distributed system. The implementation validated the feasibility of the proposed architecture, confirmed interoperability between multiple backends, and ensured secure and tested communication through JWT and TDD.

The results demonstrate technical maturity and readiness for the next phase, where the system will be deployed and validated through Docker containers, acceptance testing, and CI/CD automation.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The **SportGear Online** project successfully implemented a complete distributed architecture integrating two backends — one developed in Java using *Spring Boot* and another in Python using *FastAPI* — along with a frontend built in *React*.

This stage demonstrated that it is possible to maintain **security, modularity, and scalability** while combining heterogeneous technologies within a single unified platform.

The main conclusions drawn from this work are as follows:

- The integration between modules was successfully achieved, ensuring secure communication through **JWT** and **REST APIs**.
- The adoption of **Test-Driven Development (TDD)** practices using *JUnit* and *pytest* guaranteed continuous validation of requirements and code quality.
- The **separation of responsibilities** between authentication and business logic promoted a clean, maintainable, and easily extensible architecture.
- The **React** frontend effectively consumed the APIs from both backends, demonstrating correct full-stack interaction.
- The resulting system provides a robust functional foundation for the next development phase focused on deployment and automation.

In summary, this workshop confirmed that the system design proposed in earlier stages can be successfully implemented in a real distributed environment, maintaining coherence between conceptual design and technical execution.

6.2 Achievements

Throughout this workshop, several important technical and methodological goals were successfully accomplished:

- Development and integration of two independent backends using **Spring Boot** (Java) for authentication and **FastAPI** (Python) for business logic and data management.
- Implementation of a **React**-based frontend capable of consuming both backends' REST APIs, achieving a functional and responsive user interface.

- Configuration of secure communication between services through the use of **JSON Web Tokens (JWT)** for authentication and authorization.
- Application of **Test-Driven Development (TDD)** principles using *JUnit* and *pytest*, ensuring continuous verification of functionalities and reliable test coverage.
- Definition of clear **API endpoints and routes**, supporting modular interaction between system components.
- Establishment of a consistent project structure with separate layers for presentation, business logic, and persistence, following clean architecture practices.

These achievements collectively demonstrate that the distributed system design can be effectively implemented with modern frameworks and best practices in full-stack development.

6.3 Challenges

During the development and integration process, several challenges were encountered and addressed:

- Managing interoperability between the **Java** and **Python** backends required careful design of API contracts and response formats.
- Synchronizing asynchronous operations and ensuring data consistency between services posed technical challenges, especially in multi-request flows.
- Securing endpoints through JWT validation introduced complexity in token management and session control.
- Establishing a unified testing workflow for two different backends demanded a clear separation of test environments and dependencies.
- Coordinating frontend API consumption and backend updates required continuous testing and refactoring to maintain stability.

Despite these difficulties, the team successfully resolved integration issues and achieved a stable, functional distributed architecture that set the foundation for the next stage of deployment and automation.

6.4 Future Work

The next stage of the project, corresponding to **Workshop 4**, will focus on deployment, testing, and automation. The goal is to transform the functional system into a production-ready environment using modern DevOps practices. The planned tasks include:

- **Containerization with Docker and Docker Compose:** Each system component (Java backend, Python backend, and React frontend) will be packaged into independent containers to ensure consistency, portability, and scalability across environments.
- **Acceptance Testing with Cucumber:** Development of feature files and step definitions to validate user stories through behavior-driven testing (BDD).

- **API Stress Testing with Apache JMeter:** Design and execution of stress test plans to evaluate system performance under concurrent user requests.
- **CI/CD Pipeline with GitHub Actions:** Creation of an automated workflow to run tests, build Docker images, and prepare continuous integration pipelines.

These activities will validate the stability, scalability, and maintainability of the system in realistic deployment conditions, preparing SportGear Online for its final production delivery.

6.5 Final Reflection

The project showed that combining business modeling and software design methods can produce stronger and more meaningful system architectures. The experience of designing SportGear Online also demonstrates the importance of teamwork, planning, and clear communication between technical and business roles.

