



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 Babatiba 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 Babatiba 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

1	Introduction	1
1.1	Background	1
1.2	Problem Statement	1
1.3	Aims and Objectives	1
1.3.1	General Aim	1
1.3.2	Specific Objectives	1
1.4	Solution Approach	2
1.5	Summary of Contributions and Achievements	2
1.6	Organization of the Report	2
2	Literature Review	4
2.1	Software Architecture and Full-Stack Development	4
2.2	Backend Frameworks and Integration Technologies	4
2.2.1	Spring Boot for Authentication	4
2.2.2	FastAPI for Business Logic	4
2.3	Frontend Technologies	5
2.4	Software Testing and Quality Assurance	5
2.5	Security and Data Management	5
2.6	Summary	5
3	Methodology	6
3.1	System Architecture	6
3.2	Backend 1: Spring Boot Authentication Service	7
3.2.1	Structure and Components	7
3.2.2	Authentication Controller	7
3.2.3	JWT Utility	8
3.2.4	Web Security Configuration	9
3.2.5	Backend 2: FastAPI Business Logic Service	10
3.2.6	Project Organization	10
3.2.7	Database Connection	11
3.2.8	Data Models and Schemas	11
3.2.9	CRUD Operations	11
3.3	REST API Endpoints	12
3.3.1	Authentication API (Spring Boot)	12
3.3.2	Business Logic API (FastAPI)	13
3.4	Testing Methodology	14
3.4.1	Java Backend Testing	14
3.4.2	Python Backend Testing	15
3.5	Summary	15

4 Results	16
4.1 Unit Test Execution Results	16
4.1.1 Java Backend	16
4.1.2 Python Backend	16
4.2 API Testing with Postman	17
4.2.1 Java Authentication API (Spring Boot)	17
4.2.2 Python Business Logic API (FastAPI)	18
4.3 Evidence of Web GUI Integration	19
4.4 Summary	24
5 Discussion and Analysis	25
5.1 Introduction	25
5.2 Integration of Business and Technical Perspectives	25
5.3 Usability and User Experience	25
5.4 Object-Oriented Structure and System Consistency	25
5.5 Architectural Design and Scalability	26
5.6 Process Efficiency and Realism	26
5.7 Evaluation of Methodology	26
5.8 Limitations	26
5.9 Summary	27
6 Conclusions and Future Work	28
6.1 Conclusions	28
6.2 Achievements	28
6.3 Challenges	29
6.4 Future Work	29
6.5 Final Reflection	29
7 Reflection	30
7.1 Learning and Skills Developed	30
7.2 Challenges and How They Were Addressed	30
7.3 What I Would Do Differently	31
7.4 Impact on Future Work	31
7.5 Personal Reflection	31
References	32

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 \(2025b\)](#), 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 \(2025a\)](#). 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.

