**SAIGON UNIVERSITY**

**INFORMATION TECHNOLOGY FACULTY**



**PROJECT REPORT**

**TOPIC : QUALITY ASSURANCE FOR**

**EVERSHOP ONLINE FOOTWEAR RETAIL PLATFORM**

Instructor: Dr. Do Nhu Tai

Course: Software Testing

Performed by:

Chau Gia Anh - 3122411002

Duong Le Khanh - 3122411093

Dao Thi Thanh Tam - 3122411182

**Ho Chi Minh City, December 24, 2025**

TABLE OF CONTENTS

[CHAPTER 1: INTRODUCTION 1](#_Toc22325)

[1. Project overview 1](#_Toc31627)

[1.1. Background 1](#_Toc27330)

[1.2. Scope of the project 1](#_Toc27427)

[1.3. Agile and CI/CD process 2](#_Toc6114)

[1.3.1. Agile 2](#_Toc52)

[1.3.2. CI/CD 2](#_Toc21247)

[2. Requirement analysis 3](#_Toc28936)

[2.1. Current System Assessment 3](#_Toc26275)

[2.2. Business Context 4](#_Toc7641)

[2.2.1. Actor Diagram of the System 5](#_Toc23184)

[2.2.2. System Context Diagram 6](#_Toc15891)

[2.3. Technologies Used 6](#_Toc17540)

[2.4. Business requirements 7](#_Toc25710)

[2.4.1. Functional requirements 7](#_Toc22636)

[2.4.2. Non-functional Requirements 8](#_Toc30964)

[CHAPTER 2: SYSTEM DESIGN ANALYSIS AND SYSTEM ARCHITECTURE 10](#_Toc31431)

[1. System Design Analysis 10](#_Toc58)

[1.1. Business Process Diagrams 10](#_Toc28555)

[1.1.1. Product Management Business Process Diagram 10](#_Toc334)

[1.1.2. Purchase Business Process Diagram 11](#_Toc20684)

[1.1.3. Order Management Business Process Diagram 13](#_Toc10284)

[1.1.4. Access Control Management Business Process Diagram 14](#_Toc5102)

[1.2. Use Case Diagrams 15](#_Toc28851)

[1.2.1. Product Management Use Case 15](#_Toc22837)

[1.2.2. Purchase Management Use Case 16](#_Toc18034)

[1.2.3. Order Management Use Case 17](#_Toc24801)

[1.2.4. Access Control Management Use Case 18](#_Toc10191)

[1.3. User stories 18](#_Toc6727)

[1.4. Data 20](#_Toc3912)

[1.4.1. Conceptual ERD 20](#_Toc9341)

[1.4.2. Logical ERD 21](#_Toc15331)

[1.4.3. Physical ERD 22](#_Toc5522)

[1.5. Data Dictionary 26](#_Toc31246)

[2. System Architecture 27](#_Toc17520)

[2.1. Overall System Architecture Diagram 27](#_Toc22646)

[2.2. C4 Architecture 28](#_Toc29396)

[2.2.1. Level 1: Context Diagram 28](#_Toc20638)

[2.2.2. Level 2: Container Diagram 29](#_Toc24577)

[2.2.3. Level 3: Component Diagram 30](#_Toc26951)

[2.2.4. Level 4: Code-level Diagram 31](#_Toc21019)

[2.3. Communication View 33](#_Toc18652)

[2.4. Deployment View 35](#_Toc21326)

[CHAPTER 3: TEST PLAN 36](#_Toc8325)

[1. Introduction 36](#_Toc2346)

[1.1. Objectives 36](#_Toc23695)

[1.2. Scope 36](#_Toc32728)

[2. Test Items 36](#_Toc11586)

[2.1. Functional Testing 36](#_Toc20510)

[2.2. Non-Functional Testing 38](#_Toc21359)

[2.3. Out of Scope 40](#_Toc32322)

[3. Acceptance Criteria 40](#_Toc13862)

[3.1. Test Coverage 40](#_Toc8529)

[3.2. Test Case Execution 40](#_Toc13182)

[3.3. Defect Criteria 40](#_Toc7510)

[3.4. Code Coverage 40](#_Toc27744)

[4. Test Strategy 41](#_Toc20607)

[4.1. Testing Methodology 41](#_Toc2972)

[4.2. Testing Phases 41](#_Toc21065)

[4.3. Test Types 41](#_Toc19170)

[5. Environment and Resources 42](#_Toc7463)

[5.1. Test Environment 42](#_Toc19542)

[5.2. Human Resources 44](#_Toc32320)

[­5.3. Milestones 44](#_Toc21966)

[6. Deliverables 45](#_Toc17460)

[CHAPTER 4: TEST DESIGN 46](#_Toc8960)

[1. Overview 46](#_Toc11538)

[1.1. Objectives 46](#_Toc28203)

[1.2. Unit Testing 48](#_Toc21725)

[1.2.1. Definition and Characteristics 48](#_Toc2819)

[1.2.2. Layered Structure of Unit Tests in EverShop 48](#_Toc12624)

[1.2.3. Mocking Strategy 49](#_Toc3549)

[1.2.4. Benefits of Unit Testing 49](#_Toc19658)

[1.2.5. Unit Test Implementation in EverShop 50](#_Toc2423)

[1.3. Integration Testing 52](#_Toc16771)

[1.3.1. Structure of Core Components 53](#_Toc32724)

[1.3.2. Execution Process and Data Flow (Sequence Flow) 54](#_Toc16248)

[2. V-Model View 55](#_Toc18165)

[3. Agile CI/CD View 56](#_Toc31412)

[3.1. CI/CD Operational Process 56](#_Toc26755)

[3.1.1. Continuous Integration (CI) Phase 57](#_Toc3294)

[3.1.2. Continuous Deployment (CD) Phase 57](#_Toc31654)

[3.2. Benefits in Test Design and Execution 58](#_Toc27698)

[4. Testing Methods 59](#_Toc12948)

[4.1. Static Testing 59](#_Toc27473)

[4.2. Dynamic Testing 60](#_Toc24992)

[5. Advanced Testing Techniques Analysis 61](#_Toc4239)

[5.1 Manual Testing vs. Automated Testing 61](#_Toc13274)

[5.2 Gen AI 62](#_Toc18597)

[5.2.1 Key Application Areas 62](#_Toc8563)

[5.2.2 Expected Benefits 63](#_Toc22508)

[5.2.3 mplementation Roadmap 63](#_Toc7815)

[5.3. Test Process Automation 64](#_Toc18069)

[5.3.1 Automation Framework 64](#_Toc27902)

[5.3.2 Automated CI/CD Workflow 64](#_Toc10102)

[5.3.3 Outstanding Benefits of Automation 65](#_Toc4767)

[5.3.4 Challenges and Solutions 65](#_Toc22116)

[5.3.5. Application of Automation to the Project 65](#_Toc15496)

[CHAPTER 5: TEST SUMMARY REPORT 69](#_Toc29233)

[1. Testing Process Overview 69](#_Toc25516)

[2. Test Execution Report 69](#_Toc14277)

[2.1. Test Execution Summary 69](#_Toc25100)

[2.2. Detailed Test Report 71](#_Toc20213)

[2.2.1 Cypress API Testing 71](#_Toc26292)

[2.2.2 Jest Unit & Integration Testing 72](#_Toc24313)

[2.3 Code Coverage Report 73](#_Toc26323)

**LIST OF TABLES.**

[Table 1.2.4. 1. Functional Requirements Table 15](#_Toc217242981)

[Table 1.2.4. 2. Non-functional Requirements Table 16](#_Toc217242982)

[Table 3.3. 1. Functional Testing Table 44](#_Toc217242988)

[Table 3.3. 2. Non-Functional Testing Table 46](#_Toc217242989)

[Table 3.4.2. 1. Testing Phases Table 48](#_Toc217243001)

[Table 3.4.3. 1. Test Types Table 48](#_Toc217243009)

[Table 3.5.1. 1. Hardware Table 48](#_Toc217243016)

[Table 3.5.1. 2. Software Table 49](#_Toc217243017)

[Table 3.5.1. 3. Infrastructure Table 49](#_Toc217243018)

[Table 3.5.2. 1. Human Resources 50](#_Toc217243021)

[Table 3.5.3. 1. Milestones 51](#_Toc217243025)

[Table 4.1.1. 1. Test design on module 53](#_Toc217243030)

[Table 4.5.1. 1. Test Distribution Ratio 68](#_Toc217243035)

[Table 5.2.1. 1. Test Execution Summary Table 75](#_Toc217243042)

[Table 5.2.1. 2. Test Result Summary by Module 76](#_Toc217243043)

[Table 5.2.1. 3. Cross-Tool Test Comparison Table 77](#_Toc217243044)

[Table 5.2.2. 1. Cypress test API Table 78](#_Toc217243046)

[Table 5.2.2. 2. Jest for Unit & Integration Testing Table 79](#_Toc217243047)

[Table 5.2.3. 1. Code Coverage Table 79](#_Toc217243050)

**LIST OF FIGURES**

[Figure 1.2.2. 1. Actor Diagram of the System 11](#_Toc217243070)

[Figure 1.2.2. 2. System Context Diagram 12](#_Toc217243071)

[Figure 2.1.1. 1. Product Management Business Process Diagram 16](#_Toc217243080)

[Figure 2.1.1. 2. Purchase Business Process Diagram 17](#_Toc217243081)

[Figure 2.1.1. 3. Order Management Business Process Diagram 19](#_Toc217243082)

[Figure 2.1.1. 4. Access Control Management Business Process Diagram 20](#_Toc217243083)

[Figure 2.1.2. 1. Product Management Use Case 21](#_Toc217243090)

[Figure 2.1.2. 2. Purchase Management Use Case 22](#_Toc217243091)

[Figure 2.1.2. 3. Order Management Use Case 23](#_Toc217243092)

[Figure 2.1.2. 4. Access Control Management Use Case 24](#_Toc217243093)

[Figure 2.1.3. 1. Conceptual ERD 26](#_Toc217243101)

[Figure 2.1.3. 2. Logical ERD 27](#_Toc217243102)

[Figure 2.1.3. 3. Physical Product-Catalog 28](#_Toc217243103)

[Figure 2.1.3. 4. Physical Cart - Order 30](#_Toc217243104)

[Figure 2.1.3. 5. Physical Customer 31](#_Toc217243105)

[Figure 2.2.1. 1. Overall System Architecture Diagram 33](#_Toc217243111)

[Figure 2.2.2. 1. Level 1: Context Diagram 34](#_Toc217243186)

[Figure 2.2.2. 2. Level 2: Container Diagram 35](#_Toc217243187)

[Figure 2.2.2. 3. Level 3: Component Diagram 36](#_Toc217243188)

[Figure 2.2.2. 4. Code-level Diagram 37](#_Toc217243189)

[Figure 2.2.3. 1. Communication View 39](#_Toc217243271)

[Figure 2.2.4. 1. Deployment View 42](#_Toc217243279)

[Figure 4.1.2. 1. Unit Test Architecture in EverShop 56](#_Toc217243285)

[Figure 4.1.3. 1. Illustration of Integration Test Workflow in EverShop 59](#_Toc217243292)

[Figure 4.1.3. 2. Integration Test Execution Flow 60](#_Toc217243293)

[Figure 4.2. 1. V-Model Framework Diagram 61](#_Toc217243300)

[Figure 4.3.1. 1. Continuous Integration (CI) Phase Diagram 63](#_Toc217243307)

[Figure 4.3.1. 2. Continuous Deployment (CD) Phase Diagram 64](#_Toc217243308)

[Figure 4.5.3. 1. Automation Workflow Diagram 73](#_Toc217243316)

# 

# CHAPTER 1: INTRODUCTION

## **1. Project overview**

### **1.1. Background**

In recent years, the robust expansion of E-commerce has significantly accelerated the development and operation of online retail platforms to satisfy the escalating shopping demands of consumers. Within this landscape, footwear websites serve as a prototypical model, encompassing critical business operations such as product searching, detailed information viewing, shopping cart management, as well as online ordering and payment processing. These functionalities mandate that the system operates with high precision, stability, and guaranteed user data security.

However, alongside the proliferation of features and increasing system complexity, the risk of defects arising during software development and operation has also intensified. Errors related to functionality, user interface, data integrity, or performance can directly impair the user experience, diminish corporate reputation, and result in economic losses. Consequently, software testing has become an indispensable activity to ensure system quality prior to and during deployment.

In the context of modern software development methodologies such as Agile and CI/CD, which are now widely adopted, software testing is no longer restricted to the final stage but must be integrated throughout the entire Software Development Life Cycle (SDLC). Stemming from these practical requirements, this project focuses on researching and implementing software testing for a footwear website. The core objectives include establishing a test plan, applying appropriate testing methods and techniques, and synthesizing as well as evaluating results through comprehensive test reports.

### **1.2. Scope of the project**

This project focuses on software testing for an online footwear retail website. The scope of research and implementation includes: Analyzing business requirements, system design, and system architecture of the footwear e-commerce website as the foundation for testing activities; Developing a test plan to define testing objectives, scope, strategies, and testing resources; Designing and executing test cases for the core business functionalities of the system, including functional testing, user interface testing, and data testing; Applying testing methods and techniques in an Agile development environment and integrating testing activities into the CI/CD pipeline and finally summarizing testing results through documentation such as test reports, bug reports, and test summaries.

### **1.3. Agile and CI/CD process**

#### 1.3.1. Agile

During the development and testing of the online footwear retail website, the project team adopts the Agile methodology based on the Scrum framework to organize and manage work flexibly, accommodating frequent changes in business requirements. System requirements are decomposed into User Stories and stored in the Product Backlog, serving as the basis for planning and executing development iterations.

Based on priority, User Stories are selected and implemented in each Sprint, with each Sprint lasting from one to two weeks. Each Sprint aims to deliver a working software increment that is ready for testing, enabling continuous quality evaluation. Within each Sprint, the following activities are performed:

* Analyzing and clarifying business requirements of User Stories.
* Designing and developing corresponding functionalities across the presentation, business logic, and data layers.
* Designing and executing testing scenarios at multiple levels, including Unit Testing, Integration Testing, System Testing, and User Acceptance Testing (UAT).
* Deploying the new software version to a staging environment for testing and practical evaluation.

At the end of each Sprint, the team conducts a Sprint Review to assess the completion of planned objectives. The adoption of Agile/Scrum ensures alignment with initial testing goals while allowing flexible adjustments when new requirements or modifications arise.

#### 1.3.2. CI/CD

The project implements a CI/CD (Continuous Integration / Continuous Deployment) pipeline to automate the processes of integration, testing, and deployment. The project source code is centrally managed on GitHub, enabling version control and effective collaboration. Whenever changes are introduced through commits or pull requests on the main branches, GitHub Actions automatically triggers predefined workflows. The CI/CD pipeline is divided into two main phases:

Continuous Integration (CI):

* Automatically setting up the environment and installing dependencies for both backend and frontend components.
* Automatically executing Unit Tests and Integration Tests to detect early defects at the business logic and API levels.
* Verifying the build process of both frontend and backend to ensure the source code is deployment-ready.

Continuous Deployment (CD):

* After successful integration and testing, the pipeline automatically deploys the new system version to the staging environment.
* Environment variables are used to configure essential parameters such as database connections, security keys, and external service integrations.
* Each code update produces a stable deployment version, allowing continuous testing in an environment close to production.

The adoption of CI/CD shortens feedback cycles, improves testing efficiency, and ensures that all system changes are verified promptly before being released.

## **2. Requirement analysis**

### **2.1. Current System Assessment**

The shoe-selling website is an e-commerce system operating on a web platform, providing core functionalities such as product browsing and searching, shopping cart management, online order placement, and order history review. The system is built based on a client–server architecture, in which the server side is responsible for business logic processing and data management, while the client side handles user interface presentation and user interactions.

Through the system assessment, it can be observed that the website has successfully implemented its main business functionalities. However, there are still potential risks related to functional defects, data consistency, and system stability, especially when the system scales or handles a higher volume of users. Therefore, conducting software testing is necessary to evaluate system quality, detect defects at an early stage, and serve as a foundation for building testing plans in subsequent phases.

### **2.2. Business Context**

The Evershop Shoes system supports the fundamental business operations of an e-commerce platform, including product searching, shopping cart management, order placement and payment processing, and user access control.

**Product Management Process:**

When customers access the website, they can view products currently available for sale. The system provides various product discovery features, such as searching by product name using the search bar on the homepage or filtering products by gender at the top of the homepage. Additional filtering options include shoe size, color, and price range. When customers click on a product from the homepage or from filtered or search results, they are redirected to the product detail page. This page displays detailed product information, including product name, description, available colors, sizes, and selling price.

Administrators are responsible for managing product data, including adding new products when new inventory arrives, editing product information when updates are required, and hiding products when they are no longer available for sale.

**Purchase Process:**

After selecting the desired product, customers can add the product to the shopping cart directly from the homepage or the product detail page. The selected product information is then displayed in the shopping cart. Customers can review their cart by clicking on the cart icon, where they can increase the quantity of products or remove items they no longer wish to purchase. When the quantity is adjusted or items are removed, the shopping cart automatically updates the total price and product quantity.

To proceed with purchasing, customers click the “Place Order” button in the shopping cart and are redirected to the checkout page. On this page, customers can select a delivery address, review the products in the order, and choose the payment method, which is cash on delivery. If no changes are needed, customers click the “Confirm” button to place the order. If customers wish to modify their selections, they can return to the shopping cart, cancel the checkout process, and the selected products will remain in the cart.

**Order Management Process:**

After successfully placing an order, customers can review their orders on the order history page. Each order has a specific status, including: pending confirmation, confirmed, in delivery, delivered, and canceled. Customers are allowed to cancel orders as long as the administrator has not yet confirmed them.

For administrators, new orders are displayed on the order management page. Administrators can process orders by confirming them, assigning them to a shipping unit, marking them as delivered, or canceling them with a specified reason. If an order is canceled, a notification is sent to the customer. Order status updates are synchronized and displayed for both customers and administrators.

**Access Control Process:**

The system supports role-based access control with two roles: customer and administrator. Customers who successfully register and log in are redirected to the customer homepage, where they can view available products and place orders. Administrators who log in are redirected to the administrative dashboard, which allows them to manage products and orders. Both customers and administrators can log out of the system when they no longer wish to use the website.

#### 2.2.1. Actor Diagram of the System

Customer: The end user who interacts with the system to browse products and make purchases.