This work not only provided academic experience but also practical understanding of how modern e-commerce systems are planned, structured, and prepared for implementation, improving our skills for collaborative work as required in the software development industry.

Chapter 7

Reflection

This chapter reflects on the learning experience gained while designing SportGear Online. It describes the skills developed, the main challenges faced, and how the project would be approached differently in the future.

7.1 Learning and Skills Developed

Working on this project improved both technical and soft skills:

- **Modelling and design:** I gained practical experience using Business Model Canvas, User Story Mapping, CRC Cards, UML class diagrams, and BPMN. These tools helped me see how business needs translate into software structure.
- **System thinking:** The project strengthened my ability to think across layers—from business strategy to user flows to technical architecture.
- **Communication:** Writing clear user stories and CRC Cards improved how I explain technical ideas to non-technical stakeholders.
- **Teamwork and coordination:** Collaborating with teammates reinforced planning, division of work, and merging different design pieces into one coherent model.

7.2 Challenges and How They Were Addressed

Several challenges appeared during the work:

- **Maintaining consistency:** Ensuring that the BMC, user stories, CRC Cards, and UML diagrams matched each other required repeated checks. We addressed this by scheduling review sessions and keeping a shared document linking each user story to its related classes and processes.
- **Scope definition:** Deciding what to include in the MVP required trade-offs. The user story map helped prioritize essential features and delay lower-priority items.
- **Technical uncertainty:** Some architectural choices (e.g., precise deployment details or third-party integrations) remained hypothetical. We mitigated risk by sketching multiple deployment options and noting assumptions in the documentation.

7.3 What I Would Do Differently

If I were to repeat this project, I would:

- Start earlier with a small runnable prototype to validate critical design decisions sooner.
- Run brief usability tests on early UI mockups to get direct user feedback before finalizing flows.
- Keep a tighter change log for diagrams and models so each revision is easier to track and justify.

7.4 Impact on Future Work

This project provided a strong foundation for practical software development. The experience of connecting business models to software architecture will guide future projects, especially those that require both strategic and technical alignment. The methods used here are reusable and will help when moving from design to implementation and testing.

7.5 Personal Reflection

Designing SportGear Online was a valuable learning process. I learned to balance clarity and detail, and to use simple models to guide complex technical decisions. The project highlighted the importance of continuous review and communication across team roles. Overall, the exercise increased my confidence in leading the early design stages of software projects and prepared me for the next step: implementation and real-world validation.

References

Auth0 (2025), 'Manual de jwt (json web tokens handbook)', https://auth0.com/resources/ebooks/jwt-handbook-es. Accessed November 7, 2025.

Beck, K. (2003), *Test-Driven Development: By Example*, Addison-Wesley, Boston, MA, USA. ISBN: 9780321146533.

Cohesity (2025), 'Data management security - cohesity glossary', https://www.cohesity.com/glossary/data-management-security/. Accessed November 7, 2025.

GeeksforGeeks (2024), 'Backend development - geeksforgeeks', https://www.geeksforgeeks.org/blogs/backend-development/. Accessed November 7, 2025.

GeeksforGeeks (2025*a*), 'Differences between software testing and quality assurance', https://www.geeksforgeeks.org/differences-between-software-testing-and-quality-assurance/. Accessed November 7, 2025.

GeeksforGeeks (2025*b*), 'List of front-end technologies - geeksforgeeks', https://www.geeksforgeeks.org/html/list-of-front-end-technologies/. Accessed November 7, 2025.

Reaboi, P. (2024), 'Understanding full stack development architecture: A comprehensive guide', https://medium.com/@p.reaboi.frontend/understanding-full-stack-development-architecture-a-comprehensive-guide-548f8cba6d91. Accessed November 7, 2025.

Suárez, L. (2025), 'Fastapi best practices and design patterns: Building quality python apis', https://medium.com/@lautisuarez081/fastapi-best-practices-and-design-patterns-building-quality-python-apis-31774ff3c28a. Accessed November 7, 2025.