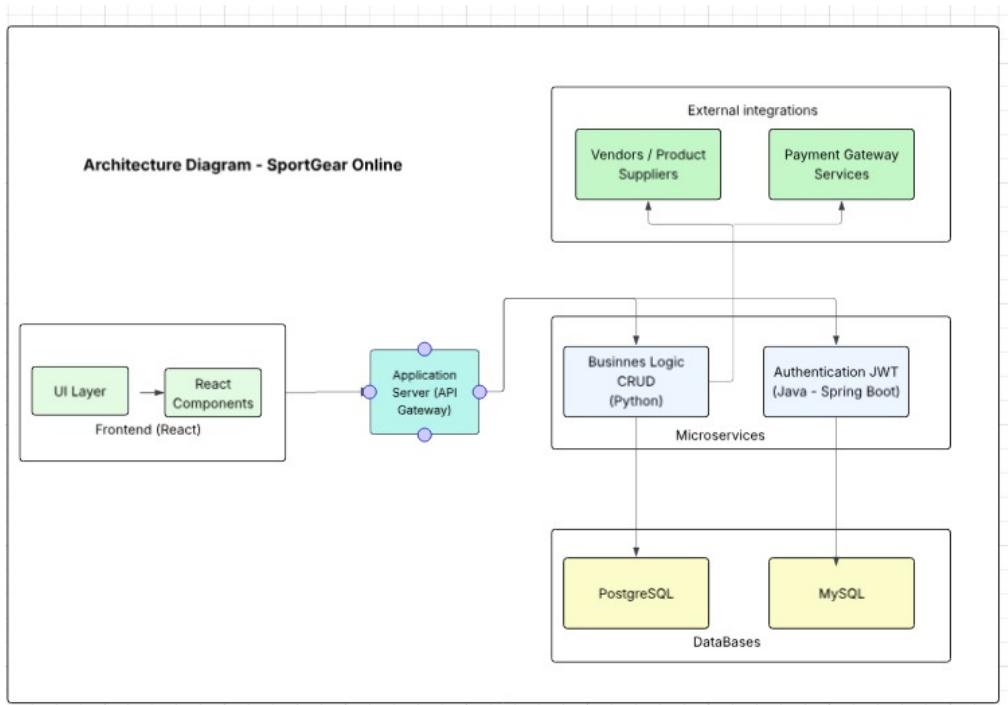


Figure 3.1: General architecture of the SportGear Online platform showing the two backends and the frontend integration.

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:

```

1 src/main/java/com/sportgear/auth/
2   |-- controller/
3   |   ----|-- AuthController.java
4   |-- model/
5   |   ----|-- User.java
6   |-- repository/
7   |   ----|-- UserRepository.java
8   |-- security/
9   |   ----|-- JwtUtils.java
10  |   ----|-- SecurityConfig.java
11  |-- 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.

```

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.

```
@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.

```

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()) // ☐ Necessary for APIs REST
            .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS)) // ☐ API without sessions
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/api/auth/**").permitAll() // ☐ all the authentication 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.

```

1 app/
2   |-- main.py
3   |-- routes/
4   |   |-- products.py
5   |   |-- orders.py
6   |   |-- payments.py
7   |-- models/
8   |   |-- product.py
9   |   |-- order.py
10  |   |-- payment.py
11  |-- schemas/
12  |   |-- product_schema.py
13  |   |-- order_schema.py
14  |   |-- payment_schema.py
15  |-- 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.

```

1   from sqlalchemy import create_engine
2   from sqlalchemy.ext.declarative import declarative_base
3   from sqlalchemy.orm import sessionmaker
4   from app.config import settings
5
6   engine = create_engine(settings.DATABASE_URL)
7   SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
8   Base = declarative_base()
9
10  def get_db():
11      db = SessionLocal()
12      try:
13          yield db
14      finally:
15          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.

```

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)

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)

```



```

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")

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.

```

1 @router.get("/products")
2 def get_products(db: Session = Depends(get_db)):
3     return db.query(Product).all()
4
5 @router.post("/products")

```

```

6 def create_product(product: ProductSchema, db: Session = Depends(get_db)):
7     new_product = Product(**product.dict())
8     db.add(new_product)
9     db.commit()
10    db.refresh(new_product)
11    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.

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.

```

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.

```

@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.

```

1 @SpringBootTest
2 @AutoConfigureMockMvc
3 class AuthControllerTest {
4
5     @Autowired
6     private MockMvc mockMvc;
7
8     @Test
9     void testLoginShouldReturnJwtToken() throws Exception {
10         mockMvc.perform(post("/api/auth/login")
11                         .contentType(MediaType.APPLICATION_JSON)
12                         .content("{\"username\":\"admin\", \"password\":\"1234\"}"))
13             .andExpect(status().isOk());
14     }
15 }

