GameHub - E-commerce Platform Requirements Document

I. Requirements Document

1. Introduction

1.1 Project Name:

GameHub - E-commerce Platform

1.2 Purpose

The purpose of this document is to outline the functional and non-functional requirements for the development of a Game Code E-commerce Platform (GameHub). This platform will serve as a central hub for distributing digital game codes to gamers, that facilitates the seamless purchase of digital codes, ensuring both a rich user experience.

1.3 Scope

The Game Code E-commerce Platform will encompass the following key features and functionalities:

- User Registration and Authentication
- User Profile Management
- Product listing, categorization, and search
- Shopping cart and checkout process.
- Payment gateway integration
- Security and Privacy

1.4 Audience

This document is intended for the product owner and project developers involved in the development of the GameHub Platform.

2. Functional Requirements

2.1 User Management

- Allow new users to register for an account to purchase and manage digital codes.
- Enable registered users to securely and efficiently log into their accounts.
- Allow users to create and manage personal profiles, including updating personal information.
- Provide a secure method for users to reset passwords and recover accounts.

- Enable admins to manage and assign user roles and permissions.
- Admins should have tools to monitor and manage user accounts.

2.2 Product Management

- Allow admins to list and categorize products for easy customer navigation.
- Provide comprehensive product details for customers.
- Streamline the process of creating and cataloging new products.
- Enable quick and accurate updates to product information.
- Secure process for deleting or archiving products.
- Implement an accurate inventory management system.
- Provide reporting and analytics tools for admins.
- Implement a search system with filters for customers.

2.3 Shopping Experience

- A user-friendly system for managing shopping carts.
- Ensure a secure and efficient checkout process.
- Automatically distribute digital codes post-purchase.
- Enable customers to easily access their order history.
- Allow customers to create and manage a Wishlist.

2.4 Review System

- Allow customers to submit reviews and ratings for purchased products.
- Enable potential buyers to view and navigate through product reviews.
- Admins can moderate and manage customer reviews.

3. Non-Functional Requirements

3.1 Performance

- The platform should be responsive and capable of handling high user traffic.
- Response times for actions like game purchases should be minimal.

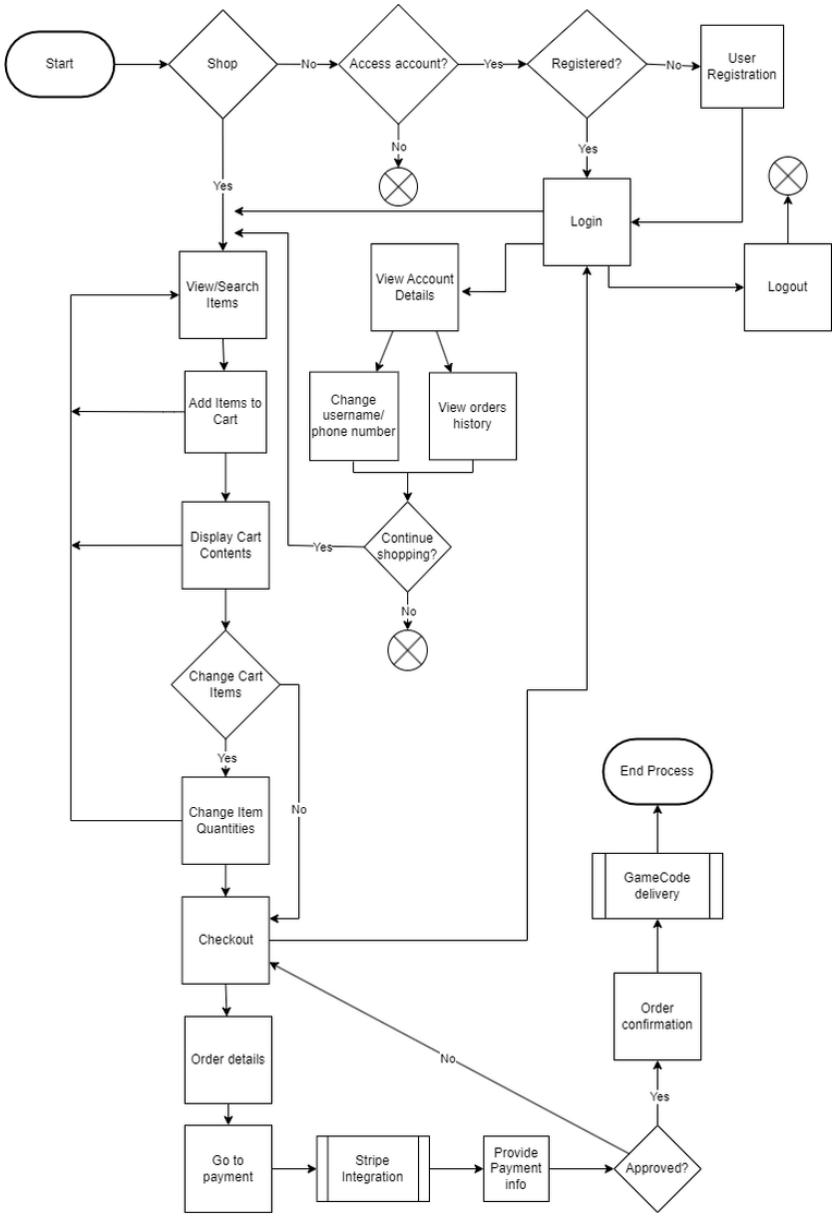
3.2 Scalability

- The platform should be designed for easy scalability to accommodate future growth.

3.3 Security

- SSL implementation for secure data transmission.
- SQL injection protection and regular security audits.
- Data Encryption and Protection: Implement comprehensive data encryption and protection measures.