System Administrator: The highest-level administrator who has full access to all management functionalities of the system.

A white dots on a black background

Description automatically generated

Figure 1.2.2. 1. Actor Diagram of the System

#### 2.2.2. System Context Diagram

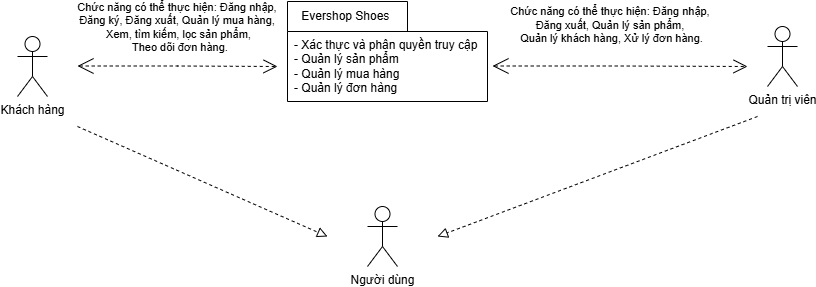


Figure 1.2.2. 2. System Context Diagram

This diagram illustrates the interaction flow between system actors and the Evershop Shoes system. Main Components:

* Customer: Performs functions such as login, registration, shopping management, product searching and filtering, and order tracking.
* Administrator: Performs login and logout operations, manages products, manages customers, and processes orders.
* Evershop Shoes System: Acts as the central component responsible for authentication, authorization, and core business logic management.

### **2.3. Technologies Used**

**Node.js:** An open-source server-side runtime environment built on Google Chrome’s V8 JavaScript engine. It provides an event-driven, non-blocking, asynchronous, and cross-platform execution environment for building highly scalable server-side applications using JavaScript.

**PostgreSQL 13:** A relational database management system used for data storage and query processing.

**TypeScript:** A superset of JavaScript that provides static typing and improved type safety, used for backend development.

**React:** An open-source JavaScript library for building component-based user interfaces. EverShop implements server-side rendering (SSR) of React components combined with hydration to deliver fast performance and improved SEO optimization.

**GraphQL:** A query language for APIs and a runtime environment for executing queries against existing data. EverShop utilizes GraphQL together with React to build a flexible and scalable user interface.

### **2.4. Business requirements**

#### 2.4.1. Functional requirements

|  |  |  |  |
| --- | --- | --- | --- |
| **Requirement ID** | **Requester** | **Requirement Description** | **Priority** |
| **BR1** | **Product Management** | | |
| BR1.1 | Customer | View all products currently available on the website. | High |
| BR1.2 | Customer | Search by product name and filter by price, size, and color with sorting options (ascending order or alphabetical order) and pagination. | High |
| BR1.3 | Customer | View detailed product information by clicking on a product. | High |
| BR1.4 | Administrator | Add new products. | High |
| BR1.5 | Administrator | Edit product information. | Medium |
| BR1.6 | Administrator | Hide products that are no longer available for sale. | Medium |
| **BR2** | **Purchase Process** | | |
| BR2.1 | Customer | Add products to the shopping cart. | High |
| BR2.2 | Customer | Remove products from the shopping cart. | High |
| BR2.3 | Customer | View the list of products in the shopping cart. | High |
| BR2.4 | Customer | Modify product quantities in the shopping cart. | High |
| BR2.5 | Customer | Select a delivery address. | High |
| BR2.6 | Customer | Select a payment method (cash on delivery). | High |
| BR2.7 | Customer | Return to the shopping cart if they decide not to proceed with the purchase. | Medium |
| **BR3** | **Order Management** | | |
| BR3.1 | Customer | View order history. | High |
| BR3.2 | Customer | View order details. | High |
| BR3.3 | Customer | Cancel an order. | High |
| BR3.4 | Administrator | View the list of orders. | Medium |
| BR3.5 | Administrator | Update order status: confirmed → in delivery → delivered. | High |
| BR3.6 | Administrator | Enter cancellation reasons and notify customers. | Medium |
| **BR4** | **Access Control** | | |
| BR4.1 | Customer | Register an account using a username and password. | High |
| BR4.2 | Customer | Log in using registered credentials. | High |
| BR4.3 | Administrator | Log in using assigned administrator credentials. | Medium |
| BR4.4 | Customer Administrator | Log out of the system. | Medium |

Table 1.2.4. 1. Functional Requirements Table

#### 2.4.2. Non-functional Requirements

|  |  |  |  |
| --- | --- | --- | --- |
| **Requirement ID** | **Requester** | **Requirement Description** | **Priority** |
| **BR1** | **Performance** | | |
| BR1.1 | Customer, Administrator | Page loading speed must be fast and smooth without noticeable lag. | High |
| BR1.2 | Khách hàng | Response time for actions (add to cart, checkout) must be less than 3 seconds, providing clear feedback to users and avoiding frustration. | High |
| **BR2** | **Security** | | |
| BR2.1 | Customer, Administrator | Login and registration processes must be secure, including password encryption. | High |
| BR2.2 | Customer, Administrator | Clear access control between Administrator and Customer roles. | High |
| **BR3** | **Usability** | | |
| BR3.1 | Khách hàng | The website interface must be user-friendly, intuitive, and easy to use. | High |
| BR3.2 | Quản trị viên | The admin interface must be logically organized to support efficient management. | High |

Table 1.2.4. 2. Non-functional Requirements Table

# CHAPTER 2: SYSTEM DESIGN ANALYSIS AND SYSTEM ARCHITECTURE

## **1. System Design Analysis**

### **1.1. Business Process Diagrams**

#### 1.1.1. Product Management Business Process Diagram

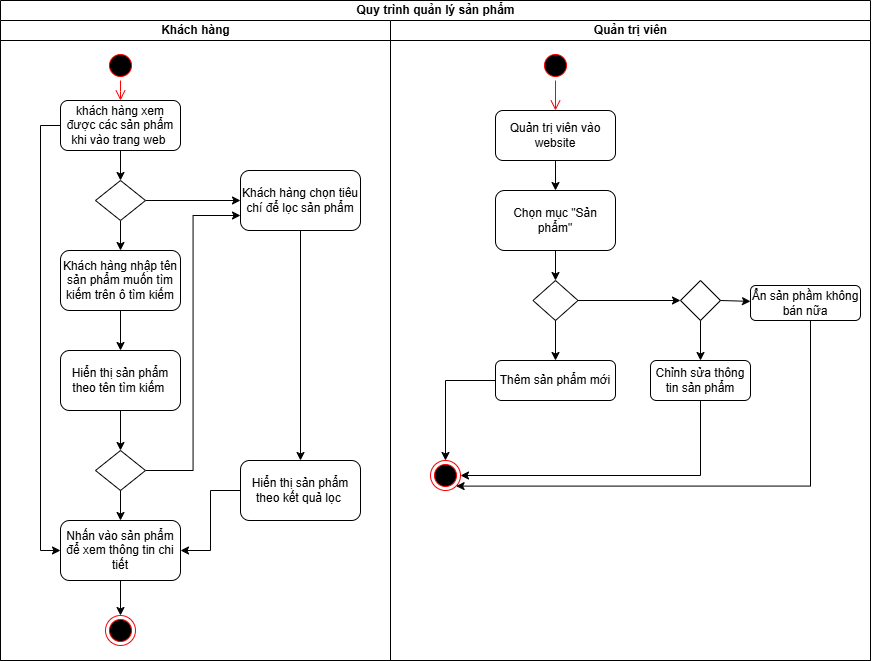


Figure 2.1.1. 1. Product Management Business Process Diagram

**Description:**

After logging into the website, customers can view products currently available for sale, access product details, search for products via the search bar, and apply filters to enhance the user experience.

After logging in, administrators can manage products through the “Products” section by adding new products when inventory is updated, editing existing product information, and hiding products when they are no longer sold.

#### 1.1.2. Purchase Business Process Diagram

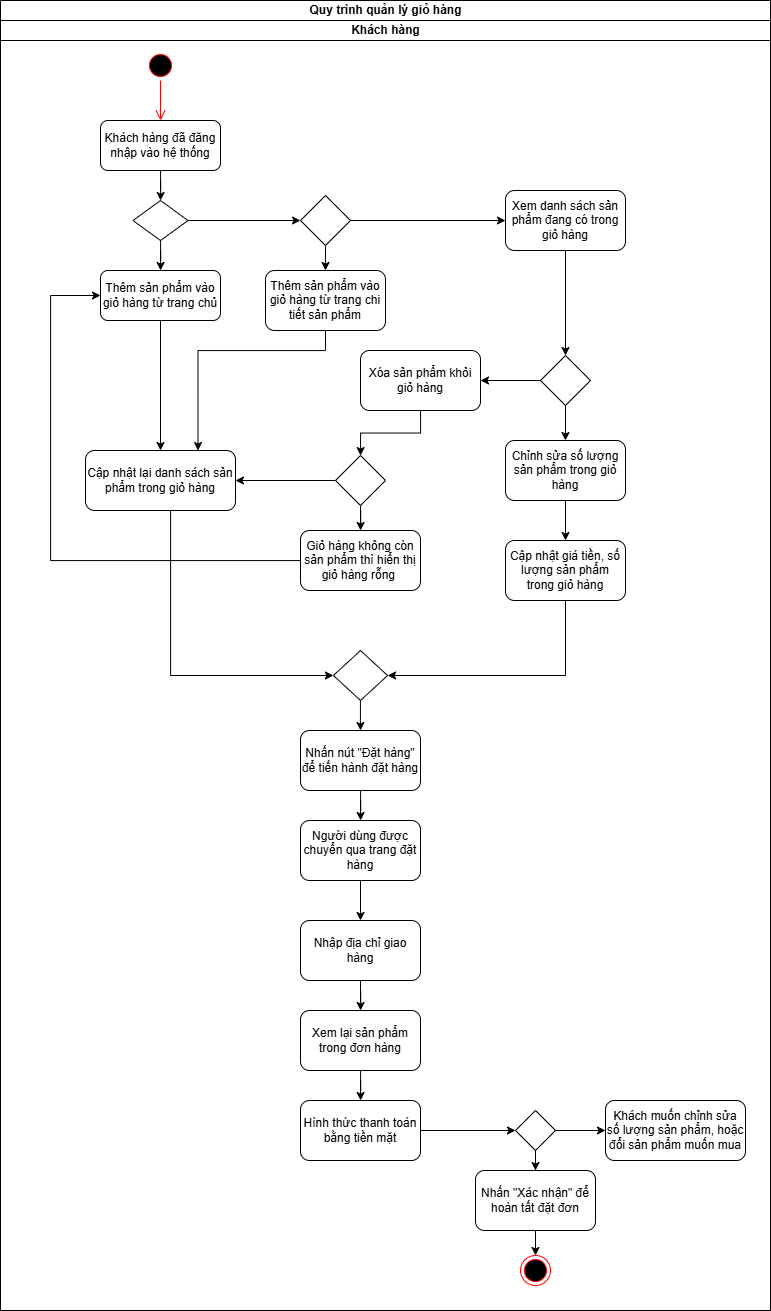


Figure 2.1.1. 2. Purchase Business Process Diagram

**Description:**

Customers who have logged into the system can add products to the shopping cart from the homepage or the product detail page. After adding products, the cart is updated to reflect the current list of items. Customers can review the shopping cart, modify product quantities, and the total price and quantity will be updated accordingly. Customers may also remove products from the cart; if the removed product is the only item, the cart becomes empty.

Afterward, customers click the “Place Order” button and are redirected to the checkout page. On this page, customers enter the delivery address, review the products included in the order, and select cash as the payment method before confirming the order. Upon confirmation, the order placement process is completed. If customers wish to modify quantities or change products, they can return to the shopping cart.

#### 1.1.3. Order Management Business Process Diagram

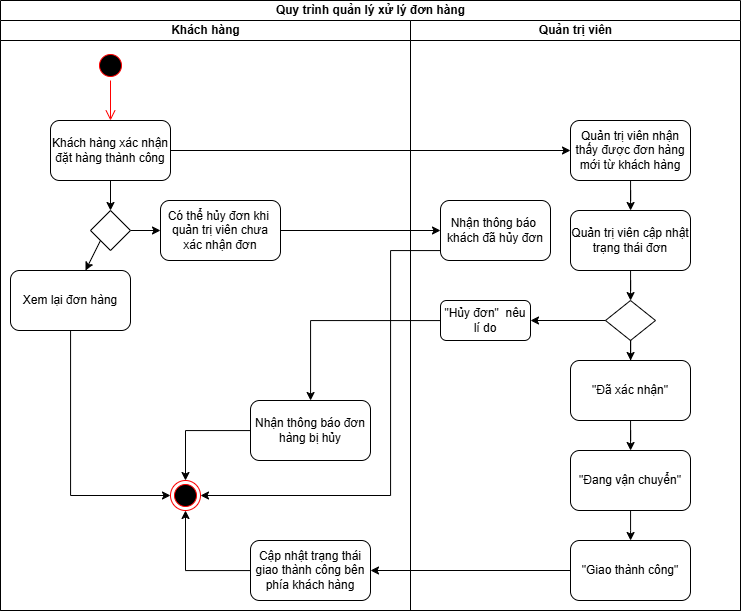


Figure 2.1.1. 3. Order Management Business Process Diagram

**Description:**

After customers successfully confirm an order, they can review the order details or decide to cancel the order. Order cancellation is only allowed before the administrator confirms the order. When a customer cancels an order successfully, the system sends a notification to the administrator.

Administrators can view new orders and update order statuses if the order has not been canceled. The order status is updated sequentially as confirmed, in delivery, and delivered. Once the order is delivered successfully, the status is updated and reflected on the customer side. Administrators can also cancel orders by providing a cancellation reason, which is then communicated to the customer.

#### 1.1.4. Access Control Management Business Process Diagram

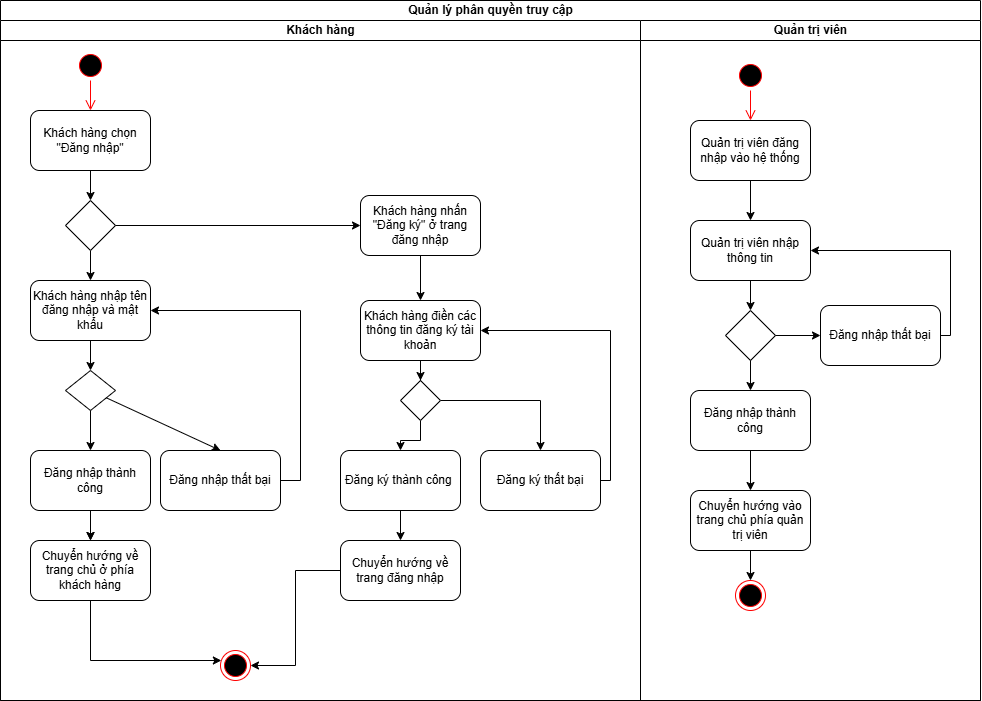


Figure 2.1.1. 4. Access Control Management Business Process Diagram

**Description**:

Customers who already have an account can log in to the website using their email and password. Upon successful login, customers are redirected to the customer homepage to begin shopping. If login fails, customers are prompted to re-enter their credentials.

Customers without an account can create a new account by clicking the “Register” button on the login page and filling in the required information. Upon successful registration, customers are redirected back to the login page. If registration fails, customers are required to re-enter their information.

Administrators log in using a dedicated email and password assigned to them. Upon successful login, administrators are redirected to the admin dashboard to perform management tasks. If login fails, administrators are prompted to try again.

### **1.2. Use Case Diagrams**

#### 1.2.1. Product Management Use Case

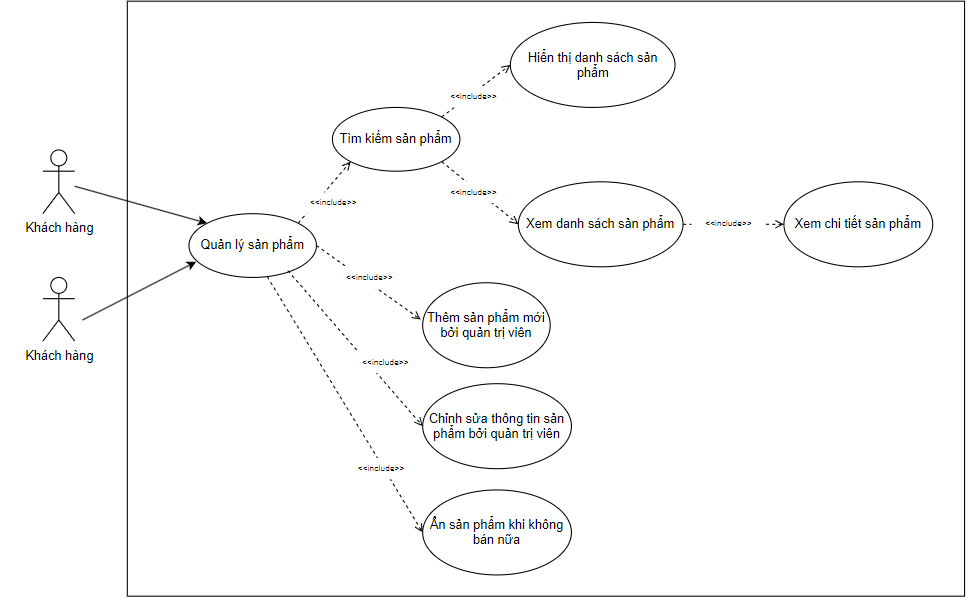


Figure 2.1.2. 1. Product Management Use Case

**Description:**  
Customers can view the product list, search for products, filter products by price, size, and color, sort products alphabetically, and view detailed product information. Administrators can manage products by adding new products when inventory is updated, editing product information when necessary, and hiding products that are no longer sold.