```

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.

```
1 from fastapi.testclient import TestClient
2 from app.main import app
3
4 client = TestClient(app)
5
6 def test_get_products():
7     response = client.get("/api/v1/products")
8     assert response.status_code == 200
9     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.EcommerceBackendApplicationTests
[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  28%  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/api>

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

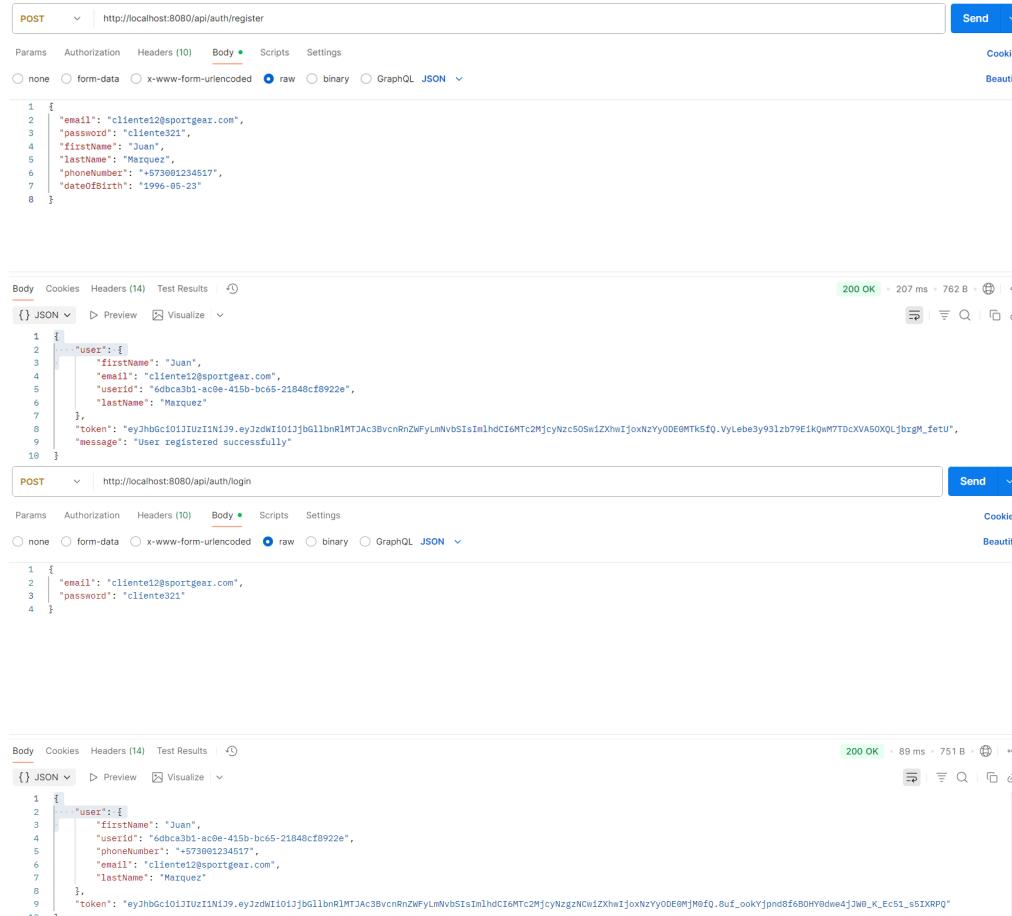


Figure 4.3: Postman request validating Java backend authentication endpoints for user registration and login. The response includes a valid JWT token.

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.

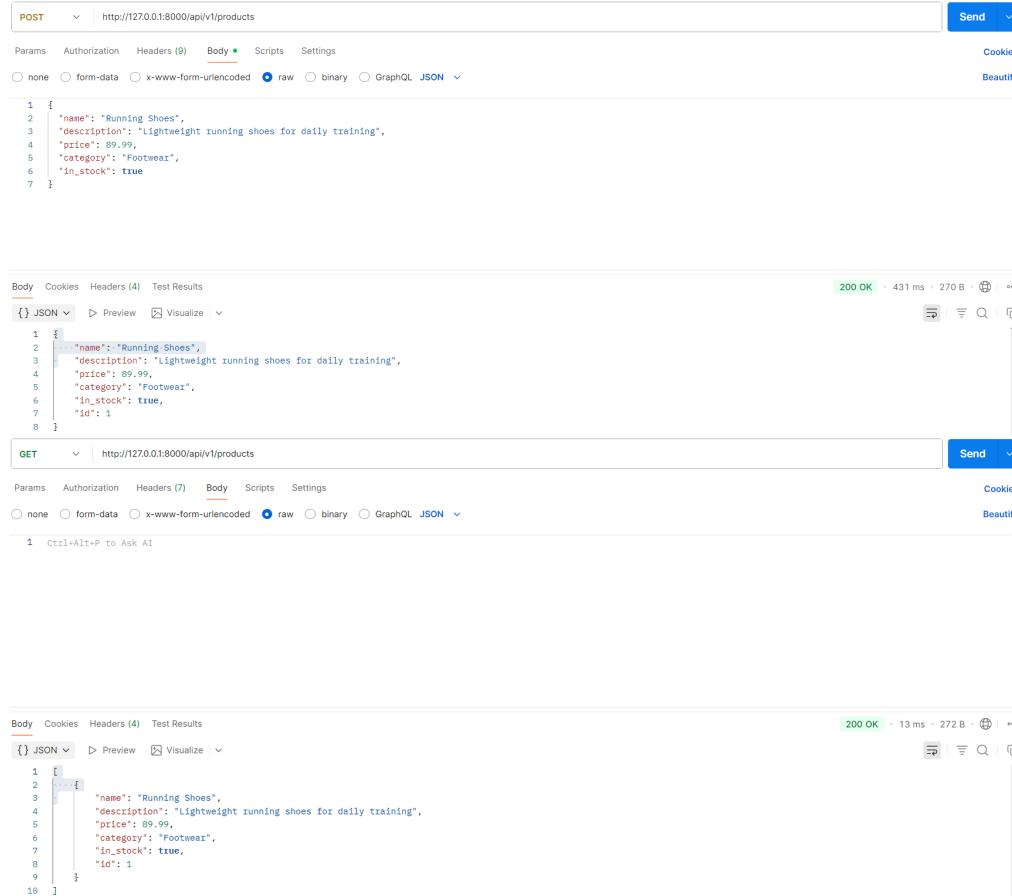


Figure 4.4: Postman request examples for FastAPI business logic endpoints. Successful GET and POST requests confirm product data persistence and response validation.

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.

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.

The screenshot shows the SportGear Online login page and a Postman API request for /api/auth/login.

SportGear Online Login Page:

- Header:** SportGear Online - Tu tienda de equipamiento deportivo
- Title:** Iniciar Sesión
- Text:** Ingresa tus credenciales para acceder
- Form Fields:**
 - Email: test@sportgear.com
 - Contraseña: [REDACTED]
- Buttons:**
 - Iniciar Sesión
 - ¿No tienes cuenta? Regístrate

Postman API Request:

- Method:** POST
- URL:** http://localhost:8081/api/auth/login
- Body:** raw JSON (selected)