4. Usability

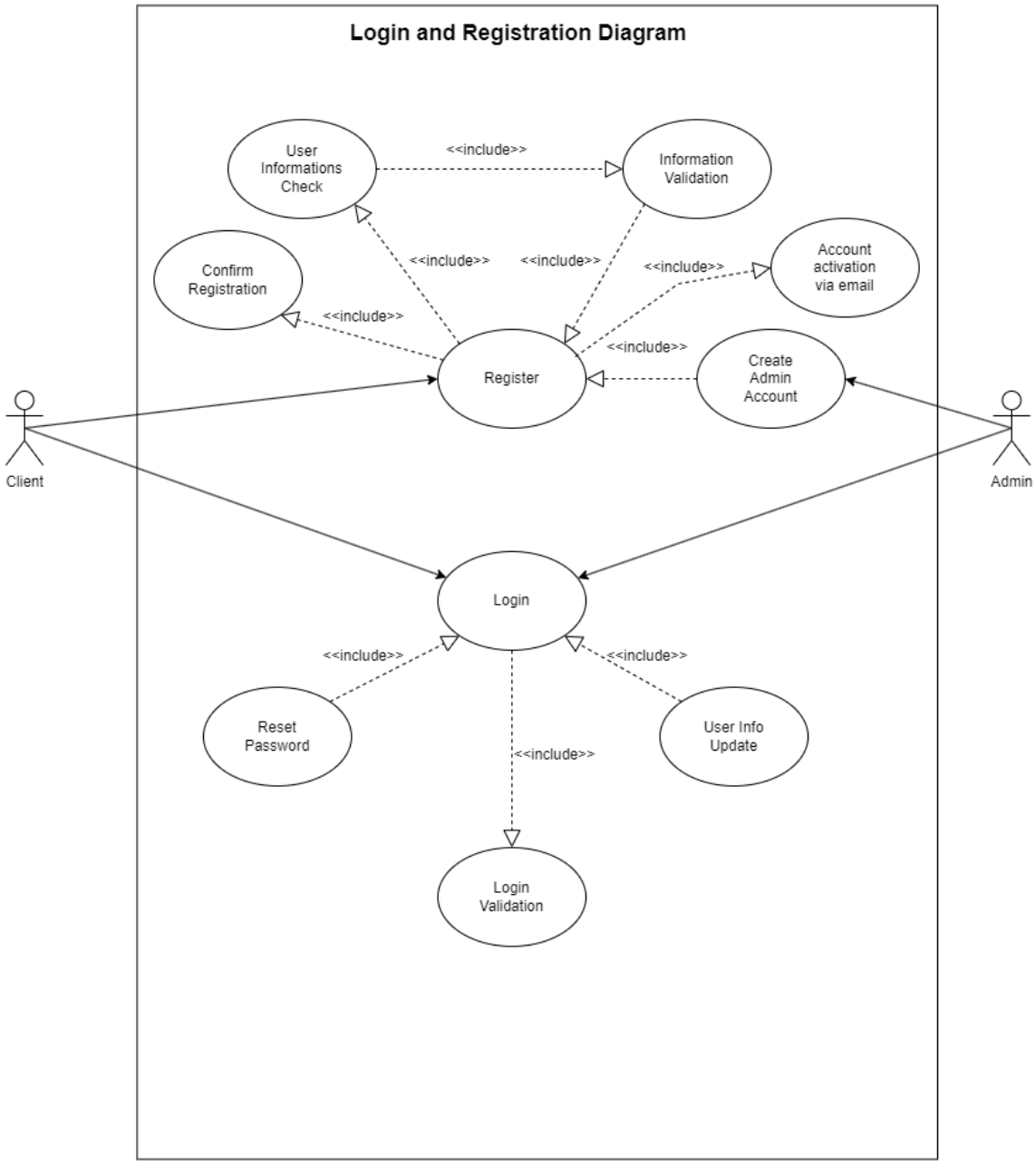


1. - Flowchart Diagram

The flowchart illustrates the customer journey in the online shopping platform. It starts with the customer deciding to shop or access their account. If accessing an account, they must login or register. Upon logging in, the customer can view or edit their account details, or proceed to shop. In the shopping process, the customer can search for items, add them to their cart, and review the cart's contents. They can update item quantities or continue shopping before proceeding to checkout.

At checkout, the customer finalizes their order details and moves to payment, which is processed via Stripe integration. If the payment is approved, the order is confirmed, and a game code is delivered, concluding the process.

Throughout the process, there are points where the customer can choose to end the session or continue shopping, and the flowchart accommodates updates to the shopping cart and user account details. The flowchart ends once the game code is successfully delivered or if the customer opts to exit the process.

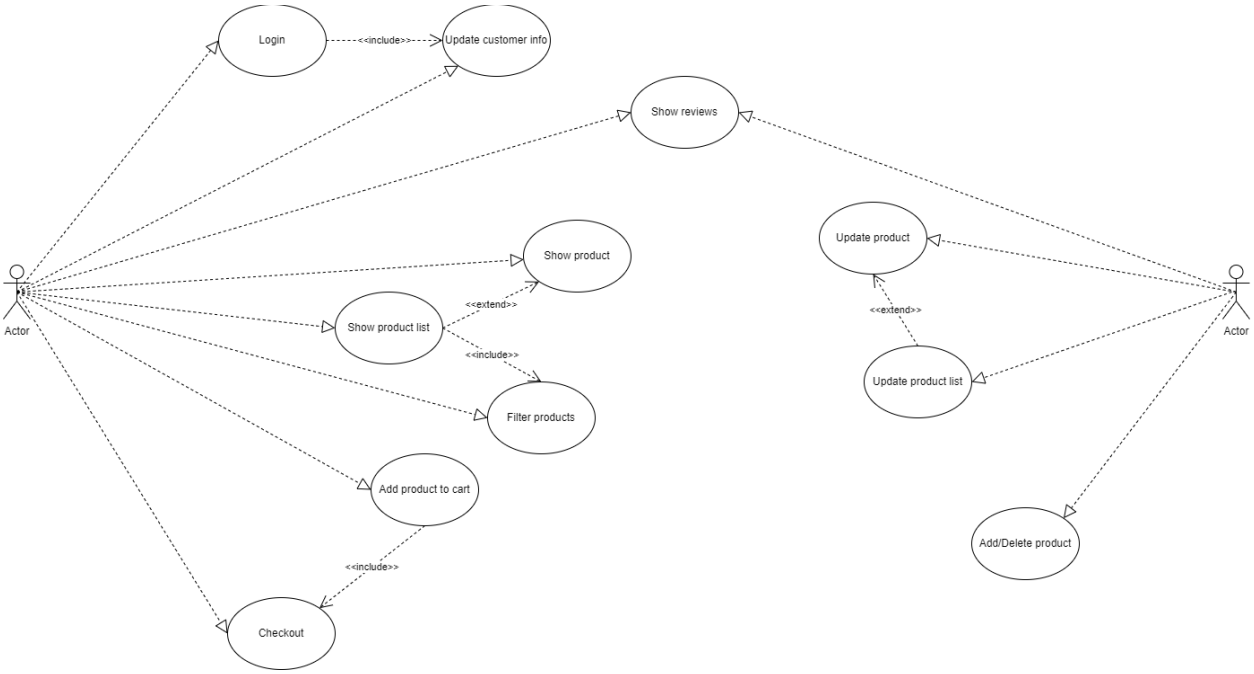


2. – Use Case Diagram for Login process

This diagram encapsulates the processes of user registration and login within the application. Registration is a multi-step procedure that begins with the client providing information, which is then checked and validated. An important part of this process is the account activation, typically confirmed via an email link. For administrative access, there is a parallel track allowing the creation of an admin account, which presumably requires additional validations.