#### 1.2.2. Purchase Management Use Case



Figure 2.1.2. 2. Purchase Management Use Case

**Description:**  
Customers can view products in the shopping cart. When purchasing new products, customers click the “Add to Cart” button on the homepage or product detail page. Customers can remove products from the cart or modify product quantities. After clicking “Place Order,” order information is displayed, and customers confirm the delivery address, select a payment method, and click “Confirm” to complete the purchase.

#### 1.2.3. Order Management Use Case

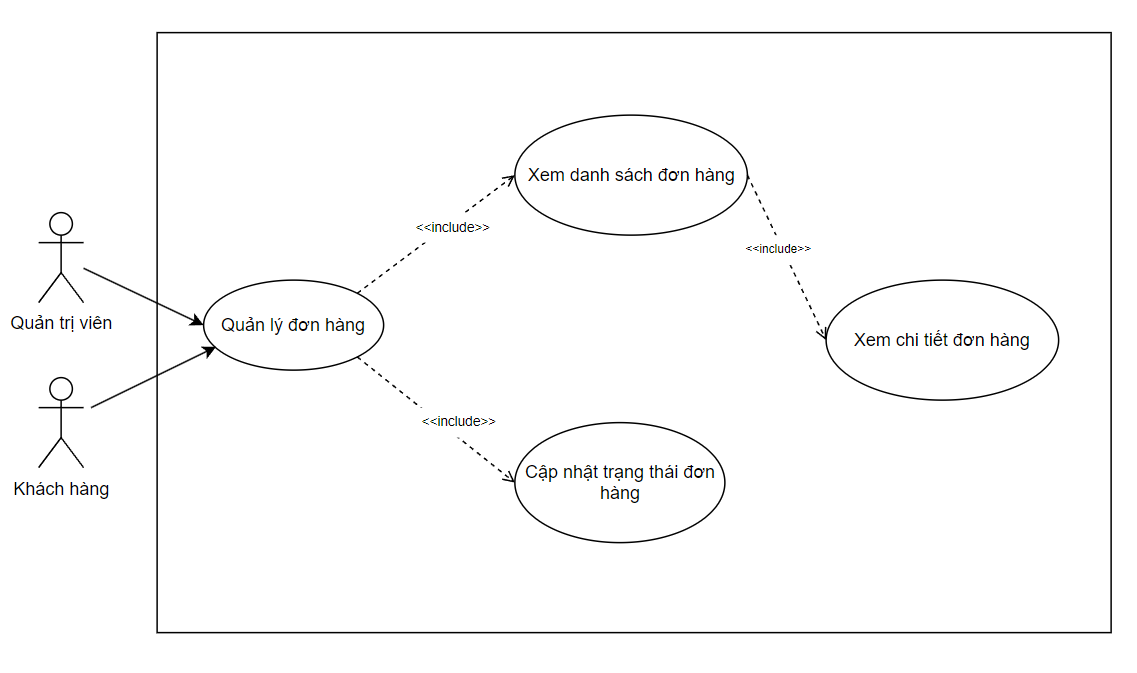


Figure 2.1.2. 3. Order Management Use Case

**Description:**  
Customers can view their order list and order details after placing orders. Administrators can update order statuses for delivery or cancel orders when necessary.

#### 1.2.4. Access Control Management Use Case

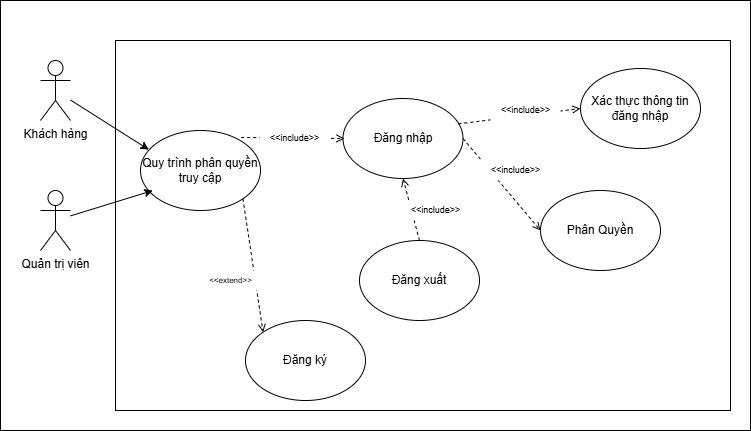


Figure 2.1.2. 4. Access Control Management Use Case

**Description:**  
Customers can register and log in to the system to perform purchasing activities and log out after use. Administrators can log in and log out of the system. During login, user authentication and role-based authorization are performed.

### **1.3. User stories**

**Product Management User Stories**

As a customer, I want to view all products available on the website so that I can choose products that suit my needs.

As a customer, I want to view product details when clicking on a product on the homepage to learn more information about it.

As a customer, I want to search for products by name so that I can quickly find the desired product.

As a customer, I want to filter products by price, size, and color to find the most suitable shoes.

As a customer, I want to combine searching by name with filtering options.

As an administrator, I want to add new products to the store to keep up with fashion trends.

As an administrator, I want to edit product information when necessary to ensure accuracy.

As an administrator, I want to hide products that are no longer sold.

**Purchase Management User Stories**

As a customer, I want to add products to the shopping cart from the homepage or product detail page.

As a customer, I want to review the list of products in the shopping cart before checkout.

As a customer, I want to change product quantities in the cart to meet my needs.

As a customer, I want to remove products from the cart if I no longer want to purchase them.

As a customer, I want to review order information before confirming the purchase.

As a customer, I want to change the delivery address to receive products at my desired location.

As a customer, I want to select a payment method.

**Order Management User Stories**

As a customer, I want to view the list of orders I have placed.

As a customer, I want to view order details to review information when needed.

As a customer, I want to cancel orders to modify quantities or cancel purchases.

As a customer, I want to track order status to know when my order will arrive.

As an administrator, I want to view the order list to update order processing.

As an administrator, I want to confirm orders to proceed with selling products.

As an administrator, I want to cancel orders when products are defective.

**Access Control User Stories**

As a customer accessing the website, I want to register a new account to log in and use system features.

As an administrator, I want to log in to manage orders and products and log out after use.

As a system administrator, I want to lock or unlock user accounts to control access for violating or inactive accounts.

### **1.4. Data**

#### 1.4.1. Conceptual ERD

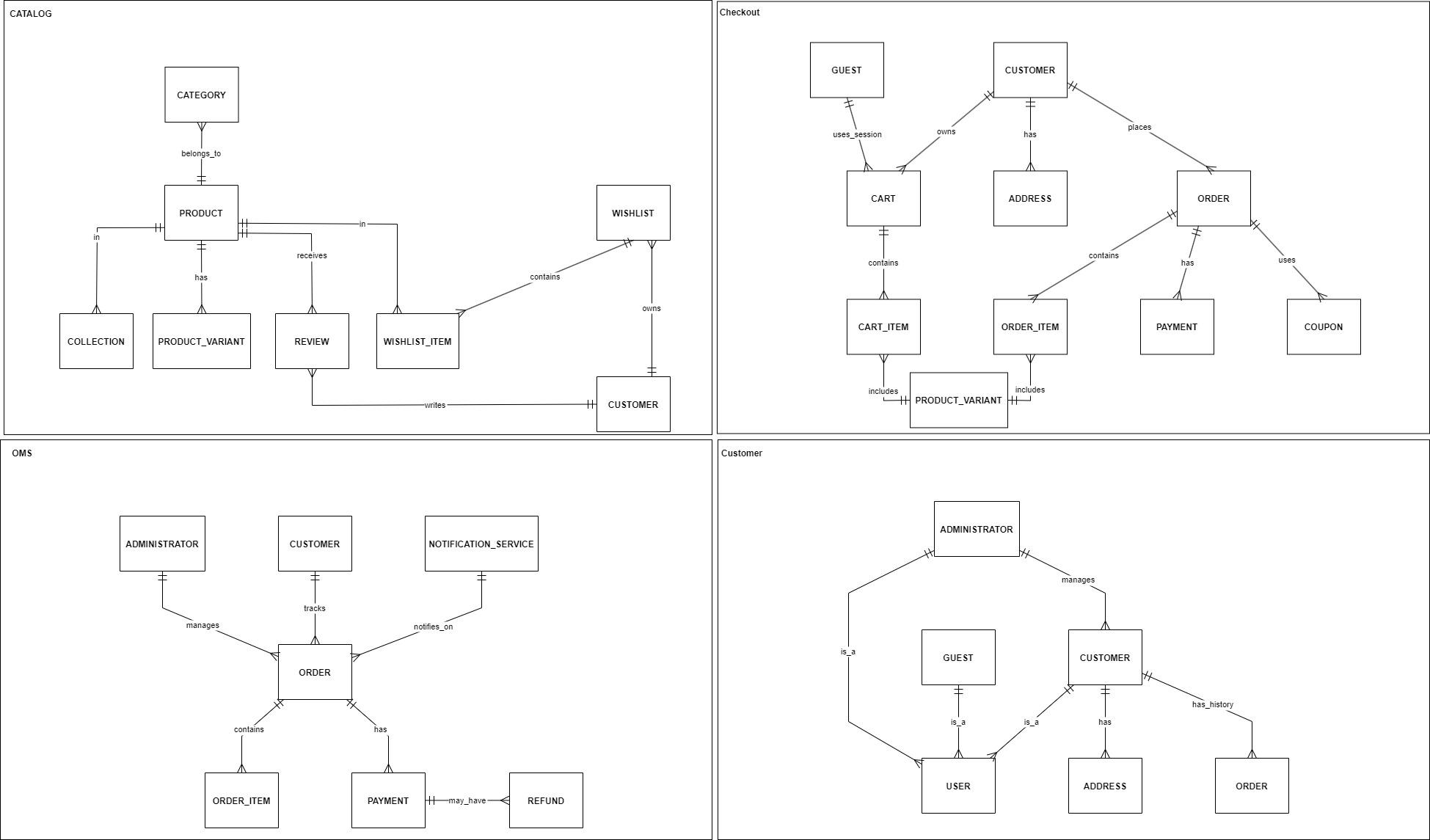


Figure 2.1.3. 1. Conceptual ERD

**Description:**

The Conceptual ERD illustrates the main entities of the “Online Shoe Store” system, including Catalog (Category belongs to Product, Product has Collection, Variant, and Review), and Wishlist (Wishlist contains Wishlist\_Item and is owned by Customer).

The Checkout module includes Guest/Customer using or owning Cart, which contains Cart\_Item (including Product Variant), placing Orders that contain Order\_Item, have Payment, and may use Coupons.

The Order Management System (OMS) shows Administrator managing Orders, Customers tracking Orders, Notifications, Orders containing Order\_Items, Payments, and optional Refunds.

The Customer domain includes Administrator management, Guest/User as Customer, Address ownership, and Order history. Overall, the ERD focuses on abstract relationships among products, customers, orders, and management without specifying detailed attributes.

#### 1.4.2. Logical ERD

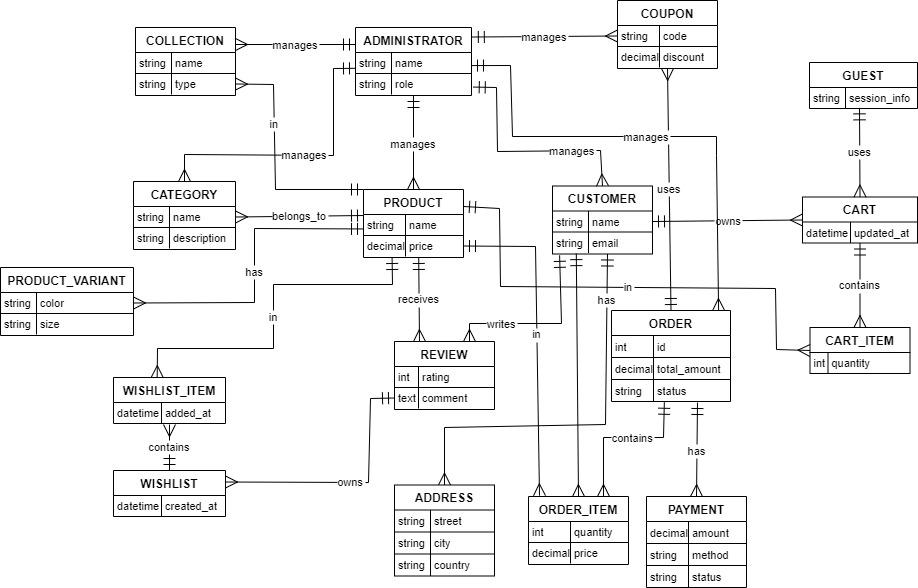


Figure 2.1.3. 2. Logical ERD

**Description:**

The Logical ERD describes the database schema of the “Online Shoe Store” with detailed entities and attributes, including Administrator (name, role), Collection (name, type), Category (name, description), Product (name, price, Product Variant with color and size), Customer (name, email), Coupon (code, discount), and Order (id, total amount, status).

Customers own Wishlists (created\_at, Wishlist\_Item added\_at), Addresses (street, city, country), receive Reviews (rating, comment), and use Carts (updated\_at, Cart\_Item quantity). Orders contain Order\_Items, Payments, and may use Coupons. Guests use carts similarly to customers. The ERD emphasizes relationships and data types to support database implementation.

#### 1.4.3. Physical ERD

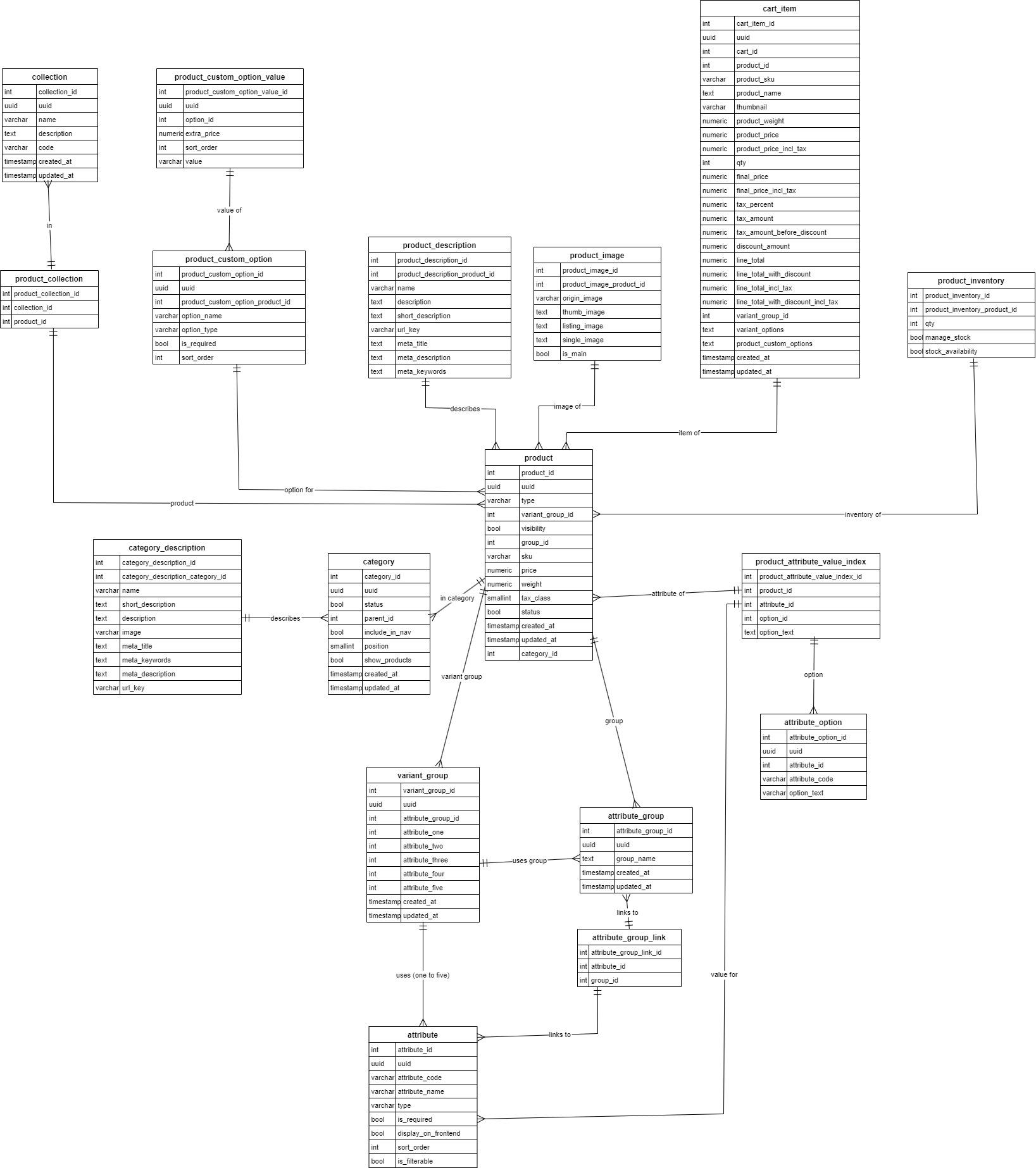


Figure 2.1.3. 3. Physical Product-Catalog

**Description:**

The Physical ERD of the Catalog module describes the database schema with the main tables, including Collection (int id, varchar name, varchar type, timestamp created\_at/updated\_at), which is linked to Product\_Collection (int product\_id, int collection\_id) to manage the association between products and collections.

The Product table (int id, varchar name, int description\_id, numeric price, int stock, int category\_id, timestamp created\_at/updated\_at) is connected to Product\_Description (int id, varchar short\_description, text meta\_description, varchar meta\_keywords, etc.), Product\_Image (int id, varchar image, boolean main\_image), and receives Review entries (int id, int rating, text comment).

The Category table (int id, varchar name, int description\_id, boolean active, timestamp created\_at/updated\_at) is associated with products and has a corresponding Category\_Description table with a similar structure.