```

1 {
2   "email": "test@sportgear.com",
3   "password": "password123"
4 }
```
- Response Body:** JSON (selected)


```

1 {
2   "token": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ0ZXN0QHNwb3J0Z2Vhc15jb20iLCJpYX0i
3   "user": {
4     "lastName": "Doe",
5     "email": "test@sportgear.com",
6     "phoneNumber": "+1234567890",
7     "userId": "6eab26da-ica8-4dd4-b613-9dc3bcc9d84",
8     "firstName": "John"
9   }
10 }
```

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.

The top part of the image shows a product catalog interface with four items displayed in a grid:

- FOOTBALL**: Professional Football, 4 stars, \$50.
- BASKETBALL**: Basketball Pro, 4 stars, \$40.
- FOOTWEAR**: Running Shoes, 4 stars, \$80.
- FITNESS**: Yoga Mat, 4 stars, \$30.

The bottom part of the image shows a terminal window displaying a JSON API response for the products:

```

GET http://localhost:8001/api/v1/products?skip=0&limit=100
[{"id": 1, "name": "Professional Football", "description": "Official size 5 football for professional matches", "price": 49.99, "category": "Football", "in_stock": true}, {"id": 2, "name": "Basketball Pro", "description": "High-quality indoor/outdoor basketball", "price": 39.99, "category": "Basketball", "in_stock": true}, {"id": 3, "name": "Running Shoes", "description": "Lightweight running shoes with cushion technology", "price": 89.99, "category": "Footwear", "in_stock": true}, {"id": 4, "name": "Yoga Mat", "description": "Non-slip premium yoga mat", "price": 29.99, "category": "Fitness", "in_stock": true}]
  
```

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.

The screenshot shows a payment processing interface. On the left, there's a form for 'Dirección de Envío' (Shipping Address) with a placeholder 'Calle, número, ciudad, código postal'. Below it is a section for 'Método de Pago' (Payment Method) with a dropdown menu set to 'Tarjeta de Crédito'. Underneath are fields for 'Número de Tarjeta' (Card Number) containing '1234 5678 9012 3456', 'Vencimiento' (Expiry) with 'MM/YY' and '123', and 'CVV'. On the right, there's a 'Resumen de Orden' (Order Summary) table showing three items: 'Basketball Pro' (\$40), 'Running Shoes' (\$90), and 'Yoga Mat' (\$30). The total 'Subtotal' is \$160, with 'Envío' (Shipping) listed as 'Gratis' (Free). A large 'Confirmar y Pagar' (Confirm and Pay) button is at the bottom. Below the interface is a terminal window showing a cURL command to post data to '/api/v1/payments' in JSON format:

```

Params Authorization Headers (10) Body Scripts Settings
 none  form-data  x-www-form-urlencoded  raw  binary  GraphQL  JSON
1 {
2   "order_id": 43,
3   "amount": 269.98,
4   "method": "credit_card",
5   "status": "pending"
6 }

```

Body Cookies Headers (4) Test Results 200 OK

```

{} JSON  Preview  Visualize
1 {
2   "order_id": 43,
3   "amount": 269.98,
4   "method": "credit_card",
5   "id": 4,
6   "status": "pending"
7 }

```

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.

The screenshot shows two parts: a user interface and a Postman API request.

User Interface: A web page titled "Mis Órdenes" (My Orders) showing a single order. The order details are as follows:

- Orden #4
- 7 de noviembre de 2025, 21:18
- Dirección de Envío: Cra 26 #45c -31
- Total: \$160

Buttons at the bottom include "Ver Detalles" and "Cancelar Orden".

Postman API Request:

- Method: GET
- URL: `http://localhost:8001/api/v1/orders/4`
- Headers (4): Includes Authorization token validation.
- Body: JSON response showing the order details:

```

1  {
2    "user_id": "6eab26da-1ca8-4dd4-b613-9dc3bcc9d04",
3    "total": 160.0,
4    "shipping_address": "Cra 26 #45c -31",
5    "id": 4,
6    "status": "pending",
7    "created_at": "2025-11-07T21:18:09.151148",
8    "updated_at": "2025-11-07T21:18:09.151151"
9  }
  
```

The Postman response status is 200 OK.

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 findings of the project and analyzes how each result contributes to the design of SportGear Online. The discussion connects the business, user, and technical parts of the project and evaluates how well the proposed design meets the original objectives.

5.2 Integration of Business and Technical Perspectives

One of the most important results of the project is the clear connection between the business strategy and the system architecture. The Business Model Canvas (BMC) defines how the platform creates and delivers value, while the UML Class Diagram and the Architecture Diagrams translate that strategy into a technical structure.

This integration shows that the system is not only functional but also aligned with the company's goals. For example, the "Value Proposition" in the BMC — offering a reliable and easy-to-use online store — is supported by the modular design of the system, which ensures that the website can be expanded and maintained efficiently.

5.3 Usability and User Experience

The use of User Stories and User Story Mapping made it possible to design a system focused on user needs. Each function was analyzed from the customer's point of view, which helped define the order in which features should be implemented.

The Web UI Mockups also support this goal by providing a simple and intuitive interface. Navigation, search filters, and checkout steps were designed to minimize user effort and improve satisfaction. This alignment between functional design and user experience increases the platform's potential success in the market.

5.4 Object-Oriented Structure and System Consistency

The use of CRC Cards and UML Class Diagrams gave the project a strong internal structure. Each class has a clear role and communicates only with the classes that are necessary.

This object-oriented approach improves reusability and maintenance, as recommended by Larman ? and Booch et al. ?. It also makes future updates easier, since new features can be added without breaking existing components.

5.5 Architectural Design and Scalability

The Integration and Deployment Architecture Diagrams show that the design can grow over time. The system uses independent modules connected by APIs, which makes it easier to replace or improve each part.

For example, the payment service can be updated or changed without affecting the rest of the platform. This design follows modern practices of enterprise development ? and ensures that the platform can adapt to new technologies or market changes.

5.6 Process Efficiency and Realism

The BPMN Diagrams connect business activities with the system workflow. This makes the model realistic because it shows what happens in the real process — from when the customer makes an order until the product is delivered.

Modeling these processes helped detect possible bottlenecks, such as stock confirmation or payment verification, before implementation. As a result, the proposed system is more efficient and reliable.

5.7 Evaluation of Methodology

The combination of different modeling tools proved to be effective. Each one added value at a different stage:

- The **BMC** clarified business goals.
- **User Stories** identified user needs.
- **CRC Cards** defined responsibilities.
- **UML Diagrams** formalized relationships.
- **Architecture Diagrams** ensured scalability.
- **BPMN** connected all operations.

This progressive process made the design consistent and easy to understand. However, one limitation is that the models were not yet implemented in code, so technical validation through testing is still needed.

5.8 Limitations

Although the project produced a complete conceptual model, it has some limitations:

- No real implementation or system testing has been done yet.
- The user interface mockups are static and not interactive.
- The deployment plan assumes cloud hosting but does not include detailed cost analysis.

Future work could focus on turning these designs into a functional prototype to confirm the system's performance in a real environment.

5.9 Summary

The analysis shows that the project meets its objectives by combining business logic, user experience, and technical design. The tools used allowed for a complete view of the system, from strategy to implementation plan. The design is coherent, scalable, and aligned with modern software engineering principles, providing a strong base for future development of SportGear Online.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This project presented the conceptual and structural design of SportGear Online, an e-commerce platform specialized in selling sports products. The work followed a complete methodology that connects business strategy, user needs, and software architecture into one consistent framework.

The main conclusion is that using different modeling tools together gives a clear and organized view of how the system should work. Each tool played an important role:

- The **Business Model Canvas (BMC)** defined the business logic and main objectives.
- The **User Stories** and **User Story Mapping** focused on the user experience and product priorities.
- The **CRC Cards** and **UML Class Diagrams** structured the internal logic of the system.
- The **Integration and Deployment Diagrams** showed how the system can be implemented and connected to external services.
- The **BPMN Diagrams** described how business processes are executed in a realistic way.

All these results together form a solid base for the future development of the platform. The design is coherent, scalable, and easy to adapt to future needs. It also demonstrates the value of combining agile and object-oriented methods to create complete software solutions.

6.2 Achievements

During the project, several objectives were successfully completed:

- A clear link between business, users, and technology was created.
- The system structure was defined with UML models and CRC Cards.
- The workflows of the business were documented using BPMN.
- The main interface of the website was designed to improve usability.

These achievements confirm that the project's goals were met, providing both academic and practical results.

6.3 Challenges

Some challenges appeared during the process:

- Keeping a consistent connection between all modeling tools required time and coordination.
- Some diagrams needed several revisions to ensure logical accuracy.
- The translation from conceptual models to possible implementation raised technical questions not yet tested.

Despite these challenges, the project succeeded in maintaining coherence between strategy and technology.

6.4 Future Work

The next logical step for SportGear Online is to move from conceptual design to a real prototype or implementation. Possible directions for future work include:

- Developing a working prototype using modern web technologies (e.g., React, Spring Boot, or Node.js).
- Creating a real database and testing data flows between modules.
- Performing usability testing with potential users to improve the interface.
- Expanding the BPMN processes to include marketing, inventory management, and after-sales service.
- Analyzing cloud deployment options and cost estimation.

Implementing these improvements would allow technical validation of the proposed architecture and demonstrate the system's potential in real-world conditions.

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 and improve our skills for work in group as the Software Development industry needs.

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