Once registration is complete, the diagram converges on the login process, which is a critical gateway for both regular users and admins. The login procedure is safeguarded by a validation step to authenticate the user credentials. If a user forgets their password, there is a provision to reset it, ensuring they can regain access to their account.

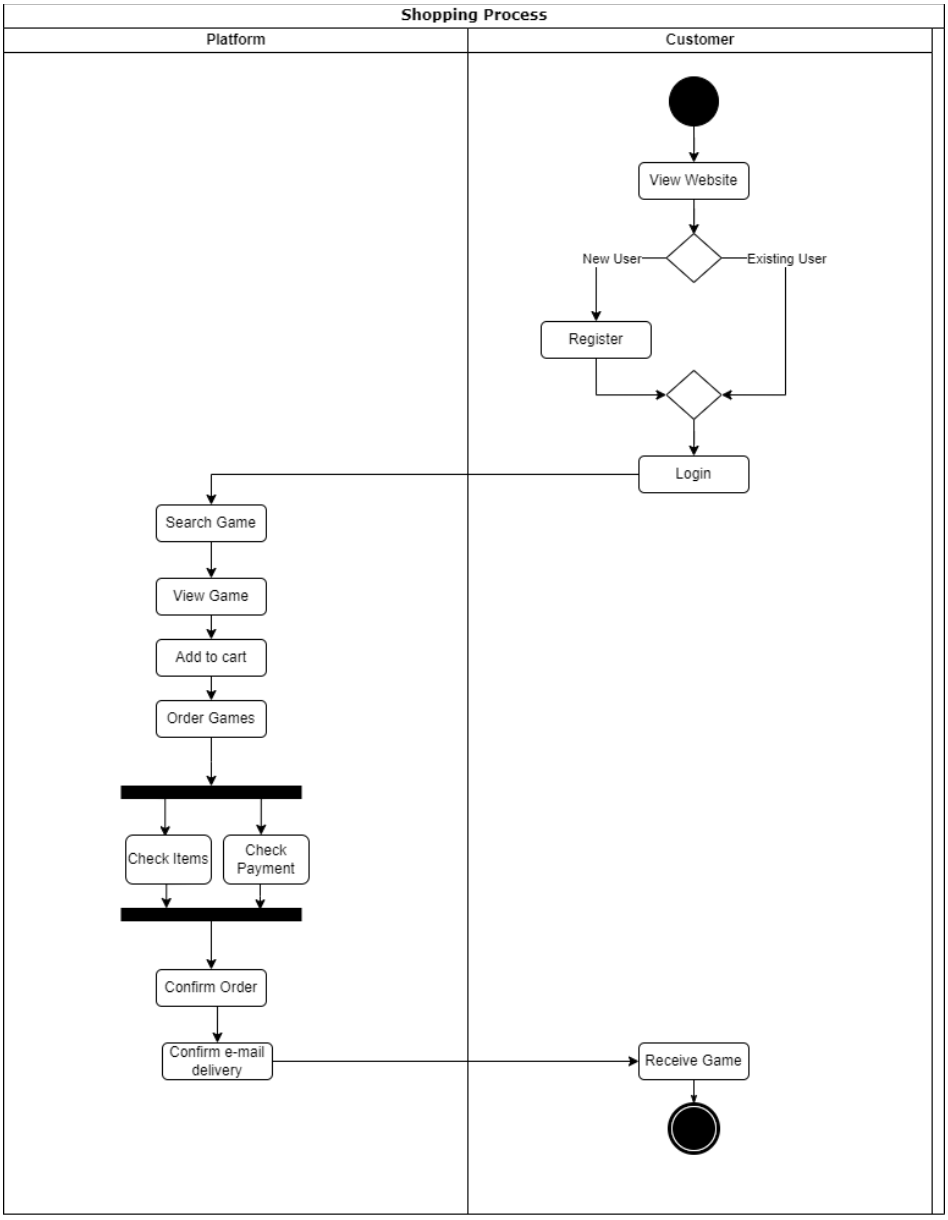
After successful login validation, users are presented with the functionality to update their personal information, keeping their profiles current.



3. – Use Case Diagram for Shopping process

This diagram delineates the interactions between users and the functionalities provided by the shopping platform application. The primary actor is the customer, who engages in various activities such as viewing products, updating information, and making purchases. Upon login, the customer can navigate through a list of products. This includes viewing individual product details, filtering products to refine their search, and checking reviews left by other users. In parallel, the customer has the capability to update their personal information at any point after logging in, ensuring their profile remains accurate. When the customer selects a product, they can add it to their cart. After shopping, they proceed to checkout, completing the purchase process.