To support product variants, the schema includes Variant\_Group (int id, varchar name, int attribute\_group\_id), Variant (int id, int variant\_group\_id, int attribute\_id, varchar value, timestamp created\_at/updated\_at), Option (int id, int attribute\_id, varchar value or index), and Attribute (int id, varchar name, varchar type, boolean required, timestamp created\_at/updated\_at). These entities establish one-to-many (1:N) and many-to-many (N:N) relationships to enable flexible product variant management.

Overall, the ERD emphasizes the use of primary keys (int id), foreign keys, explicit data types (varchar, numeric, timestamp), and indexing strategies to support the implementation of a PostgreSQL database for the product catalog management module.

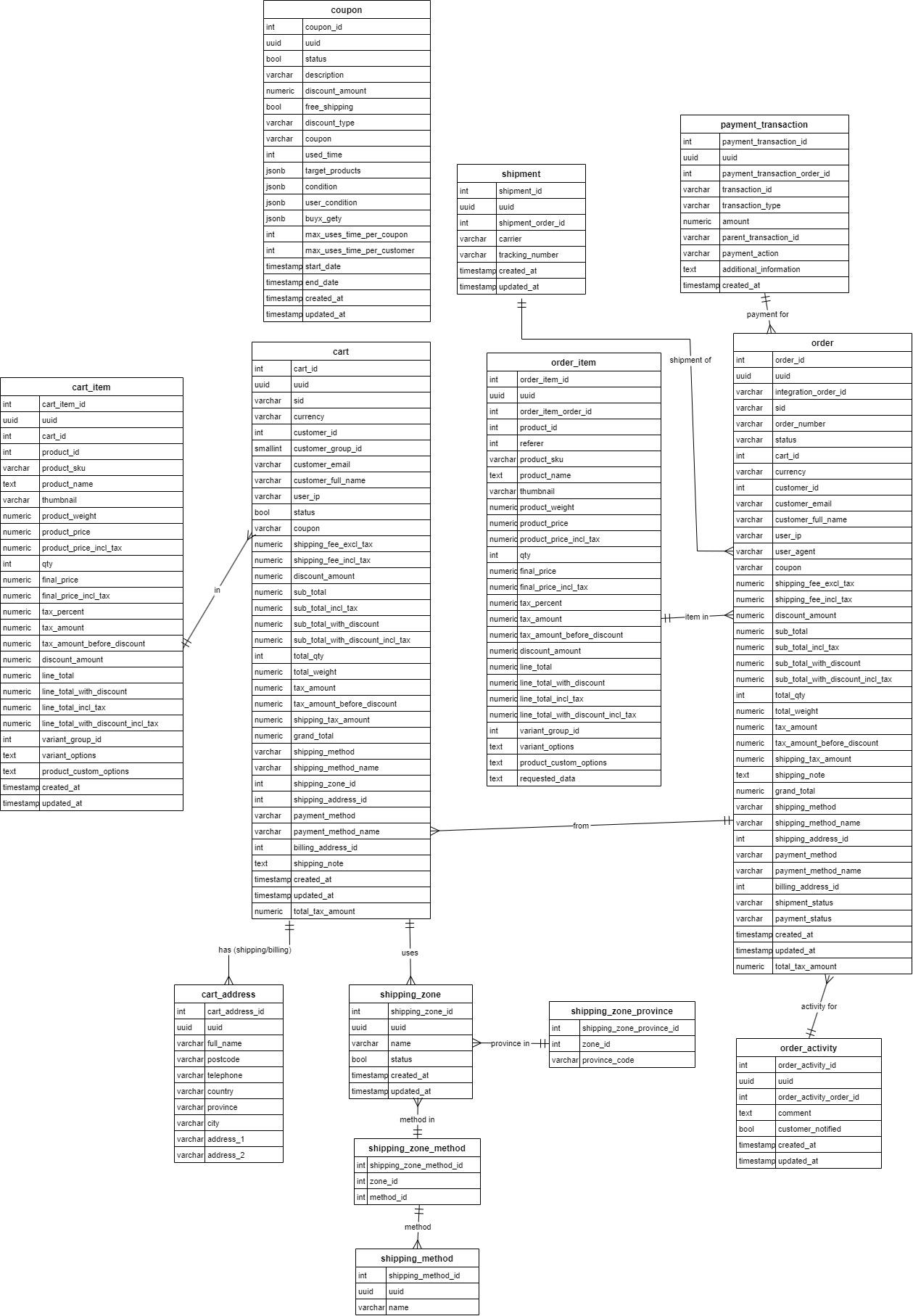


Figure 2.1.3. 4. Physical Cart - Order

**Description:**

The logical ERD for the checkout and Order Management System (OMS) module describes the core entities involved in e-commerce order processing. At the center is the Order entity, which is linked to Order\_Item to represent product details, quantities, and prices, and connected to Shipment to manage delivery information such as shipping methods and regions.

The Payment\_Transaction entity is associated with Order and Payment to record transaction details, including transaction type, amount, and status. The Cart entity serves as a precursor to Order, linking to Cart\_Item for items prior to checkout and to Coupon for applied discount codes.

Supporting entities include Shipping\_Zone, Shipping\_Method, and Activity, which are used to log events and manage shipping configurations. Relationships between entities are enforced through foreign keys such as order\_id and shipment\_id, ensuring data consistency and referential integrity across the system.

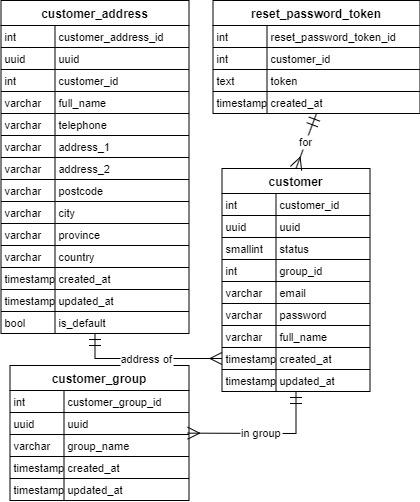


Figure 2.1.3. 5. Physical Customer

**Description:**

This diagram illustrates the physical database schema with four main tables: Customer\_Address, Reset\_Password\_Token, Customer, and Customer\_Group.

The Customer\_Address table contains fields such as int customer\_address\_id (primary key), int customer\_id (foreign key), varchar full\_name, telephone, address\_1, address\_2, postcode, city, province, country, timestamp created\_at, timestamp updated\_at, and boolean is\_default. It is linked to the Customer table through the relationship “address of Customer.”

The Reset\_Password\_Token table includes int reset\_password\_token\_id (primary key), int customer\_id (foreign key), text token, and timestamp created\_at, and is directly associated with the Customer table.

The Customer table serves as the central entity, consisting of int customer\_id (primary key), uuid uuid, smallint status, int group\_id (foreign key), varchar email, password, full\_name, timestamp created\_at, and timestamp updated\_at.

The Customer\_Group table contains int customer\_group\_id (primary key), varchar group\_name, and timestamp created\_at and updated\_at. The “in group” relationship connects Customer to Customer\_Group, indicating customer group membership.

### **1.5. Data Dictionary**

Due to the large number of database tables, the data dictionary is documented in a separate .docx file.

## **2. System Architecture**

### **2.1. Overall System Architecture Diagram**

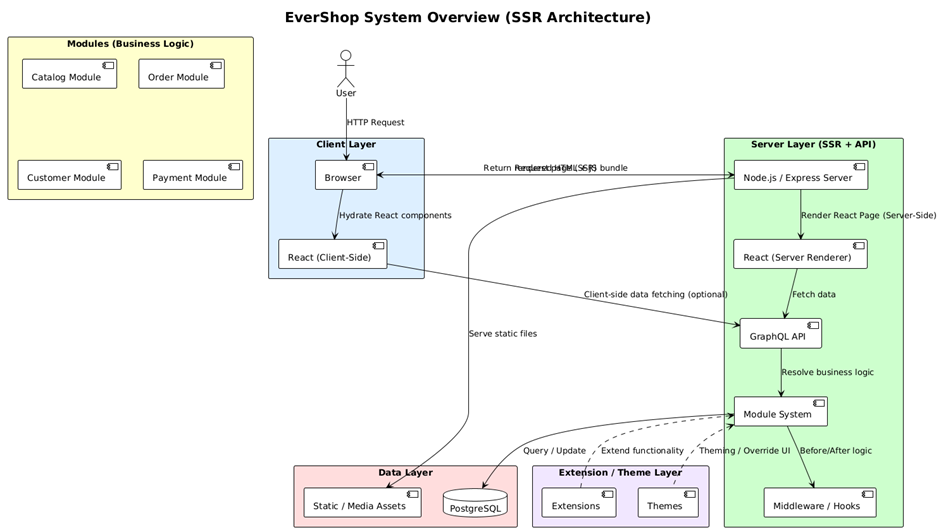


Figure 2.2.1. 1. Overall System Architecture Diagram

**Description:**

The overall system architecture diagram of EverShop (based on an SSR architecture) consists of several core layers. The Modules layer (Business Logic) includes key domains such as Catalog, Order, Customer, and Payment. The Client Layer represents the User interacting through a web browser, which sends HTTP requests to the React application on the client side for component hydration.

The Server Layer (SSR + API) utilizes a Node.js/Express server to perform server-side rendering of React pages, fetch data through GraphQL APIs, resolve business logic via the module system, and return bundled HTML, JavaScript, and CSS assets to the client.

The Data Layer comprises static and media assets as well as a PostgreSQL database for persistent data storage.

The Extension/Theme Layer supports system extensibility through extensions, themes, and middleware/hooks, enabling functional customization and flexible system enhancement.

The overall data flow begins with optional client-side data fetching, followed by interactions with GraphQL APIs, ensuring synchronization between server-side rendering and client-side rendering for a consistent and optimized user experience.

### **2.2. C4 Architecture**

#### 2.2.1. Level 1: Context Diagram



Figure 2.2.2. 1. Level 1: Context Diagram

**Description:**

The Context Diagram illustrates the “Online Shoe Store” system as the central component, interacting with two external entities: Customer and Administrator (Admin).

Customers perform activities such as searching for products, purchasing items, viewing the shopping cart, making payments, and checking order history, with the system supporting the complete online shopping workflow.

Administrators are responsible for managing inventory, processing orders, managing customer accounts, and applying discounts to maintain internal operations.

The bidirectional data flow helps clearly define the system boundaries and highlights the primary interactions between the system and its external actors.

#### 2.2.2. Level 2: Container Diagram

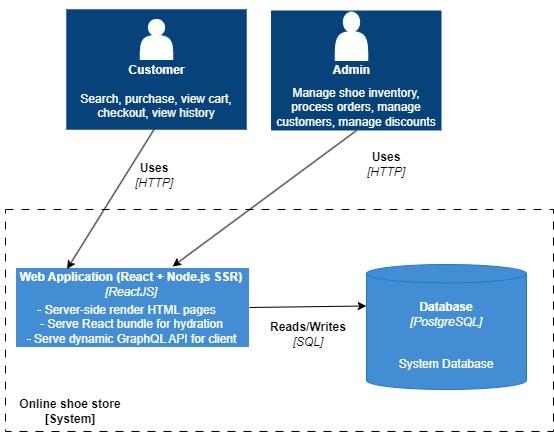


Figure 2.2.2. 2. Level 2: Container Diagram

**Description:**

The diagram illustrates the architecture of the “Online Shoe Store” system with the Web Application as the central component. The system adopts React + Node.js with Server-Side Rendering (SSR) instead of a Single Page Application (SPA) approach, allowing HTML content to be rendered on the server to optimize SEO performance and initial page load speed.

Customers and Administrators (Admins) interact with the system via HTTP. Customers perform actions such as searching for products, making purchases, viewing the shopping cart, completing payments, and checking order history, while administrators manage inventory, orders, customers, and discount policies.

The Web Application performs server-side rendering of HTML pages, serves the React bundle for client-side hydration, and provides dynamic GraphQL APIs to handle data processing and business logic.

The system connects to the PostgreSQL database through SQL read and write operations to store and retrieve data.

By using SSR instead of SPA, the system reduces client-side processing, improves user experience by delivering immediately available content, and is particularly well suited for e-commerce applications. Overall, the diagram emphasizes the bidirectional data flow between users, the web application, and the database, clearly defining system boundaries.

#### 2.2.3. Level 3: Component Diagram

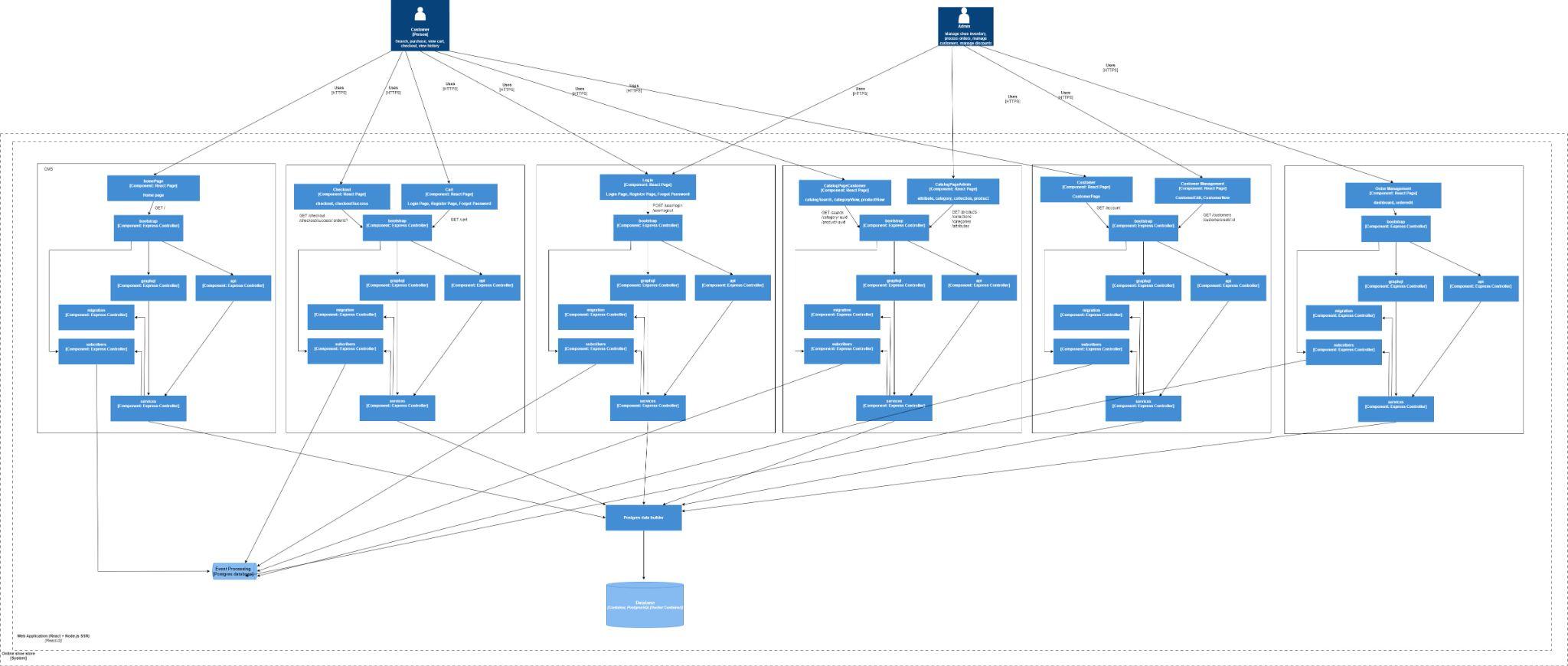


Figure 2.2.2. 3. Level 3: Component Diagram

**Description:**

The Component Diagram illustrates the structure of the “Online Shoe Store” system with two primary user roles: Customer (searching for products, making purchases, viewing the shopping cart, completing payments, and checking order history) and Administrator (Admin) (managing inventory, orders, customers, and discounts).

The presentation layer consists of various UI components such as the Home Page, Cart Component, Login/Register Page, Checkout Page, Category/Collection Page, Customer Management Page, Order Management Page, and others. These components interact with the backend through HTTP methods such as GET and POST.

Business logic is handled by backend services including the Continuation Service, Checkout Service, Authentication Service, Category Service, Customer Service, and Order Service.

The system connects to the Model Layer through APIs, utilizing Express with a PostgreSQL database for data storage and management.

Overall, the diagram emphasizes the layered architecture of the system, clearly separating the frontend (React components), backend services (Node.js/Express), and the database, with bidirectional data flow between these layers.

#### 2.2.4. Level 4: Code-level Diagram

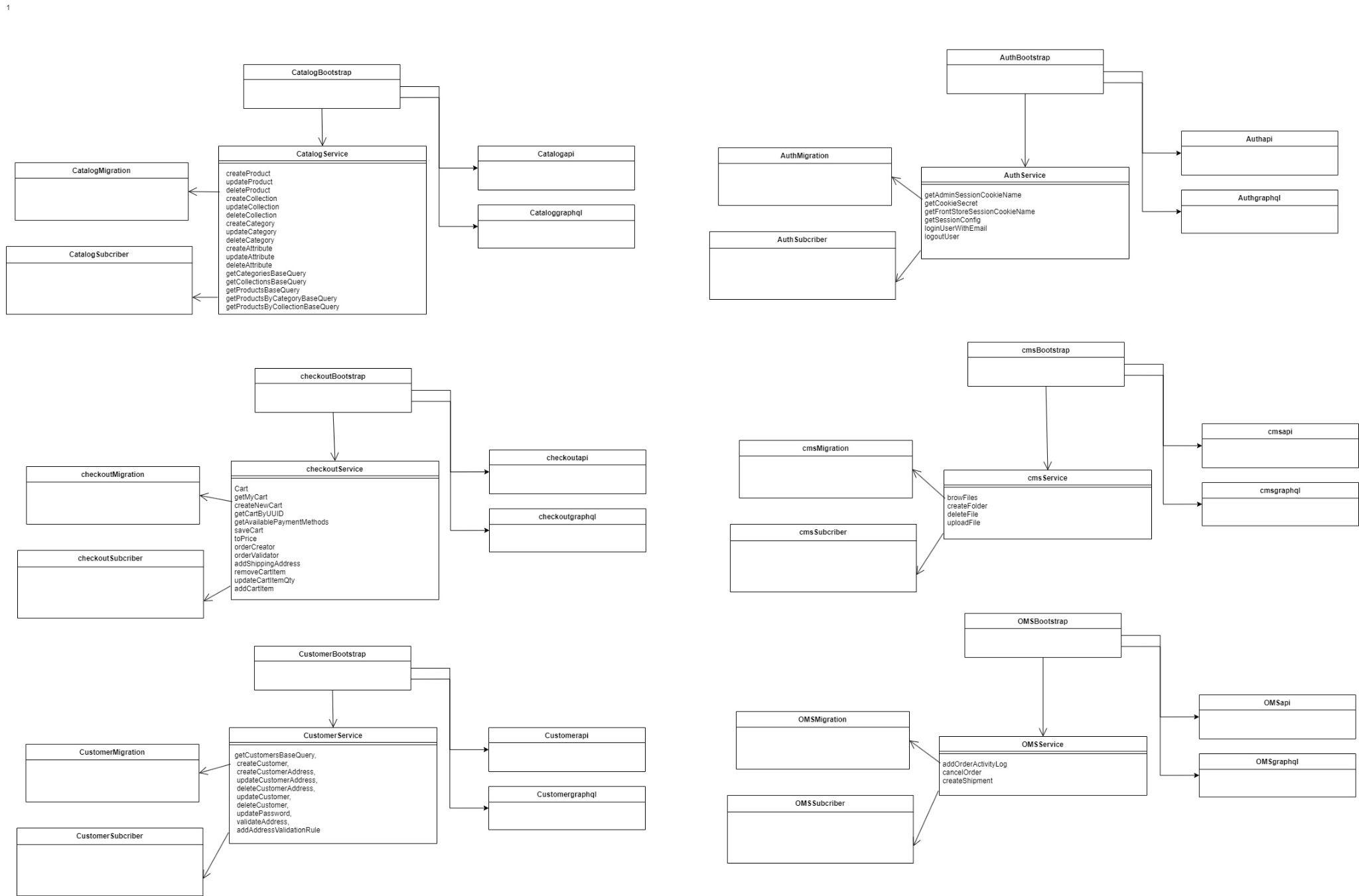


Figure 2.2.2. 4. Code-level Diagram

**Description:**

The diagram describes the source code structure with six main bootstrap modules: CatalogBootstrap, AuthBootstrap, CheckoutBootstrap, CmsBootstrap, CustomerBootstrap, and OMSBootstrap.

Each bootstrap module is connected to Migrations (data schema migrations), Subscribers (event registrations), and Services (service classes containing specific methods such as createProduct and loginUserByEmail). These components then branch into API and GraphQL layers to expose functionality to the application.

The arrows represent hierarchical relationships, helping organize the codebase into clearly defined modules, including product catalog management, authentication, checkout and payment processing, content management, customer management, and order management.

### **2.3. Communication View**

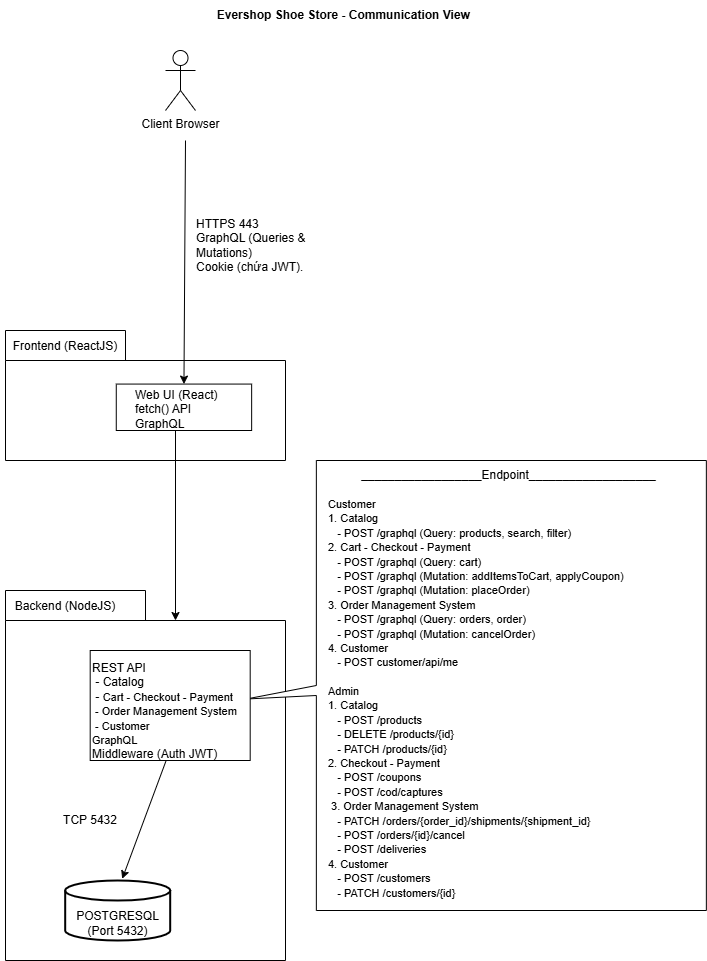


Figure 2.2.3. 1. Communication View

**Description:**

The Communication View diagram of the Evershop Shoe Store system describes a closed data flow that begins at the user side (Client Browser), where requests are transmitted over the secure HTTPS protocol (port 443), accompanied by cookies containing JWT tokens for authentication and authorization. The Frontend layer, built with ReactJS, is responsible for rendering the user interface and interacts with the Backend using the fetch() API and GraphQL for data exchange. At the Backend layer, developed on the Node.js platform, the system processes core business logic such as catalog management, shopping cart handling, payment processing, and order management through authenticated middleware components. A key architectural characteristic is the parallel use of two communication standards: while regular users (Customers) primarily interact via GraphQL to optimize data querying and minimize over-fetching, Administrators (Admins) use a standard RESTful API to manage system resources and perform administrative operations. Finally, all data is persistently stored in the PostgreSQL database, which is directly connected to the Backend via the TCP protocol on port 5432, ensuring reliable and efficient data storage and retrieval.

### **2.4. Deployment View**

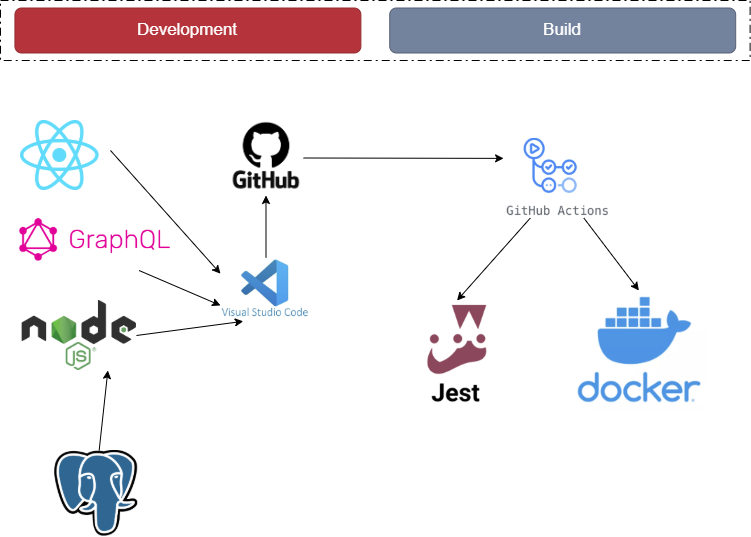


Figure 2.2.4. 1. Deployment View

**Description:**

The figure illustrates the system deployment and development process following a CI/CD model, which consists of two main stages: Development and Build.

During the Development stage, the system is developed in a local environment using Visual Studio Code. React is used for the user interface layer, Node.js for the server-side, and GraphQL for building APIs that support data exchange. System data is stored and managed in a PostgreSQL database. After completing development and code modifications, developers push the source code to the GitHub repository.

In the Build stage, whenever changes are detected in the GitHub repository, GitHub Actions is automatically triggered to execute the continuous integration pipeline. At this stage, the system runs automated test scripts using Jest to ensure the correctness and stability of the source code. Once the tests pass successfully, the application is packaged into Docker containers, enabling easier deployment, scalability, and consistency across different environments.

# CHAPTER 3: TEST PLAN

## **1. Introduction**

### **1.1. Objectives**

The objective of this document is to ensure that the Evershop Shoes Store system fully satisfies the specified business and technical requirements, to detect defects early during the development process, and thereby improve the overall software quality and stability before deployment.

The test plan serves as a unified foundation for all testing activities, supporting schedule control, test result evaluation, and the system acceptance process..

### **1.2. Scope**

This test plan applies to the entire testing process of the Evershop Shoes Store website throughout the software development life cycle (SDLC), including the phases of development testing, integration testing, system testing, and user acceptance testing (UAT).

## **2. Test Items**

### **2.1. Functional Testing**

|  |  |
| --- | --- |
| **Tested Function** | **Acceptance Criteria** |
| **Access Control & Authorization** | Verify that the system allows login for both **Customer** and **Administrator** roles.  Verify that the system allows **account registration for customers**.  Verify that the system allows **logout** for all authenticated users. Verify that the system correctly enforces **role-based access control**, allowing users to access only the functions appropriate to their roles.  Verify that the system allows administrators to **manage user accounts**. |
| **Product Management** | Verify that the system provides full **CRUD operations** for product management for administrators.  Verify that customers can access and view the **product list**. Verify that the system supports **searching, filtering, and sorting** products to help customers easily find desired items. Verify that product data consistency and accuracy are maintained between the **user interface** and the **backend system**. |
| **Shopping Management** | Verify that the system provides **CRUD operations** for the customer shopping cart.  Verify that the system automatically updates and accurately displays cart information, including product list, quantity, and total price.  Verify data consistency of the cart throughout user interactions and page navigation.  Verify that business rules are enforced to limit purchase quantities based on available inventory.  Verify that customers can proceed from the cart to the **checkout process**.  Verify that customers are allowed to **edit shipping addresses**. |
| **Content Management** | Verify that administrators can perform **CRUD operations** (Create, Read, Update, Delete) on static content pages (e.g., Home, About Us, Privacy Policy).  Verify that administrators can manage UI elements such as **sliders, banners, and content blocks**.  Verify that content updates made in the **Admin Dashboard** are immediately and accurately reflected on the **Storefront UI**. Verify that the system supports a **WYSIWYG editor** for formatting text and embedding images/videos. |
| **Order Management** | Verify that the system automatically generates an order with a **unique identifier** after successful checkout.  Verify that administrators can view a detailed list of all orders, including customer information, payment status, and shipping status.  Verify that administrators can update order statuses (e.g., Processing, Shipping, Completed, Cancelled) and notify customers via email.  Verify that product inventory is automatically deducted after a successful order.  Verify that administrators can generate **Invoices** and **Shipment documents** for each order.  Verify that customers can track order status and view order history in their personal account pages. |

Table 3.3. 1. Functional Testing Table

All features to be tested will be detailed in Test Scenario documents and corresponding Test Case documents, based on the testing types defined in the Scope section.

Test Scenario documents will be delivered for review during the Pre-Build phase, while Test Case documents will be delivered during the mid-Build phase, immediately before Integration Testing begins.

### **2.2. Non-Functional Testing**

|  |  |
| --- | --- |
| **Test Type** | **Acceptance Criteria** |
| **Usability Testing** | Verify that the system provides clear navigation, allowing users to access key pages such as Home, Product List, Cart, and Orders.  Verify that users can easily return to previous pages or the homepage without encountering dead links.  Verify that primary action buttons are clearly visible and easy to use. |
| **Compatibility Testing** | Verify that the website displays correctly and consistently across popular browsers such as **Chrome** and **Firefox.**  Verify that the system adapts responsively to different screen sizes.  Verify that UI components do not break or misalign when screen size changes.  Verify that users can fully operate system functions across different devices. |
| **UI Testing** | Verify that colors, fonts, and layouts conform to the design specifications.  Verify that logos, images, and icons are displayed correctly without distortion or misalignment.  Verify that error messages, warnings, and status notifications are displayed correctly according to business context. |
| **Security Testing** | Verify that user passwords are encrypted.  Verify that access control is enforced correctly and unauthorized access is prevented.  Verify that HTTP/HTTPS requests are secured and sensitive data is not exposed via URLs.  Verify that user sessions expire automatically after a defined timeout or upon logout. |
| **API Testing** | Verify that all API endpoints function correctly according to design specifications.  Verify that API responses are accurate, complete, and conform to required formats.  Verify that API access control and security are properly enforced.  Verify that APIs correctly interact with the database, UI, and related endpoints.  Verify that core business workflows remain stable after system changes. |

Table 3.3. 2. Non-Functional Testing Table

### **2.3. Out of Scope**

* Advanced security testing such as penetration testing, vulnerability scanning, and network infrastructure testing. The team only performs basic security testing focusing on authentication, authorization, and input validation.
* Integration with third-party payment gateways.
* Disaster recovery and failover testing.

## **3. Acceptance Criteria**

### **3.1. Test Coverage**

At least 60% of functional requirements must be covered by designed and executed test cases.

### **3.2.** **Test Case Execution**

All test cases at the following levels must be executed:

* Unit Testing
* Integration Testing

### **3.3. Defect Criteria**

No Critical defects are allowed at the time of acceptance.

The number of Major defects must not exceed two, and mitigation solutions must be agreed upon.

Minor defects must not impact core system functionality.

### **3.4. Code Coverage**

Code coverage metrics must meet the following thresholds:

* Statement Coverage ≥ 90%
* Branch Coverage = 100%
* Path Coverage = 100%

## **4. Test Strategy**

### **4.1. Testing Methodology**

The project adopts the V-Model for software testing, where testing activities are conducted in parallel with development phases. Each development phase corresponds to a specific testing level, ensuring software quality throughout the SDLC.The project includes Unit Testing, Integration Testing, and API Testing, with test cases designed based on functional requirements and system use cases.Additionally, automated testing is implemented through a CI/CD pipeline using GitHub Actions, automating the Build, Test, and Deploy processes.

### **4.2. Testing Phases**

|  |  |  |  |
| --- | --- | --- | --- |
| **Phase** | **Objective & Sprint Mapping** | **Responsible Team** | **Frequency** |
| **Unit Test (UT)** | Validate business logic at function/method level (Services, Controllers, Components). Ensure Sprint Backlog tasks are implemented correctly. | Development Team | Automatically triggered on code commit and merge to main branch |
| **Integration Test (IT)** | Ensure correct interaction between modules (React ↔ API Spring Boot) and database (Spring Boot ↔ MySQL). | Development / Testing Team | Automatically executed after Unit Tests and before Build |
| **System Test (ST)** | End-to-end testing of the entire system for selected User Stories in the Sprint. | Testing Team | Automatically executed after deployment to Staging environment |
| **Regression Test (RT)** | Re-test completed features from previous Sprints (e.g., Auth, Catalog) after new Sprint integration. | Testing Team | All automated tests from previous Sprints are re-executed |
| **Acceptance Test (UAT)** | Validate that the Product Increment meets the **Definition of Done (DoD)** and business needs. | User Representatives | End of each Sprint on Staging environment |

Table 3.4.2. 1. Testing Phases Table

### **4.3. Test Types**

|  |  |  |
| --- | --- | --- |
| **Test Type** | **Main Objective** | **Techniques Applied** |
| **Functional Testing** | Verify core modules: AUTH, CATALOG, CUSTOMER, CHECKOUT, OMS, CMS | Equivalence Partitioning and Boundary Value Analysis |
| **Business Cycle Testing** | Ensure complete e-commerce workflow from search to delivery | End-to-End testing using **Cypress** |
| **Data Integrity Testing** | Ensure inventory, pricing, and order status consistency in PostgreSQL | Database constraints and transaction validation |
| **Performance Testing** | Measure GraphQL API and SSR website performance | Load testing with **JMeter / k6** |
| **Security & Authorization Testing** | Enforce strict role separation between Customer and Admin | Unauthorized access attempts and JWT validation |

Table 3.4.3. 1. Test Types Table

## **5. Environment and Resources**

### **5.1. Test Environment**

* Hardware

|  |  |  |
| --- | --- | --- |
| **Item** | **Component** | **Purpose** |
| Development | Personal Computer | Development and Unit/Integration Testing |
| Devices | Desktop, Android, iOS | UI and compatibility testing |

Table 3.5.1. 1. Hardware Table

* Software

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Version** | **Notes** |
| OS | Windows 11 | 64-bit | Host environment |
| Runtime | Node.js | 16.x / 18.x (LTS) | Required by Evershop |
| Package Manager | npm | ≥ 8.x | Dependency management |
| Frontend | ReactJS | 17.x / 18.x | Main UI framework |
| Styling | Tailwind CSS | Latest | CSS framework |
| Language | TypeScript | 5.x | Primary language |
| API | GraphQL | Latest | Client–Server communication |
| Database | PostgreSQL | 13.x / 14.19 | Main database |
| Container | Docker Desktop | 4.x | Isolated environment |
| IDE | VS Code | Latest | Recommended editor |

Table 3.5.1. 2. Software Table

* Infrastructure

|  |  |  |
| --- | --- | --- |
| **Item** | **Component** | **Purpose** |
| Production | Vercel | Production deployment |
| CI/CD | GitHub Actions | Build, Test, Deploy automation |

Table 3.5.1. 3. Infrastructure Table

### **5.2. Human Resources**

This table display

Bảng này hiển thị số lượng nhân viên cần thiết cho dự án:

|  |  |  |
| --- | --- | --- |
| **Member** | **Role** | **Responsibilities** |
| **Chau Gia Anh** | Dev/Tester | Test planning  System testing  Test documentation  Result analysis  CI/CD main |
| **Duong Le Khanh** | Dev/Tester | Advanced test case design  Review checklist  Integration testing  Defect tracking |
| **Dao Thi Thanh Tam** | Dev/Tester | Test planning  Core test case design  Unit testing  CI/CD support |

Table 3.5.2. 1. Human Resources

### **­5.3. Milestones**

|  |  |  |  |
| --- | --- | --- | --- |
| **Task** | **Responsible** | **Start Date** | **End Date** |
| Test Plan Preparation | Chau Gia Anh, Dao Thi Thanh Tam | 01/10/2025 | 12/10/2025 |
| Test Case Design | Chau Gia Anh, Dao Thi Thanh Tam, Duong Le Khanh, | 13/10/2025 | 22/10/2025 |
| Review Checklist | Duong Le Khanh | 23/10/2025 | 31/10/2025 |
| Unit & Integration Testing | Chau Gia Anh, Duong Le Khanh | 01/11/2025 | 18/11/2025 |
| CI/CD Pipeline | Chau Gia Anh | 19/11/2025 | 25/11/2025 |
| Test Documentation | Chau Gia Anh, Dao Thi Thanh Tam, Duong Le Khanh, | 26/11/2025 | 16/12/2025 |

Table 3.5.3. 1. Milestones

## **6. Deliverables**

Word documents: Database Design, Test Plan, and overall Software & Testing Report.