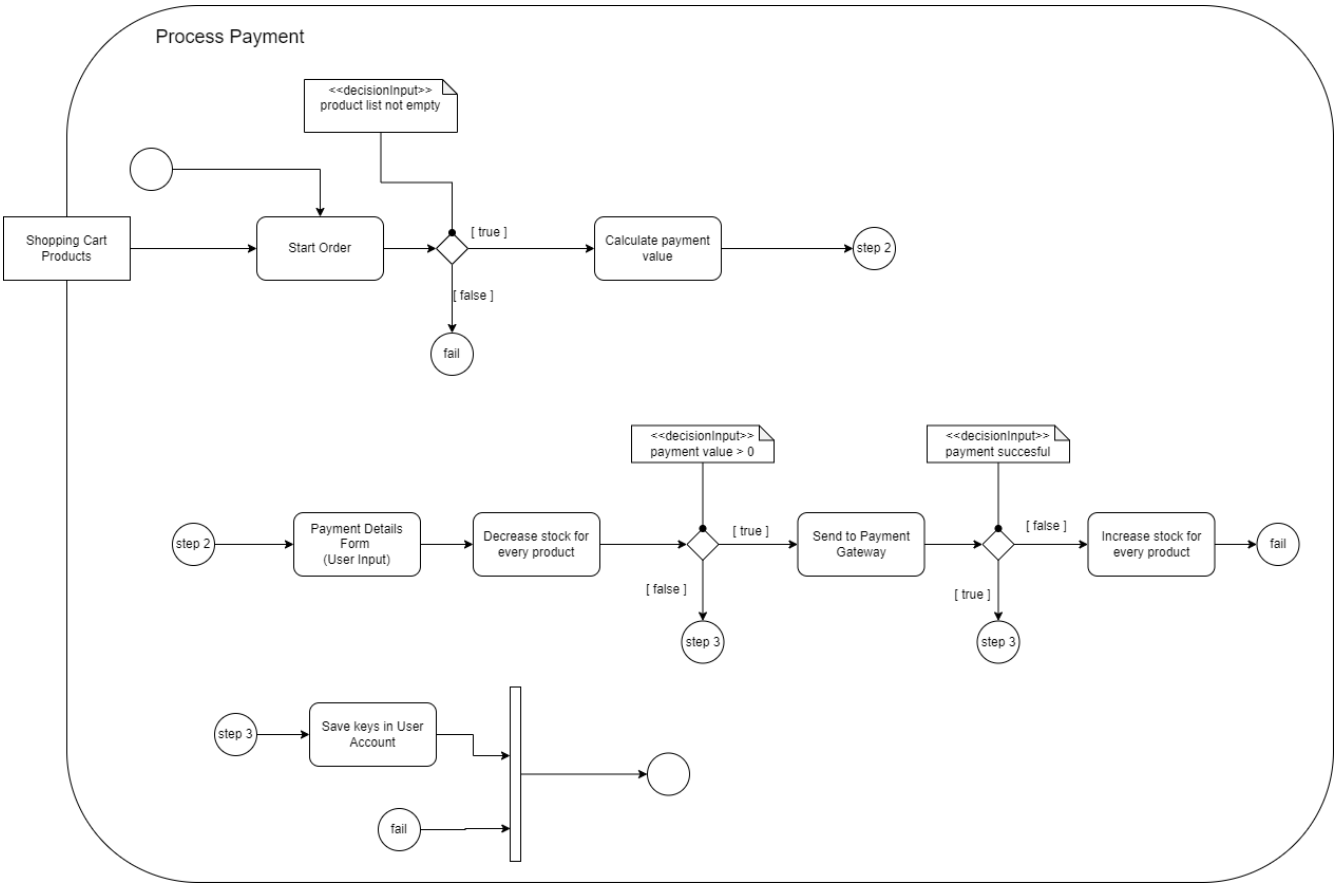
For an administrative actor, there are additional use cases related to inventory management. An admin can update individual product details or the entire product list, reflecting changes such as price adjustments, stock levels, or product descriptions. The admin also has the power to add or delete products, thereby managing the product offerings available to customers. This use case diagram is essential for understanding the roles and capabilities within the shopping platform, highlighting the system's versatility in accommodating both customer needs and administrative tasks. It outlines a clear and comprehensive structure for the fundamental operations associated with e-commerce activities, offering a snapshot of the system's functionality from both the customer's and administrator's perspectives.



4. – Activity Diagram for Shopping Process

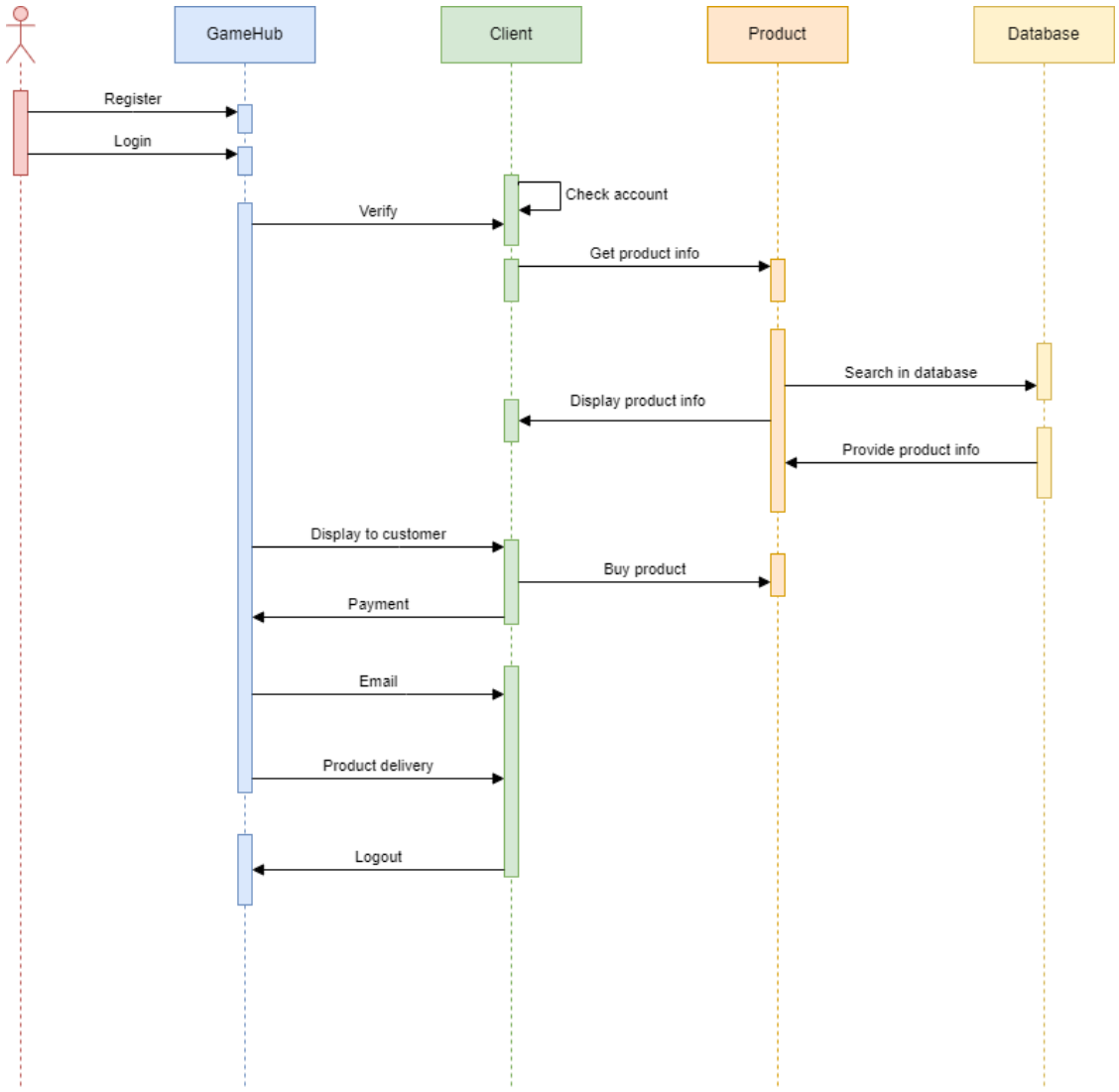
This swimlane activity diagram outlines the comprehensive process a customer follows when purchasing games from the platform. The workflow is bifurcated between the customer's actions

and the platform's processes to facilitate the transaction. Initially, the customer either registers or logs in, which is a crucial step to access the platform's services. Following this, the customer searches for and selects games, adding them to the cart for purchase. The platform then takes over to verify the items in the cart and process the payment. Upon successful payment confirmation, the platform finalizes the order and sends an email to the customer to confirm the transaction. The last step for the customer is the receipt of the game, marking the completion of the purchase.



5. – Activity Diagram for Order Placement

This diagram presents the entire process of purchasing digitals keys from the application. The order can fail at many points, and is highlighted by a node with the name 'fail' (this is only done for readability purposes). It takes the items in the cart and calculates the total cost. After that, it’s pretty straight forward. The user has to complete a form for payment, stock for each key is decreased to ensure that we can provide the keys, and if everything goes right, the payment is successful, and the user receives the keys in his account. If at any point it fails, the available stock will be increased for each product and the keys will not be delivered.

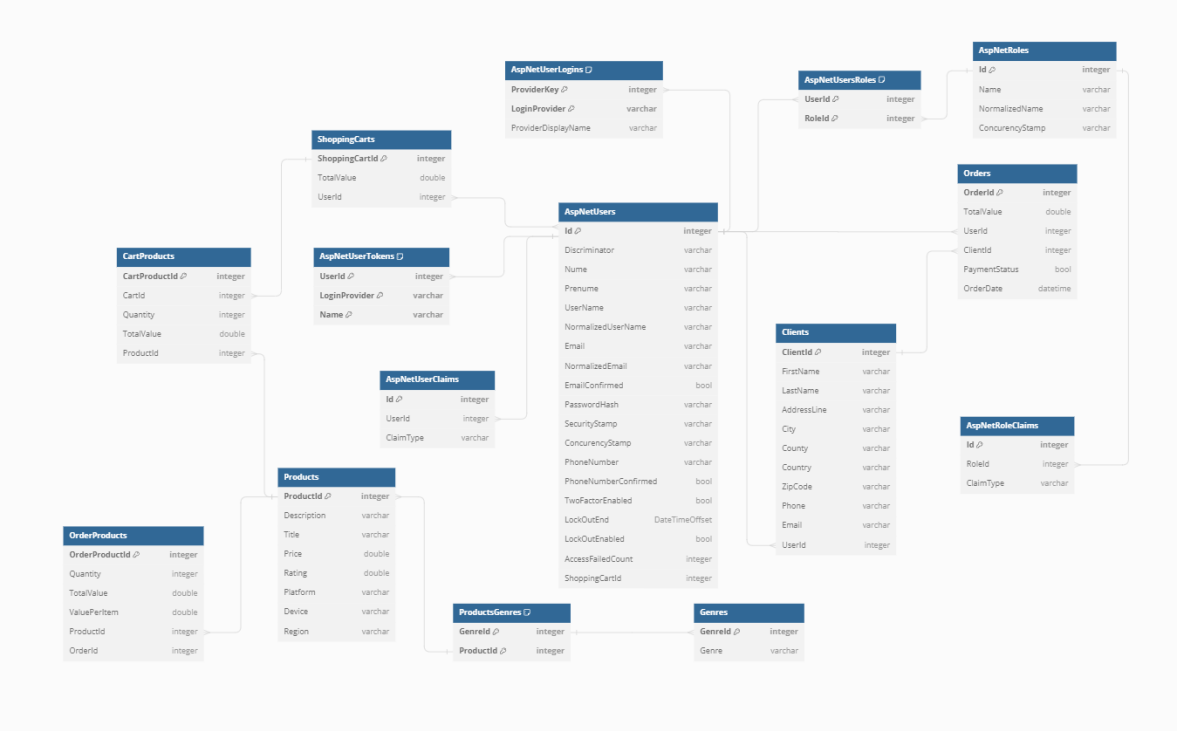


6. – Sequence Diagram

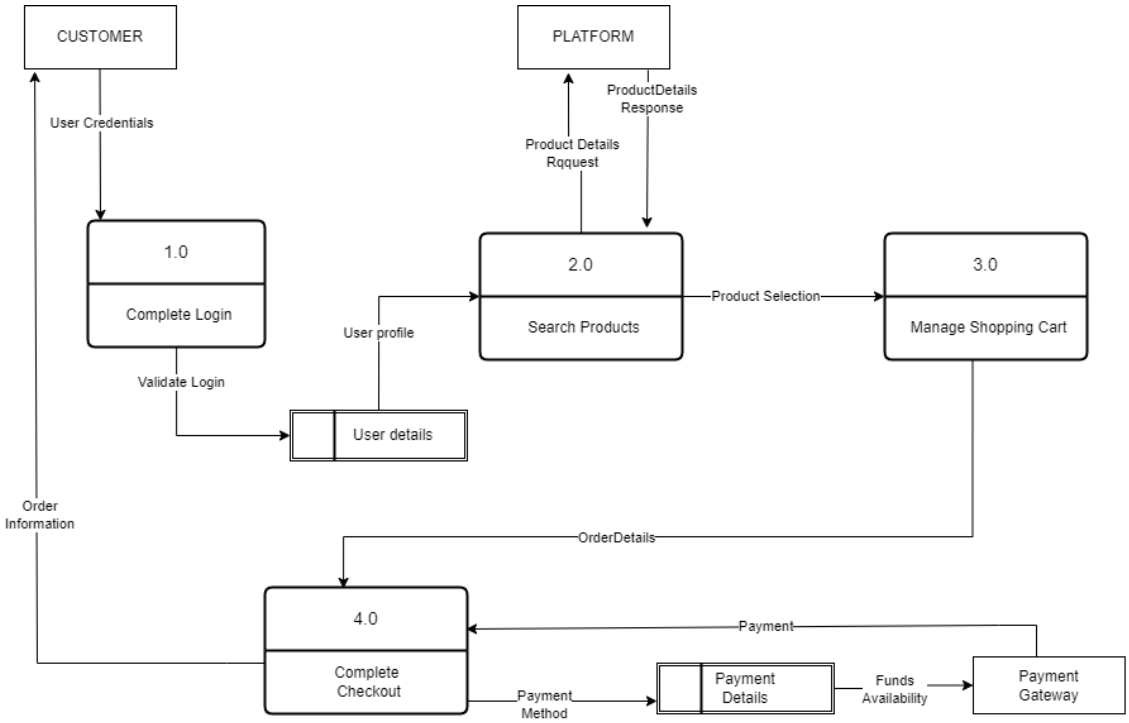
This sequence diagram depicts the structured process through which a customer acquires a digital game, via the GameHub platform. The interaction begins with the customer engaging in registration and login procedures, establishing their credentials on the platform. Following successful login, the customer's account is verified, setting the stage for a secure transaction. The customer then requests information on a product, prompting the GameHub system to retrieve data from the database. This information is displayed back to the customer, who can then decide to proceed with the purchase. The act of buying triggers a sequence of events: payment processing, confirmation via email, and eventual product delivery.