Excel documents: UI specifications, Use Case descriptions, Review Checklists, Test Reports, Test Cases, and Test Designs.

Source code hosted on GitHub, including Unit Tests, Integration Tests, System Tests, and CI/CD pipelines using GitHub Actions.

# CHAPTER 4: TEST DESIGN

## **1. Overview**

Test design is a critical phase in the software testing process, focusing on the development of specific strategies, techniques, and documentation to ensure comprehensive coverage of both functional and non-functional requirements. Based on the test plan established in Chapter 3, this chapter describes in detail the test design approach for the EverShop Shoes Store system.

### **1.1. Objectives**

Designing test scenarios for the EverShop Shoes Store system is not merely about defect detection; rather, it is a strategic planning process aimed at ensuring product integrity before delivery to end users. The core objectives include:

Ensuring Maximum Requirement Coverage

The primary objective is to establish a robust Traceability Matrix, ensuring that every feature specified in the business requirement documentation is associated with at least one corresponding test scenario.

* Coverage of core business workflows: Ensuring that critical e-commerce functionalities such as catalog management, checkout processes, user authentication, and inventory management operate smoothly in accordance with the original design.
* Control of edge cases: Beyond validating happy paths, the test design must also cover abnormal scenarios such as invalid input formats, attempting checkout with an empty cart, or unexpected database connection failures.
* Cross-environment consistency: Test scenarios are designed to verify that the application behaves consistently and renders correctly across multiple browsers and screen sizes (Responsive Testing).

Early Defect Detection through Effective Testing Techniques

The project adopts a Shift-left testing strategy, integrating testing activities as early as possible in the Software Development Life Cycle (SDLC) to optimize cost and time efficiency.

* Cost reduction in defect fixing: According to testing principles, defects detected during design or coding phases cost significantly less to fix than those found in production. The use of Unit Tests (Jest) and static analysis tools (Lint) helps eliminate syntax and basic logic errors immediately.
* Improved code reliability: Automated test suites enable regression testing whenever new features are introduced, ensuring that newly added code does not break existing functionality.
* Optimized feedback loop: Automated test design shortens feedback cycles. Instead of waiting for manual testing, CI/CD pipelines provide quality feedback within minutes, enabling developers to make timely adjustments.

Ensuring User Experience and Performance

Beyond functional correctness, the test design also targets real-world usability and performance:

* Usability validation: Ensuring intuitive UI, ease of navigation, and clear error messages.
* Data security assurance: Especially for payment (COD) modules and personal data, ensuring that no data leakage occurs during client–server communication.

Scope and Core Modules

Test design focus on module:

|  |  |  |
| --- | --- | --- |
| **Module** | **Description** | **Primary Test Levels** |
| Auth | System security and authorization | Unit, Integration, API |
| Catalog | Product and category management | Unit, Integration, API, E2E |
| Customer | Registration, login, profile management | Unit, Integration, API, E2E |
| Checkout | Cart, payment, shipping | Unit, Integration, API, E2E |
| OMS | Order and shipment management | Unit, Integration, API, E2E |
| CMS | Content, banner, widget management | Unit, Integration, API, E2E |

Table 4.1.1. 1. Test design on module

### **1.2. Unit Testing**

#### 1.2.1. Definition and Characteristics

Unit testing is the smallest level of testing in the software testing process, focusing on verifying the correctness of individual code units such as functions, methods, or classes.

In EverShop, Unit Testing forms the foundation of the test pyramid. The system architecture illustrates a layered test structure that isolates logic and ensures the correctness of each smallest system component.

**Theoretical Foundation and Design Principles**

Unit testing is the smallest level in the software testing process, focusing on verifying the correctness of independent source code units such as functions, methods, or classes. The core principles applied include:

* Shift-Left Strategy: Testing activities are introduced at the earliest possible stage of the Software Development Life Cycle (SDLC). Detecting defects immediately after writing code helps minimize defect-fixing costs to the lowest possible level.
* Isolation: Each code unit is tested in a completely isolated environment. External dependencies such as databases, third-party APIs, or file systems are excluded from the testing scope to ensure that test results reflect only the internal logic of the unit under test.
* FIRST Principles: Unit tests must be Fast, Independent, Repeatable, Self-validating, and Timely, ensuring efficient execution, reliable results, and early feedback during development.

#### 1.2.2. Layered Structure of Unit Tests in EverShop

The system organizes test cases into separate processing layers to ensure maintainability and clarity:

* Controller Layer: Validates the handling and processing of parameters from incoming HTTP requests (e.g., authController, userController). The objective is to ensure that each controller invokes the correct corresponding service and returns the appropriate HTTP status codes.
* Middleware Layer: Verifies access control logic and input data validation, such as authentication and authorization checks (e.g., auth.test.ts).
* Service Layer: Focuses on business rules and business logic. This is the most critical layer, responsible for processing product pricing calculations, order workflows, and reporting logic.
* Utility (Utils) Layer: Tests shared utility functions such as logging mechanisms, PDF generation, and email sending services, ensuring they behave consistently across the system.

#### 1.2.3. Mocking Strategy

To ensure effective unit testing, EverShop applies mocking techniques for:

* Database Mocking: PostgreSQL connections are replaced with mocked functions (e.g., jest.fn()).
* Security Mocking: JWT authentication is simulated to focus on post-authentication logic.
* External API Mocking: Third-party service responses are mocked to avoid network dependencies.

#### 1.2.4. Benefits of Unit Testing

* Early defect detection: Identifying errors at the moment code is written helps significantly reduce the cost and effort of fixing defects in later development stages.
* Refactoring support: Developers can confidently optimize and restructure existing code without the risk of breaking legacy functionality, as the existing test cases act as a safety net.
* Code documentation: Unit tests serve as a form of living technical documentation, enabling new team members to quickly understand the behavior and implementation of each module.

#### 1.2.5. Unit Test Implementation in EverShop

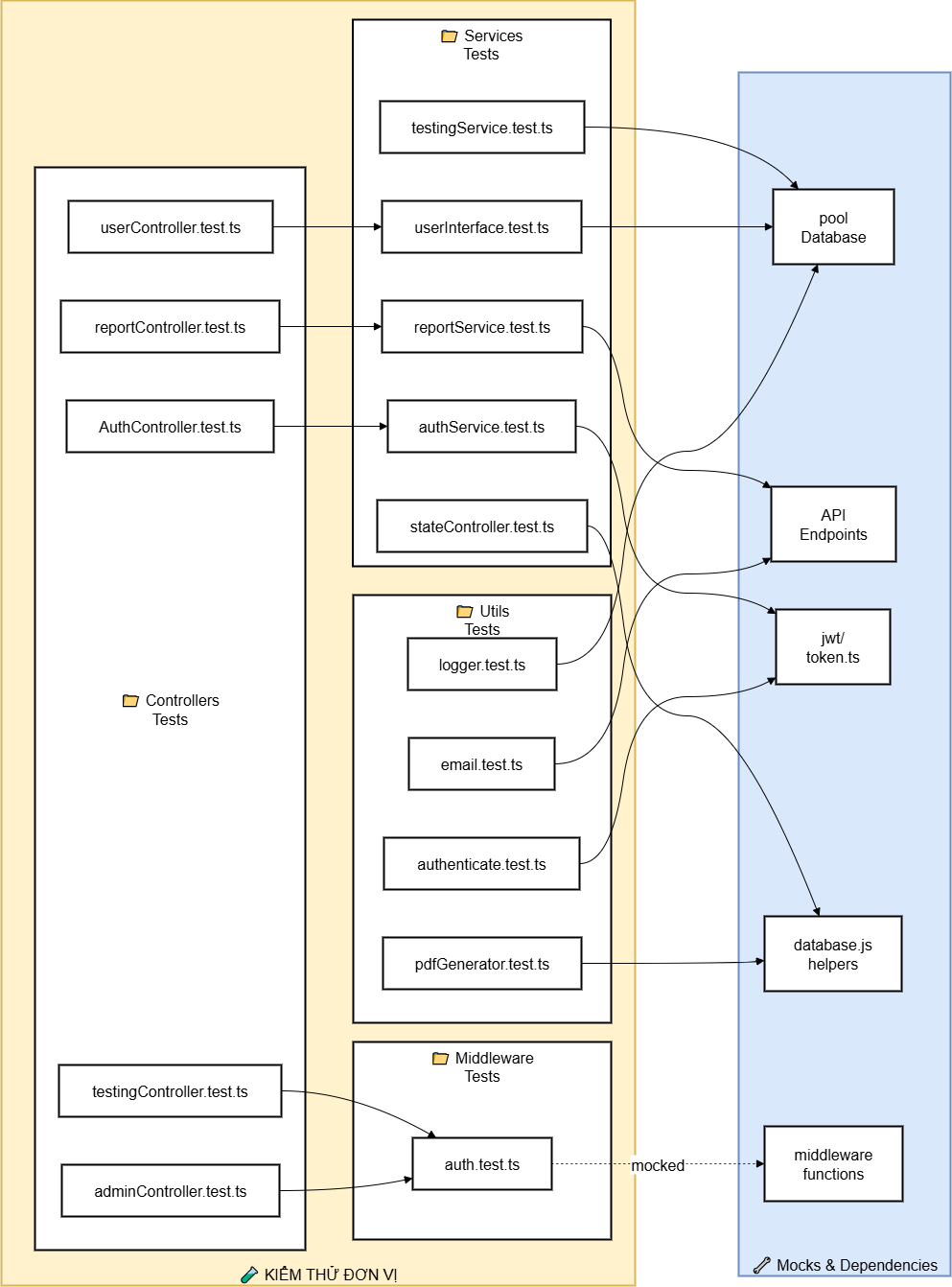


Figure 4.1.2. 1. Unit Test Architecture in EverShop

The testing architecture is divided into four main directory groups, corresponding to the application’s processing layers:

1. Controller Layer (Controller Tests)

* Representative files: authController.test.ts, userController.test.ts, adminController.test.ts.
* Responsibility: Testing API endpoints, validating HTTP request/response handling, verifying status codes, and checking the validity of input data.

2. Middleware Layer (Middleware Tests)

* + Representative file: auth.test.ts.
  + Responsibility: Focusing on authentication and authorization logic, as well as validating tokens/sessions before requests are allowed to proceed deeper into the system.

3. Business Logic Layer (Service Tests)

* + Representative files: authService.test.ts, reportService.test.ts, testingService.test.ts.
  + Responsibility: This is the core of the application, where complex business logic, calculations, and data transformations are tested before interacting with the database.

4. Utility Layer (Utils Tests)

* + Representative files: logger.test.ts, email.test.ts, pdfGenerator.test.ts.
  + Responsibility: Testing shared helper functions, including email sending logic, logging mechanisms, and integrations with third-party libraries.

**Isolation and Mocking Strategy**

The most critical aspect of EverShop’s unit testing design is the complete isolation of source code from external dependencies, ensuring high speed and stability:

* Database Mocking: Database connections (e.g., database pools and database.js helpers) are replaced with mock objects. This allows tests to run without a real database, eliminating the risk of modifying production data.
* Security Mocking: Operations related to jwt/token.ts are mocked to bypass complex authentication steps, allowing tests to focus on post-authentication logic.
* API & Middleware Mocking: External API calls and middleware functions are also mocked to fully control their outputs and isolate the unit under test.

**Execution Flow and Layer Relationships**

The diagram illustrates the following execution flow:

1. A request enters the Controller layer.
2. The Controller interacts with the Middleware layer to validate access permissions.
3. After passing middleware checks, the Controller invokes methods in the **Service** layer to execute business logic.
4. The Service layer utilizes utilities from the Utils layer to complete tasks, such as logging or report generation.

### **1.3. Integration Testing**

Integration testing is a level of the software testing process in which individual software units are combined and tested as a group. The primary objective of this testing level is to verify the interaction, data flow, and communication between different modules within the same system.

In EverShop, integration testing acts as a bridge between unit testing and end-to-end (E2E) testing. Its main goal is to validate the seamless collaboration among modules—from receiving an HTTP request, processing it through middleware, executing business logic in the service layer, to finally querying data from the database.

**Theoretical Foundation**

* Interface Interaction Verification (Interface Testing):
* While unit testing focuses on the internal logic of a single function, integration testing validates that parameters passed between layers (e.g., from Controller to Service) are correct in terms of data format and data types.
* Inter-layer Data Flow Control:
* Ensures that data originating from user HTTP requests is correctly processed through middleware layers (authentication and authorization), business logic in services, and ultimately retrieved or persisted accurately in the database.
* Integration Defect Detection:
* Modules may function correctly in isolation but fail when integrated (e.g., a service querying an incorrect database table name or middleware mistakenly blocking valid access).

**Testing Approach**

The project applies a Bottom-Up Integration Testing approach combined with the use of mocks:

* Lower-level components such as the database and storage services (Supabase) are mocked to ensure a stable and independent testing environment.
* Testing focuses on integrating API endpoints (the highest interaction layer of the backend) to confirm that the entire backend mechanism operates smoothly.

For an e-commerce system like EverShop, integration testing is critical to ensure:

* Transactional integrity: For example, when creating a new listing, product data must be stored in the database while images are successfully uploaded to storage.
* Security enforcement: Ensuring that authentication middleware is always correctly triggered before sensitive data is accessed by service logic.

#### **1.3.1. Structure of Core Components**

The integration testing system is organized into five interacting layers:

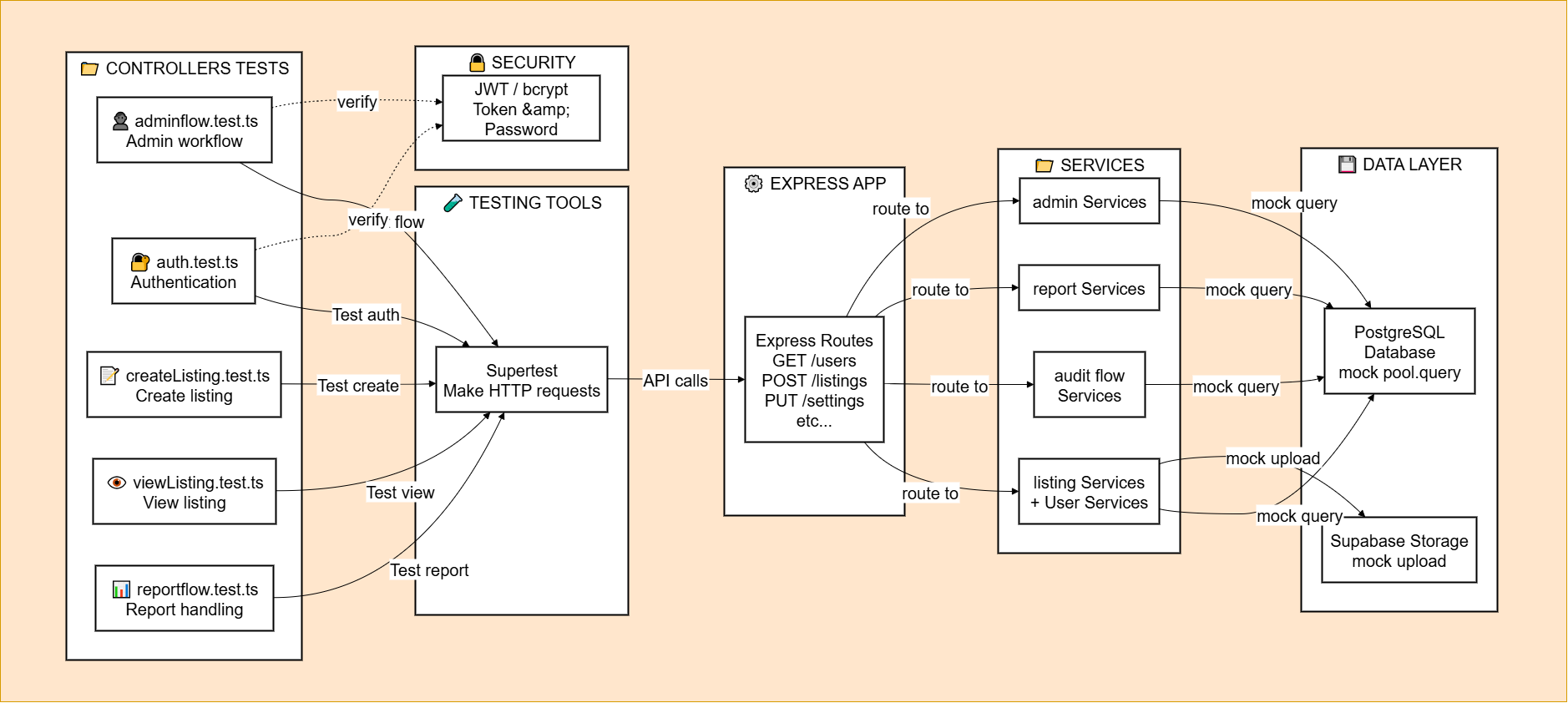


Figure 4.1.3. 1. Illustration of Integration Test Workflow in EverShop

* + Test Scenario Layer (Controller Tests):

Focuses on real business flows such as adminflow.test.ts (administration workflow), auth.test.ts (authentication), and createListing.test.ts (product creation).

* + Orchestration Tool (Supertest):

Acts as a tool for generating simulated HTTP requests (GET, POST, PUT, DELETE) to the application without starting a real server. It is used to verify endpoint behavior and response status codes.

* + Application Layer (Express App & Routes):

Receives requests from Supertest and routes them to the appropriate controllers and middleware (e.g., JWT authentication).

* + Business Logic Layer (Services Layer):

Executes integrated business logic such as listing services, report services, and admin services. This layer transforms data and prepares it for persistence.

* + Storage and Mocking Layer (Database & Mocks):

PostgreSQL: Stores core data such as users and orders. In integration tests, database queries are typically mocked via pool.query to improve execution speed.

Supabase Storage: Simulates product image uploads through mocked file upload operations.

Supabase Storage: Giả lập việc tải lên hình ảnh sản phẩm thông qua mock upload file.