If all steps are successful, the stock is adjusted to reflect the purchase, and the customer receives the product, completing the transaction. This ensures that the inventory remains up-to-date, and the customer is satisfied with a smooth purchase experience on the GameHub platform. The sequence diagram provides a clear visualization of the step-by-step interactions necessary for the effective functioning of an e-commerce system.

5. Data Management



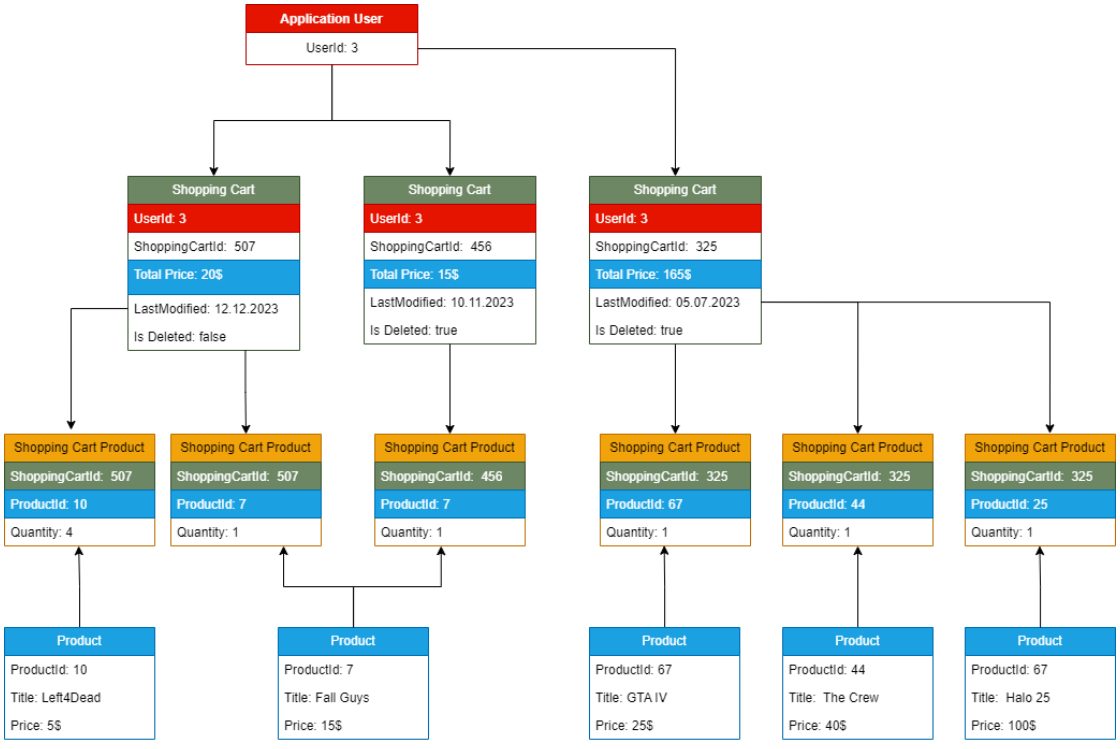
7. - Database Schema



8. – Data Flow Diagram

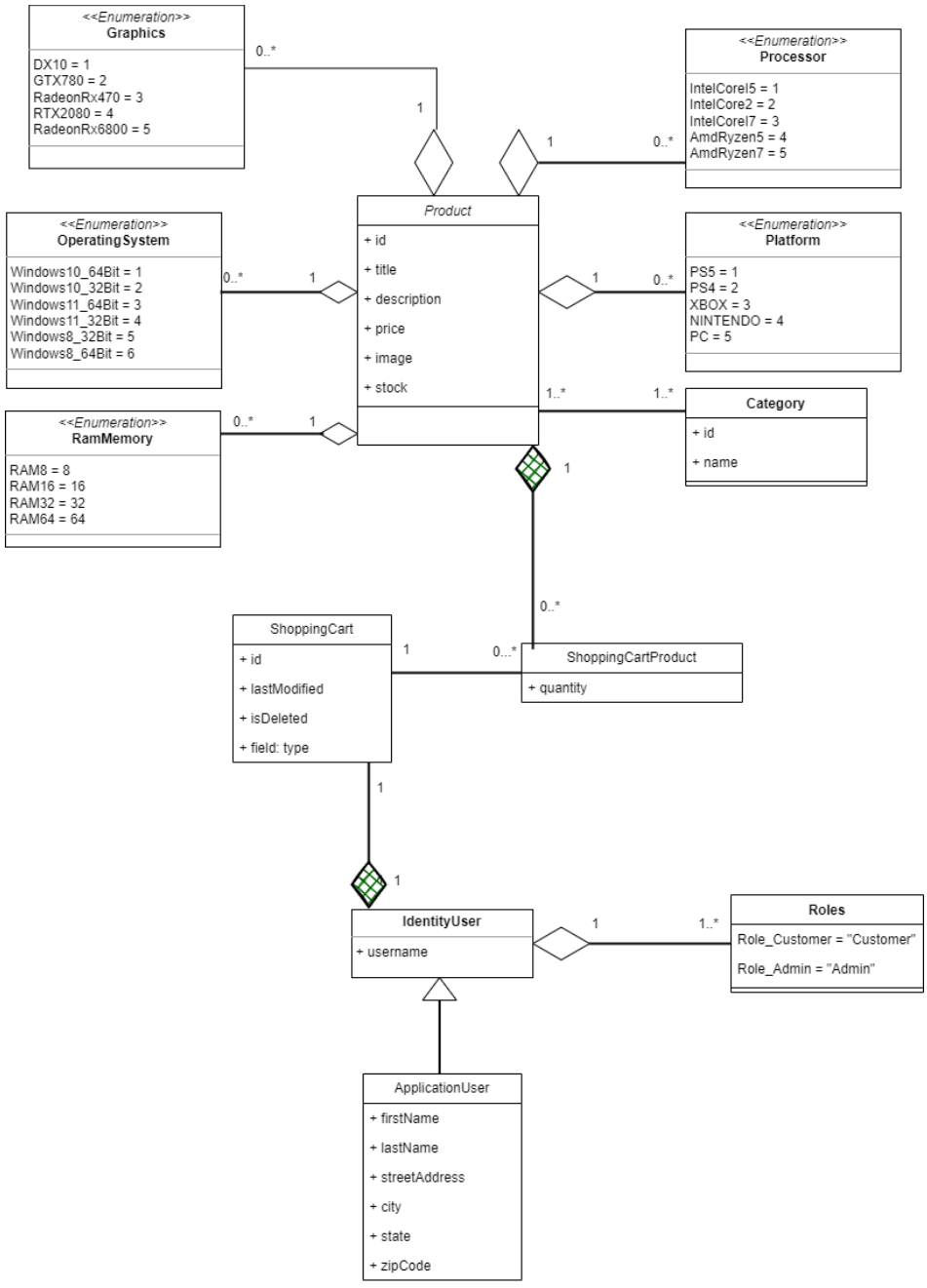
Data Flow Diagram helps us explain the flow of information of our application. The diagram of the application consists of the presence of 3 External Entities that represent outside objects that interact with the system, these are sources and destinations for the information that enters/leaves the system), in this case 'CUSTOMER' -> can send credential authentication or can receive order information , 'PLATFORM'-> provides all product information for the user to view them, 'PAYMENT GATEWAY'-> check if the current user card has funds, in which case the transaction is carried out and sends the information further.

There are present 4 Processes (type of component that represents an activity that changes the date received by creating an output): 'Complete Login' -> based on the credentials sent by the user, it validates them and in a positive case, sends the details of the account, 'Search Products' -> activity that involves searching, visualizing the details about products and initiating the product addition to the cart, 'Manage Shopping Cart' -> receives the products added by the user and initiates the order placement, 'Complete Checkout' -> receive details about the products that are in the course of purchase, it initiates the payment, receives information about the transaction and sends the confirmation information and the game code to the user. We also have 2 Data Stores (files/repos that keep the information to be used later), 'User details' -> after the user has been successfully authenticated, its data is retained to be used whenever needed throughout the duration of the session, 'Payment Details' -> card information, introduced and validated, prepared at any time to initiate the payment. Also, several Data Flows which represents the information that moves between the mentioned components.



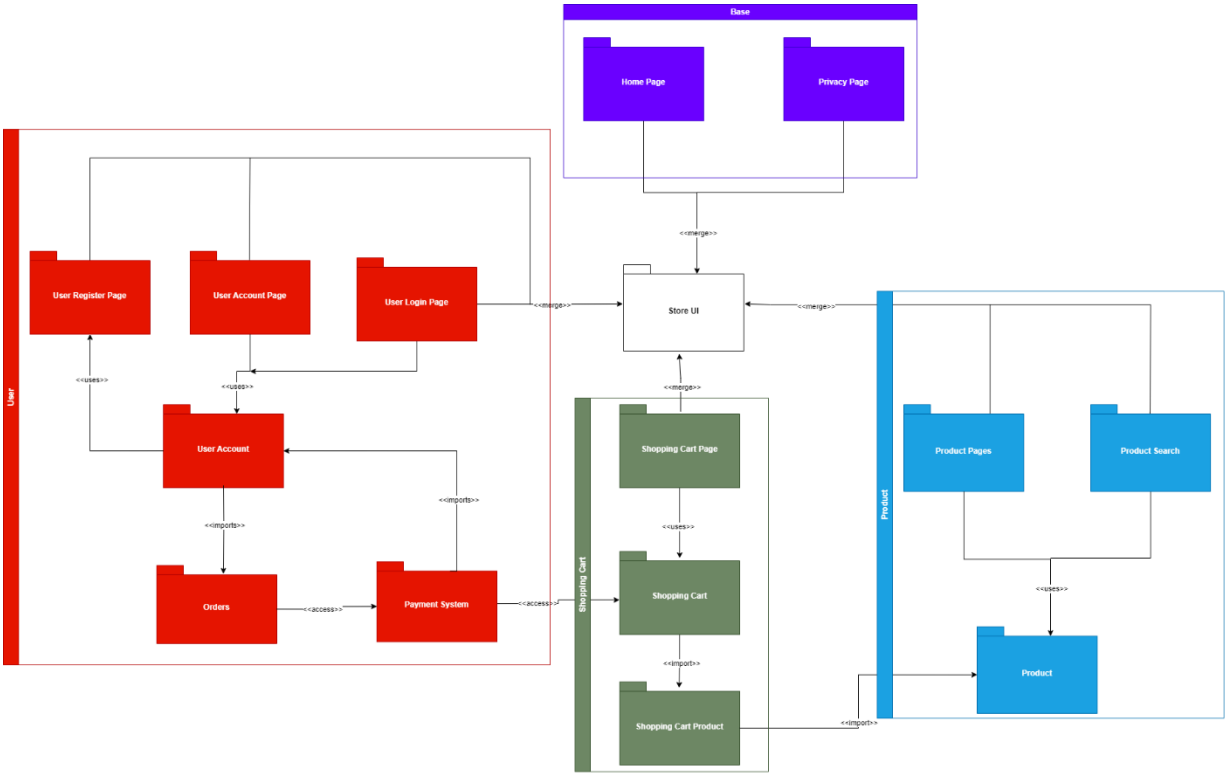
9. – Object Diagram

This diagram is an object diagram showing the state of a user's shopping cart in an online video game store. It details one user with several shopping cart sessions, indicating past and current purchases with specific games and their quantities. Each cart shows a total price, modification date, and whether it's been deleted. Products are linked to the carts, demonstrating which games are being bought. The diagram captures a snapshot of the system's state, particularly the user's interactions with their shopping carts and the products within them at a given time.



10. – Class Diagram

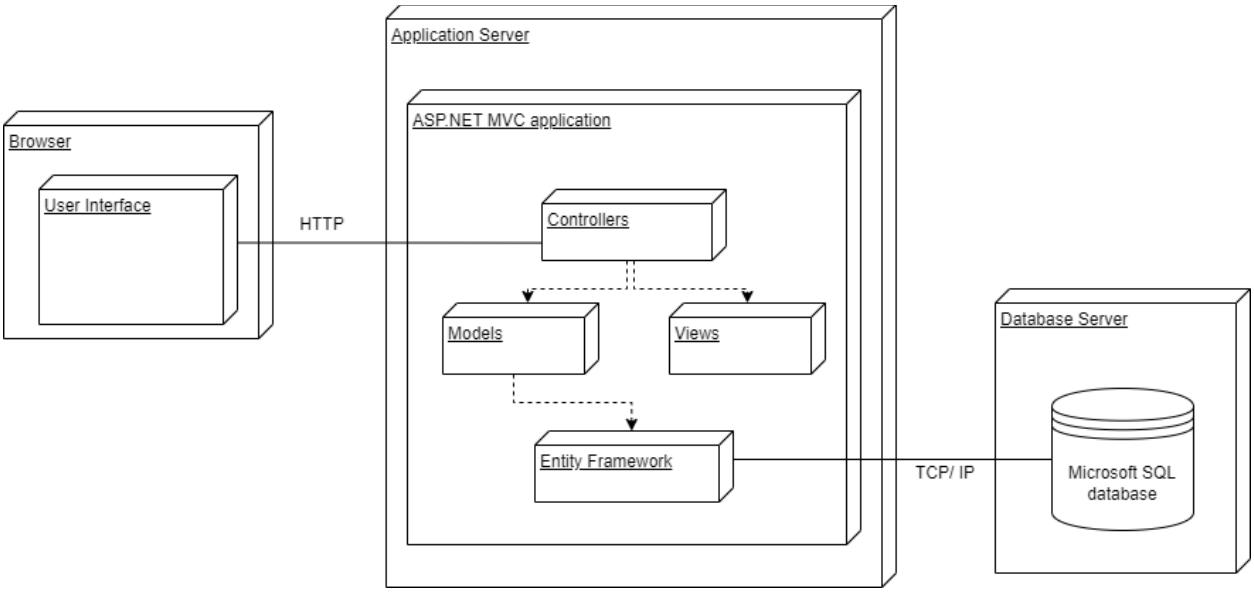
Class Diagram is a graphic repository to build and visualize the object oriented system. The diagram illustrates the links between classes. All classes are related by association (mentioning cardinality). The 'Product' class has an aggregation relationship with all ennums (Platform, Processor, Graphics, OperatingSystem and RamMemrory), which means that all ennums are part of 'Product',they can be associated with 'Product' and have a separate life span. The aggregation relationship is also found between 'IdentityUser' and 'Roles'. Composition relationships are present, between 'Products' and 'ShoppingCartProduct' and between 'IdentityUser' and 'ShoppingCart'. The composition relationship has the effect that if 'IdentityUser' is destroyed then 'ShoppingCart' is lost too. 'ShoppingCart' cannot stand by itself.
The other specific relationship OOP is the inheritance, which we find in the relationship between 'IdentityUser' and 'ApplicationUser'. 'ApplicationUser' is a subclass that inherits what 'IdentityUser' has.