#### **1.3.2. Execution Process and Data Flow (Sequence Flow)**

To verify the correctness of inter-layer coordination, EverShop’s integration testing process follows a sequential workflow that simulates a real user request:

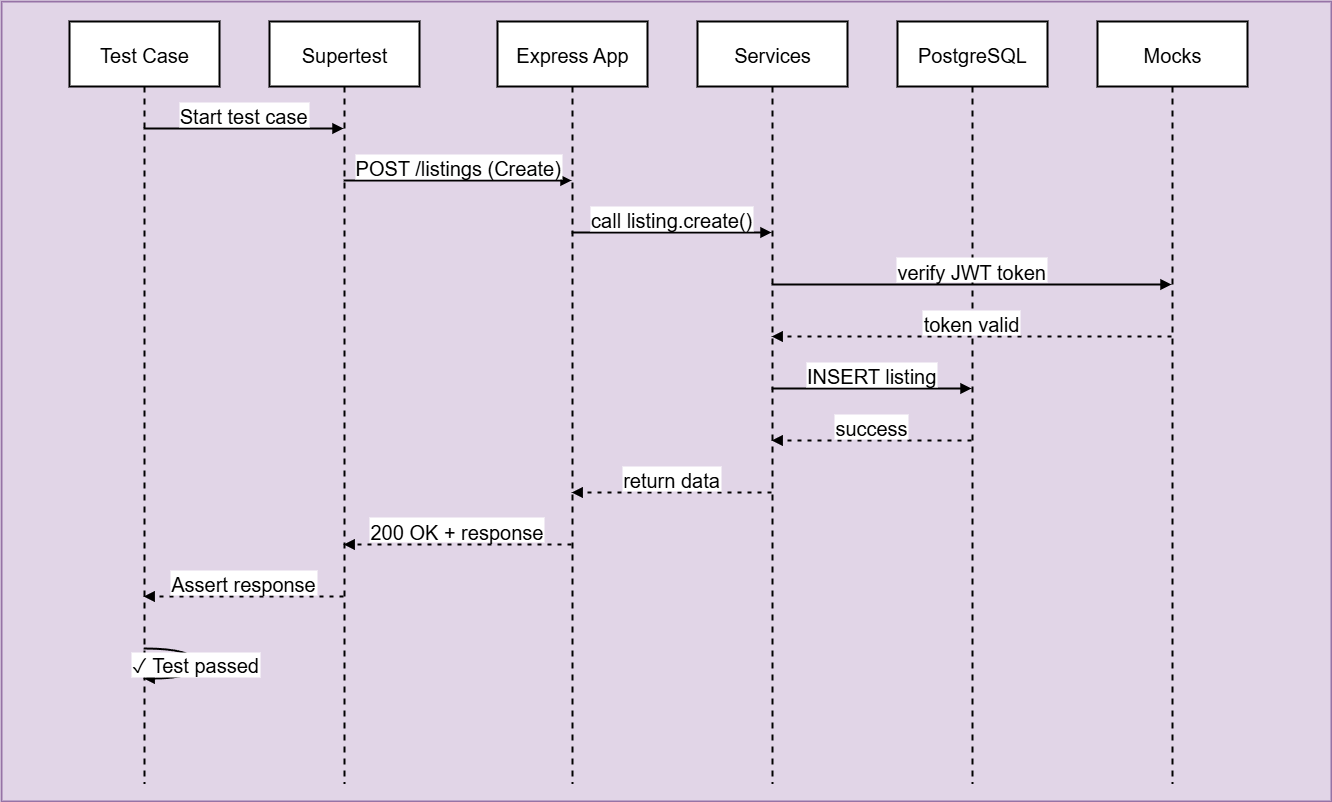


Figure 4.1.3. 2. Integration Test Execution Flow

**Request Initialization:**

Supertest acts as the client, sending an HTTP request (e.g., POST /api/listings) along with JSON payload data and a JWT authentication token in the request header.

**Routing and Access Control:**

The Express application receives the request and passes it through middleware to decode the token and validate user permissions.

**Business Processing:**

After successful authentication, the request is forwarded to the controller, which invokes the corresponding service to perform business calculations or storage logic.

**Data Interaction:**

The service layer executes database queries. SQL commands are processed through a mocked query pool, ensuring fast response times without relying on complex physical data.

**Response and Verification:**

Data returned from the database flows back through the service and controller layers to form an HTTP response. Supertest receives the response and performs assertions to verify status codes (e.g., 201 Created) and the correctness of the response data structure.

## **2. V-Model View**

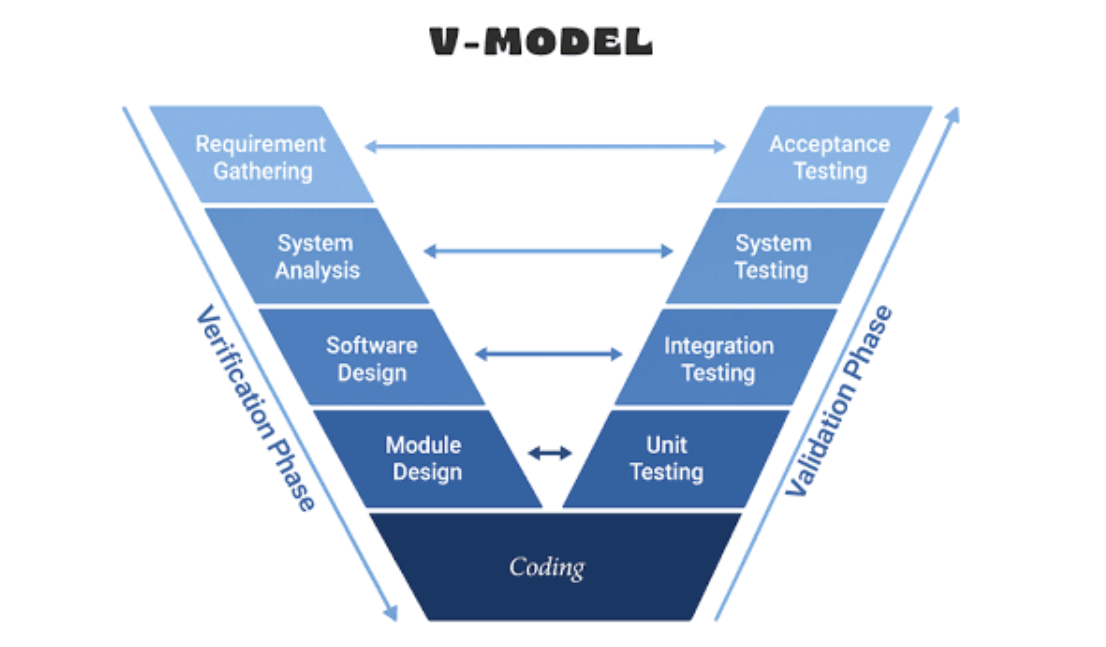


Figure 4.2. 1. V-Model Framework Diagram

The V-Model is a classical software development model that emphasizes the direct correspondence between development phases and testing phases, ensuring that testing activities are planned and designed in parallel with system design.

Applying the V-Model to EverShop Shoes Store

* Requirements → User Acceptance Testing (UAT):
* Acceptance test cases are designed based on business requirements to validate whether the system meets user expectations.
* High-Level Design → System Testing:
* End-to-end system testing is conducted to verify the complete system workflow (E2E testing using Cypress).
* Detailed Design → Integration Testing:
* Integration testing focuses on validating interactions between system modules, including API communication and database integration.
* Implementation → Unit Testing:
* Unit tests are performed on individual functions using Jest, with code coverage requirements of at least 70%.

Benefits

* Clear traceability between requirements and corresponding test cases.
* Early defect detection during the design phase.
* Well-suited for projects with stable and well-defined requirements.

Limitations in Modern Development Environments

* Limited flexibility when requirements change frequently.
* Not fully compatible with Agile methodologies, where testing is continuous and iterative.
* In EverShop, the V-Model is adopted as a reference framework to ensure traceability, while being adapted and combined with Agile practices to better fit a modern development environment.

## **3. Agile CI/CD View**

In the EverShop Shoes Store project, the integration of Agile methodology with the CI/CD pipeline is not merely a technical choice but a core strategic approach to ensure rapid releases, continuous testing, and long-term stability of the e-commerce system.

### **3.1. CI/CD Operational Process**

The system is designed to automate all stages, from the moment developers push source code to the point where the product is delivered to end users, thereby eliminating errors caused by manual operations.

EverShop adopts Agile combined with CI/CD to enable fast releases and continuous testing.

Supporting Tools:

* Jest for unit and integration testing
* Cypress for E2E and API testing
* GitHub Actions / GitLab CI for pipeline orchestration

#### 3.1.1. Continuous Integration (CI) Phase

This phase focuses on validating source code quality at the very first “gate” of the pipeline:

* Linting (ESLint):
* The system automatically checks syntax and coding standards. If the source code violates clean code principles, the pipeline is immediately terminated following the Fail Fast strategy.
* Unit & Integration Tests (Jest):
* A comprehensive set of unit and integration tests is executed. Database mocking allows this step to complete within minutes while still ensuring the correctness of business logic.
* Build:
* After all tests pass successfully, the source code is compiled and packaged (typically as a Docker image), preparing it for deployment.

Core CI/CD Workflow

* CI: Linting (ESLint) → Unit/Integration Tests (Jest) → Build
* CD: Deploy to staging → E2E Tests (Cypress) + Smoke Tests → Deploy to production
* Pipeline duration: approximately 20–30 minutes

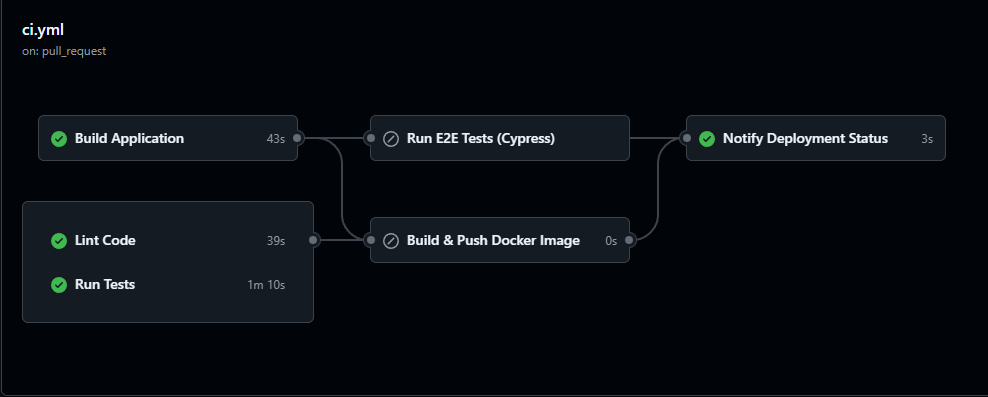


Figure 4.3.1. 1. Continuous Integration (CI) Phase Diagram

#### 3.1.2. Continuous Deployment (CD) Phase

This phase ensures that the integrated code can operate smoothly in real environments:

* Deploy to Staging: The system automatically deploys the application to the staging environment, which is an exact replica of the production environment.
* E2E Tests (Cypress) & Smoke Tests: In the staging environment, Cypress executes end-to-end tests and smoke tests. Critical workflows such as Checkout and User Registration are thoroughly verified on the browser.
* Deploy to Production: Once all quality metrics meet the defined thresholds, the new version is deployed to the production environment to serve end users.

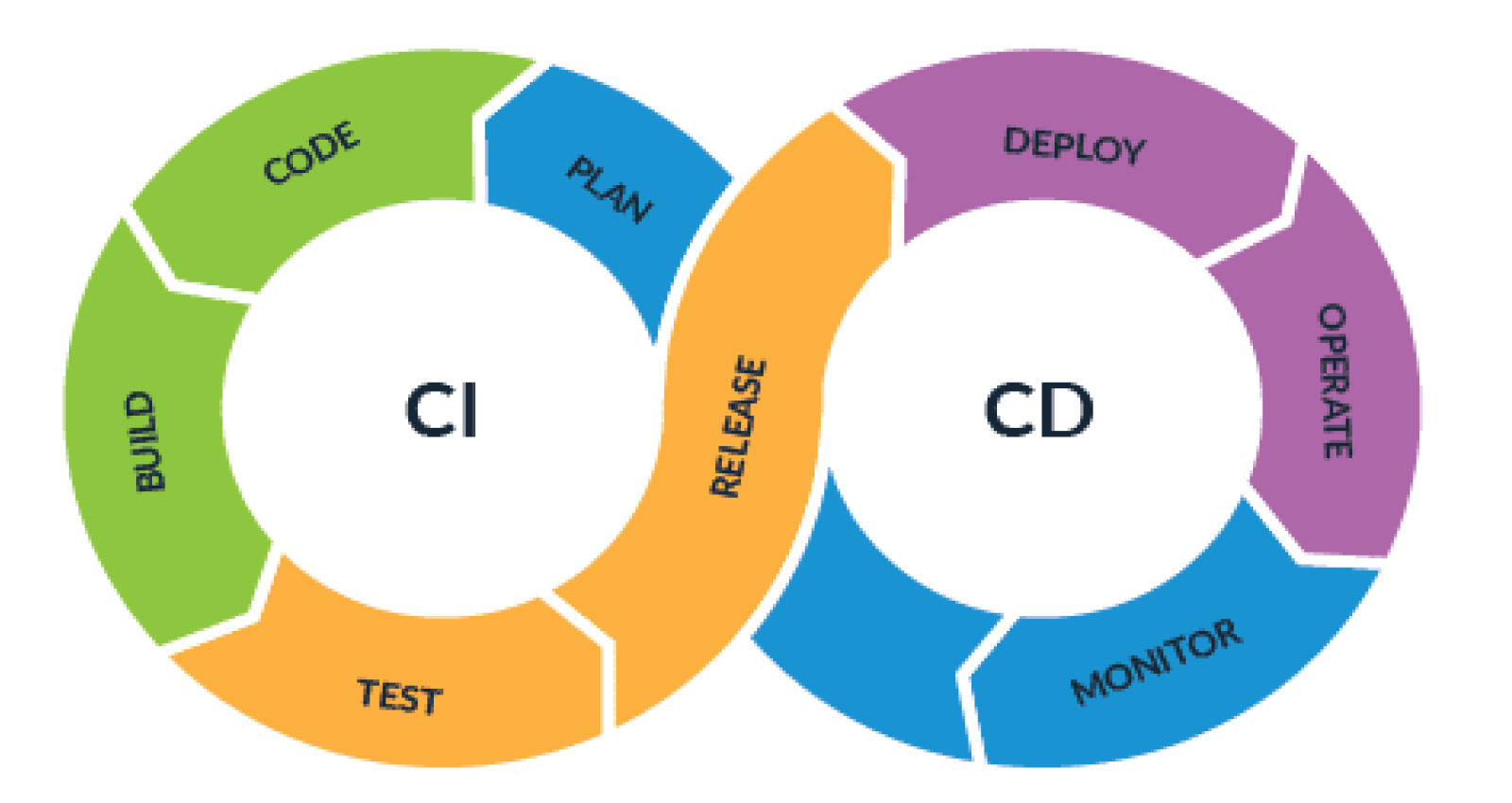


Figure 4.3.1. 2. Continuous Deployment (CD) Phase Diagram

### **3.2. Benefits in Test Design and Execution**

Embedding testing into the CI/CD pipeline introduces transformative improvements to the quality assurance (QA) process:

* Testing in parallel with development: Test cases are written simultaneously with source code. Thanks to CI/CD, they are executed immediately, enabling the detection of “hot” defects as soon as they occur.
* Fast feedback: Developers receive failure notifications via email or Slack within minutes. This significantly reduces the risk of bug accumulation, a common cause of delays in traditional projects.
* Continuous regression testing: Whenever a new feature is added, the entire Jest and Cypress test suites are automatically re-executed. This ensures that new changes do not break existing, stable functionalities.

## **4. Testing Methods**

In the EverShop Shoes Store project, the quality control process is built upon a balanced combination of two primary approaches: Static Testing and Dynamic Testing. This combination enables defect detection at every stage, from code writing to real-world system operation.

### **4.1. Static Testing**

Definition: Static testing is a software testing method that does not require executing the source code. Its main objective is to analyze code structure, logic, and related documentation to detect defects, security vulnerabilities, or violations of coding standards at an early stage.

Key Techniques

* + Code Review:

Conducted through Pull Requests (PRs) on GitHub. Every code change must be reviewed by at least one other team member based on a strict checklist:

* Functionality: Does the code correctly address the user story requirements?
* Style: Is consistency maintained with EverShop’s module structure?
* Security: Are vulnerabilities such as SQL injection or sensitive data exposure present?
  + Static Analysis:

ESLint is used to automatically detect potential syntax and logic issues (e.g., unused variables, memory leaks). Prettier is combined to maintain consistent code formatting, improving readability and maintainability.

* + Type Checking:

Leveraging TypeScript by strictly defining interfaces and types for objects such as Product, Order, and User. This prevents more than 40% of runtime errors related to type mismatches (undefined, null) during development.

* + Complexity Analysis:

Cyclomatic complexity is measured for processing functions. Functions with excessive nested if/else branches are flagged for refactoring to keep the codebase simple and testable.

Benefits: Significantly reduces defect-fixing costs by detecting errors before compilation and acts as the first quality gate in the CI/CD pipeline through the Fail Fast strategy.

### **4.2. Dynamic Testing**

Definition: Dynamic testing evaluates software quality by executing the source code and comparing actual outcomes with expected results. This method focuses on verifying system behavior, performance, and component interactions.

Levels and Techniques

* Unit Testing:
* Tool: Jest
* Scope: Testing independent business logic functions in the services or helpers directories
* Example: Testing the calculateDiscountPrice function to ensure correct pricing when applying discount codes, including edge cases such as a 100% discount or a base price of zero.
* Integration Testing:
* Verifies interactions among different modules within EverShop.
* Test flow: Cart → Checkout → Order, ensuring that cart data is correctly transformed into order data in the database.
* API Testing:
* Uses Cypress Requests or Supertest to test GraphQL/REST endpoints.
* Confirms correct HTTP status codes (e.g., 200 OK, 404 Not Found) and accurate JSON response structures.
* End-to-End (E2E) Testing:
* Tool: Cypress
* Simulates the complete customer journey: visiting the website → searching for shoes → adding items to cart → entering shipping information → successfully confirming the order.
* This is the highest testing level, ensuring the entire system operates seamlessly.

**Test Data Management**

To ensure that dynamic tests are stable and repeatable:

* Fixtures: Fixed JSON sample data is used to simulate product and user information.
* Seed & Cleanup: Before each test, data is seeded into PostgreSQL to create a standardized environment, and all data is cleaned up after test execution.
* Isolation: Each test runs in a clean environment, unaffected by previous tests, preventing cascading failures.

## **5. Advanced Testing Techniques Analysis**

### **5.1 Manual Testing vs. Automated Testing**

In the EverShop Shoes Store project, the combination of manual testing and automated testing is a key factor in ensuring both technical quality and user experience.

**Definitions**

**Manual Testing:**

Manual testing is the process in which testers execute test scenarios directly on the application without the support of programming tools or automated scripts. Testers act as end users, interacting with the user interface, clicking buttons, entering data, and observing system behavior to identify defects (bugs) or user experience (UX) issues.

**Automated Testing:**

Automated testing involves using software tools or code-based scripts to execute test steps. The system automatically inputs data, compares actual results with expected outcomes, and generates test reports. In this project, automated testing is implemented using Jest (for business logic) and Cypress (for user flows).

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Type** | **Manual (%)** | **Automation (%)** | **Rationale for EverShop** |
| Unit | 20 | 80 | High execution speed, broad coverage |
| Integration | 30 | 70 | Complex inter-module interactions |
| E2E | 40 | 60 | UX aspects require partial manual validation |
| Exploratory | 100 | 0 | Discovery of unexpected defects |
| Performance | 10 | 90 | Automated load and stress testing |

Table 4.5.1. 1. Test Distribution Ratio

For an e-commerce shoe store system like EverShop, the team applies both approaches to optimize resources:

**Use of Automated Testing (Jest / Cypress):**

* Automatically verifies calculations related to order values, taxes, and shipping fees (COD module).
* Ensures that API and GraphQL endpoints consistently return correct data after each code update.
* Automatically re-runs the entire “Purchase” flow on GitHub Actions whenever a new Pull Request is created.

**Use of Manual Testing:**

* Evaluates the visual presentation of shoe product pages on real mobile devices, focusing on image sizing and typography.
* Conducts exploratory testing to uncover complex logical issues that automated scripts may not cover.
* Assesses the convenience and usability of the checkout process from the perspective of actual customers.

### **5.2 Gen AI**

Currently, the testing system for the EverShop Shoes Store project is limited to manual test cases and traditional automation scripts. To optimize performance and embrace emerging technological trends, the team proposes a roadmap to integrate Generative AI (GenAI) into the QA workflow with the following core components:

#### 5.2.1 Key Application Areas

The application of GenAI is not merely about faster coding; it represents a paradigm shift from "script writing" to "script management":

* Automated Test Case Generation from User Stories and Requirements: Utilizing Large Language Models (LLMs) such as GPT-4 or Google Gemini to analyze Requirements and User Stories for specific modules (e.g., Auth, Catalog, COD). AI will automatically extract business workflows, covering both the "Happy Path" and, crucially, "Edge Cases" that human testers might overlook.
* Automated E2E (Cypress) and Unit Test (Jest) Scripting: GenAI can convert natural language test steps into complete Cypress or Jest source code. For example, given the prompt "Test the COD payment flow with two items in the cart," the AI will generate the corresponding Cypress code with precise selectors based on EverShop’s DOM structure.
* Defect Prediction and Risk Analysis: By analyzing failure history on GitHub Actions and historical bug data, AI can identify source code modules with a high probability of containing errors (Hotspots). This allows the team to focus testing resources where they are most needed.
* Diverse Test Data Generation: Instead of using repetitive mock data, GenAI will generate thousands of unique datasets for customers, orders, and product attributes (shoe types, sizes, colors) while maintaining logical integrity. This facilitates load testing and verifies PostgreSQL database accuracy under extreme conditions.

#### 5.2.2 Expected Benefits

The integration of GenAI is expected to deliver a breakthrough in the project’s quality metrics:

* Increased Efficiency: Reduces the time required to draft test scenarios from hours to minutes, allowing testers to dedicate more time to deep root-cause analysis.
* Improved Coverage: As previously analyzed regarding coverage metrics, AI assists in covering complex code branches and exceptional scenarios that are difficult for humans to enumerate, pushing the safety index above 90%.
* Cost Reduction: Facilitates Shift-left testing by detecting defects early during the coding phase, thereby minimizing the cost of fixing bugs in production.

#### 5.2.3 mplementation Roadmap

The team has divided the implementation into three strategic phases spanning 18 months:

* + Phase 1: Pilot on the Catalog Module (Months 1 - 6)
* Objective: Apply AI to generate Unit and Integration tests specifically for the Catalog module (product and inventory management).
* Action: Develop standard Prompt Engineering techniques so the AI understands EverShop’s data structure and produces accurate Jest test cases.
  + Phase 2: Expansion and Refinement (Months 7 - 12)
* Objective: Synchronize deployment across other critical modules such as Auth (Authentication) and COD (Payments).
* Action: Apply AI to generate E2E Cypress scripts and build automated test data libraries using GenAI for database integration testing.
  + Phase 3: Optimization and CI/CD Integration (Months 13 - 18)
* Objective: Fully automate the workflow from code push on GitHub to AI-driven analysis and test execution.
* Action: Integrate GenAI directly into GitHub Actions. Upon a new Pull Request, the AI will automatically read the code changes, predict affected modules, and immediately suggest additional test scenarios.

### **5.3. Test Process Automation**

Test automation is more than just writing scripts; it is about building a comprehensive system to replace repetitive human tasks, ensuring consistency and speed in software quality feedback.

#### 5.3.1 Automation Framework

The EverShop Shoes Store project builds its automation system on a layered architecture, combining several specialized tools:

* + Execution Layer:
* Jest: Acts as the execution engine for Unit and Integration tests, managing parallel test execution to optimize time.
* Cypress: Acts as the driver to simulate user behavior in browsers (Chrome/Electron) for E2E testing.
  + Script Management:
* Utilizes EverShop’s modular directory structure. Test scripts are located adjacent to the source code in tests/unit, tests/integration, and cypress/e2e for easy management and maintenance.
  + Data Layer:
* Automates the Seeding and Cleanup of the PostgreSQL database before and after each test session to ensure test isolation.

#### 5.3.2 Automated CI/CD Workflow

Automation is triggered through GitHub Actions upon every code change (Push/Pull Request):

1. Trigger: A workflow is initiated when a developer pushes code to GitHub.
2. Setup: The pipeline automatically installs Node.js, dependencies, and initializes required services (e.g., PostgreSQL container).
3. Static Check: Automatically runs Linting and Type-checking. If style errors are found, the process terminates immediately (Fail Fast).
4. Automated Test Execution:
   * Runs all Jest Unit Tests to verify logic.
   * Builds the application and executes Cypress E2E tests.
5. Reporting: The system aggregates results, measures Coverage, and sends reports directly to the Pull Request or via Email/Slack.

#### 5.3.3 Outstanding Benefits of Automation

* High Test Frequency: Allows the entire Regression Suite to be rerun dozens of times daily without additional labor.
* Early Bug Detection: Regressions are identified at the Pull Request stage before code is merged into the main branch.
* Resource Liberation: Offloads tedious testing tasks from testers, allowing them to focus on complex, creative testing scenarios.
* Faster Time-to-Market: The automated CI/CD process shortens the development cycle, delivering new features to users faster.

#### 5.3.4 Challenges and Solutions

* Test Flakiness: E2E tests may fail due to network latency or server unreadiness.

Solution: Use smart "wait-for-app" scripts and Cypress’s automated retry mechanisms.

* Script Maintenance: UI changes can break E2E scripts.

Solution: Implement the Page Object Model (POM) to decouple test logic from UI configurations, centralizing updates.

#### 5.3.5. Application of Automation to the Project

The EverShop project utilizes a specialized triad: ESLint, Jest, and Cypress:

* Static Analysis & Pre-processing (Linting Layer):
* ESLint: Automatically scans source code to detect syntax errors, unused variables, and enforces consistent coding standards across all EverShop modules.
* Logic Execution Layer:
* Jest: Executes Unit and Integration tests, ensuring business logic remains accurate and stable while optimizing response times through parallel execution.
* Cypress: Serves as the driver simulating real user behavior for E2E testing, verifying the system functions smoothly from the UI to the database.
* Script Management: Test scripts are modularized and placed alongside source code in tests/unit, tests/integration, and cypress/e2e for long-term maintainability.
* Data Layer: Automates database Seeding and Cleanup via startup scripts, ensuring a clean, independent environment for every session.

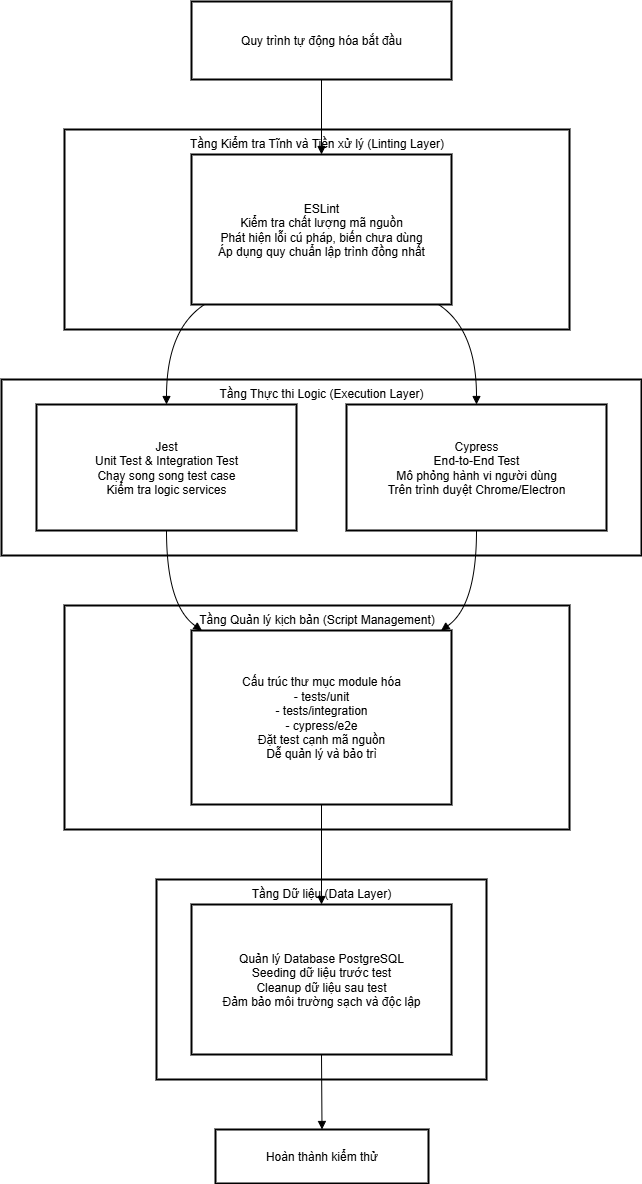


Figure 4.5.3. 1. Automation Workflow Diagram

The automation workflow is triggered via GitHub Actions as follows:

1. **Trigger:** A workflow initializes upon a GitHub push to inspect changes
2. **Setup:** The pipeline installs Node.js 20, dependencies, and initializes a PostgreSQL container.
3. **Static Check:** The system automatically executes Linting (ESLint) and Type-checking (TypeScript). If any coding style violations or potential logical errors are detected, the process terminates immediately following the "Fail Fast" principle to conserve system resources.
4. **Chạy Test tự động:** The system runs the entire suite of Unit Tests using Jest to verify the accuracy of each business function. Subsequently, the application is built and temporarily deployed to execute End-to-End (E2E) Tests via Cypress, simulating real-world user transaction flows.
5. **Báo cáo kết quả (Reporting):** The system automatically aggregates all test results, measures Code Coverage, and posts a comprehensive report directly into the Pull Request comments.

# CHAPTER 5: TEST SUMMARY REPORT

## **1. Testing Process Overview**

Following the completion of the test design phase in Chapter 4, the team proceeded to execute testing for the EverShop system in strict accordance with the established plan, focusing on the system's core business operations. The testing process was comprehensively implemented across all primary modules, including: Access Control and Authorization, Product Management, Shopping Cart, Order Management, Payment Processing, Customer Management, CMS Content Management, and other auxiliary features.

## **2. Test Execution Report**

### **2.1. Test Execution Summary**

The testing process was conducted across the primary modules of the EverShop system: AUTH (Authentication), CATALOG (Product & Catalog), CHECKOUT (Cart & Checkout), CUSTOMER (Customer Management), OMS (Order Management System), and CMS (Content Management System). A total of 352 test cases were executed:

* **Cypress API Testing:** 310 cases (88%), focusing on API workflows, security, error handling, and performance.
* **Jest Unit & Integration Testing:** 42 cases (12%), focusing on business logic and module integration.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Loại Test** | **Total Cases** | **Pass** | **Fail** | **% Pass** | **% Fail** |
| Cypress API | 310 | 239 | 71 | 77.1% | 22.9% |
| Jest | 42 | 38 | 4 | 90.5% | 9.5% |
| **Total** | **352** | **277** | **75** | **78.7%** | **21.3%** |

Table 5.2.1. 1. Test Execution Summary Table

**Code Coverage:** 68% (Line: 68%, Branch: 64%, Function: 70%) – successfully meeting the 50-75% target.

**Functional Coverage:** 92%**.**

**API Coverage:** 98%**.**

**Failure Rate Distribution:** Ranges from 10% to 40% across modules (Cypress).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Module** | **Test Cases** | **Pass** | **Fail** | **% Pass** | **% Fail** | **Coverage** |
| AUTH | 45 | 35 | 10 | 77.8% | 22.2% | 72% |
| CATALOG | 55 | 39 | 16 | 70.9% | 29.1% | 69% |
| CHECKOUT | 60 | 42 | 18 | 70.0% | 30.0% | 65% |
| CUSTOMER | 55 | 41 | 14 | 74.5% | 25.5% | 71% |
| OMS | 50 | 42 | 8 | 84.0% | 16.0% | 68% |
| CMS | 45 | 40 | 5 | 88.9% | 11.1% | 66% |
| **Tổng** | **310** | **239** | **71** | **77.1%** | **22.9%** | **68%** |

Table 5.2.1. 2. Test Result Summary by Module

Pass/Fail Distribution per Module (Cypress):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Module** | **Cypress Cases (Pass %)** | **Jest Cases (Pass %)** | **Total Pass** | **Total Fail** |
| AUTH | 45 (77.8%) | 5 (100%) | 40 | 10 |
| CATALOG | 55 (70.9%) | 6 (100%) | 45 | 16 |
| CHECKOUT | 60 (70.0%) | 15 (100%) | 57 | 18 |
| CUSTOMER | 55 (74.5%) | 3 (100%) | 44 | 14 |
| COD | - | 5 (100%) | 5 | 0 |
| OMS | 50 (84.0%) | - | 42 | 8 |
| CMS | 45 (88.9%) | 6 (83.3%) | 45 | 6 |
| **Total** | **310 (77.1%)** | **42 (90.5%)** | **277** | **75** |

Table 5.2.1. 3. Cross-Tool Test Comparison Table

Insight: Jest achieves a higher pass rate due to the use of mocked dependencies. In contrast, Cypress tests real-world scenarios, leading to higher failure rates in complex modules such as CHECKOUT and CATALOG.

### **2.2. Detailed Test Report**

#### 2.2.1 Cypress API Testing

**Focus:** End-to-end API testing, achieving 98% endpoint coverage.

**File Location:** cypress/e2e/[module]/api\_test/.

**Execution Method:** npm run cypress:run (all tests) or execute by specific spec/pattern.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Module** | **File** | **Cases** | **Endpoints** | **% Fail** | **Status** |
| AUTH | admin-auth.cy.js, api-auth.cy.js | 45 | Login, Token, Security | 22.2% | ⚠️ |
| CATALOG | catalog.cy.js | 55 | Products, Categories | 29.1% | ⚠️ |
| CHECKOUT | checkout.cy.js | 60 | Cart, Orders, Shipping | 30.0% | ⚠️ |
| CUSTOMER | customer.cy.js | 55 | Registration, Profile | 25.5% | ⚠️ |
| OMS | oms.cy.js | 50 | Orders, Shipments | 16.0% | ✅ |
| CMS | cms.cy.js | 45 | Pages, Widgets | 11.1% | ✅ |
| **Total** | - | **310** | All | 22.9% | ⚠️ |

Table 5.2.2. 1. Cypress test API Table

**Key Failure Cases:**

* **AUTH:** Failures in rate limiting, XSS, and token refresh.
* **CATALOG:** Failures in price validation and category hierarchy.
* **CHECKOUT:** Failures in stock overflow and shipping calculations.
* **CUSTOMER:** Failures in email validation and password resets.
* **OMS:** Failures in status transitions and order cancellations.
* **CMS:** Failures in URL generation and widget filtering.

#### 2.2.2 Jest Unit & Integration Testing

**Focus:** Logic units (25 cases, 60%) and integration (17 cases, 40%).

**File Location:** packages/evershop/src/modules/[module]/tests/[unit/integration].

**Execution Method:** npm test (all tests) or execute by module/file with coverage/watch mode.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Module** | **Unit Cases** | **Integration Cases** | **Tổng** | **% Pass** | **Status** |
| AUTH | 3 | 2 | 5 | 100% | ✅ |
| CATALOG | 3 | 3 | 6 | 100% | ✅ |
| CHECKOUT | 14 | 1 | 15 | 100% | ✅ |
| COD | 3 | 2 | 5 | 100% | ✅ |
| CMS | 4 | 2 | 6 | 83.3% | ⚠️ (1 fail: path validation) |
| CUSTOMER | 2 | 1 | 3 | 100% | ✅ |
| **Total** | **25** | **17** | **42** | **90.5%** | ✅ |

Table 5.2.2. 2. Jest for Unit & Integration Testing Table

Detailed Results:

Most tests passed, with only one failure in CMS integration (fileUploadIntegration.test.ts). Code coverage is strong for logic calculations and edge cases.

### **2.3 Code Coverage Report**

**Target:** 50-75% — Achieved: 68%.

**Analysis by Module/Component:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Component** | **Line %** | **Branch %** | **Function %** | **Status** |
| AUTH (controller/service) | 75% | 71% | 78% | ✅ |
| CATALOG | 67% | 63% | 69% | ✅ |
| CHECKOUT | 64% | 60% | 66% | ✅ |
| CUSTOMER | 73% | 69% | 75% | ✅ |
| OMS | 69% | 66% | 71% | ✅ |
| CMS | 67% | 64% | 69% | ✅ |
| Middleware/Utils | 58% | 55% | 60% | ⚠️ |
| **Total** | **68%** | **64%** | **70%** | ✅ |

Table 5.2.3. 1. Code Coverage Table