11. –Package Diagram

The package diagram is divided into several sections, each representing different aspects of an e-commerce system. The red section deals with user-related functionality, with packages for the user registration page, user account page, and user login page. These packages are related to the user account, which in turn has connections to order processing and the payment system. The green section represents the shopping cart aspect of the system, with packages for the shopping cart page and the individual shopping cart products. The blue section includes product-related functionality, with packages for product pages and product search. These are directly related to the specific products available in the system. The purple section represents the base or common functionality shared across the system, including the home page and privacy page.

Arrows with labels such as <<uses>>, <<merges>>, and <<access>> show the dependencies and relationships between different packages, indicating how they interact or rely on one another within the system.



12.–Deployment Diagram

This diagram illustrates the physical deployment of our application. It's divided into 2 main components, the application server, where we host the ASP.NET MVC app, and the database server, where we host our Microsoft SQL database. The relations are fairly simple. The client sends http requests to our app server, requests that are processed by the controllers. Every database manipulation request is only sent by our application to the database server.

6. System Overview

- **Architecture:** The platform will use a Model-View-Controller (MVC) architecture.
- **Technologies:** Backend developed in MVC.NET, frontend using a combination of HTML, CSS, and JavaScript (React), and SQL Server for database management.

Design Patterns

Unit of work Pattern

This pattern is essential for complex systems where numerous entities are involved in transactions, ensuring that the system's integrity is maintained. By dealing with changes to the data in a cohesive manner, the Unit of Work ensures that all changes are either committed together or rolled back together, preventing the database from ever being left in an inconsistent state.

For our project, here's a more detailed look at the integration and benefits of using unit of work:

Integration:

- Repositories: Each domain entity (e.g., Category, Order, Product, Platform, etc.) has a corresponding repository file (e.g., CategoryCRUD.cs, OrderCRUD.cs, etc.). These repositories abstract the data access logic and business logic for that entity.
- UnitOfWork Class: The UnitOfWork.cs class is responsible for maintaining a list of objects affected by a business transaction and coordinating the writing out of changes and resolving of concurrency problems.
- UnitOfWork Interface: The IUnitOfWork.cs interface defines the contract for the UnitOfWork class, making the code more modular and testable.
- Domain Logic: Business logic can be executed across multiple repositories within a single transaction managed by the UnitOfWork.

Benefits:

- Consolidated Data Operations: The Unit of Work maintains a list of changes (insertions, updates, deletions) during a business transaction and commits them together to avoid partial updates.
- Transaction Management: It simplifies transaction management by providing a single point to commit or rollback changes.
- Concurrency Control: Helps manage concurrency where multiple transactions are trying to modify the same data.
- Improved Performance: By batching operations, it reduces the number of database calls, which can significantly improve performance.
- Cleaner Code: It leads to cleaner code by reducing boilerplate for transaction management and data consistency logic in the business layer.

In an e-commerce system just like ours, an order process might involve creating an order, updating inventory, and updating user account details. With the Unit of Work, these operations can be performed in a single transaction. This way, if any step fails, the entire transaction is rolled back, preventing scenarios where the inventory is updated but the order isn't created due to an error.

In summary, the Unit of Work pattern is an essential part of ensuring data consistency and integrity in applications that deal with complex transactions involving multiple steps and entities. It is a core component of a well-architected domain-driven design and is particularly valuable in systems where business transactions are complex and involve multiple data mutations.

The Service Layer Pattern

The Service Layer pattern is an architectural pattern used in software design to abstract business logic from the presentation layer and data access code. It provides a set of service operations that implement the business logic of the application, centralizing business logic in a way that can be reused and tested independently from the user interface.

This is how the Service Layer pattern is implemented in our case:

- Abstraction: The ServiceShoppingCart class implements the IServiceShoppingCart interface, which is a typical approach in the Service Layer pattern to define a contract for

the operations that the service will provide. This allows for loose coupling between the service layer and its consumers, making it easier to replace or modify the service layer implementation without affecting the rest of the application.

- Encapsulation of Business Logic: The class ServiceShoppingCart encapsulates the business logic for operations related to shopping carts, such as adding a product to a cart (AddProduct), removing a product (DeleteProduct), and preparing for checkout (PrepareCheckout). This keeps the business logic separate from the data access code (handled by the repositories) and the presentation logic (handled by controllers or UI components).

- Use of Repositories: The service interacts with the application's repositories through the IUnitOfWork interface. This indicates that the service layer is responsible for orchestrating calls to the data layer and does not directly interact with the database. This is a common responsibility of a service layer, which works with data access objects but does not contain data access logic itself.

- Transaction Management: The service methods manage the transactions by explicitly calling _unitOfWork.Save() to commit changes to the database. This indicates that the service layer is in control of the transaction boundaries, which is a key responsibility in the Service Layer pattern.

- Exception Handling: The service methods include try-catch blocks for handling exceptions, which encapsulates the error handling within the service layer. This allows for centralized exception handling and can also be a part of ensuring that the system remains in a consistent state by rolling back transactions if an error occurs.

- DTO Usage: The service layer uses Data Transfer Objects (DTOs) like ShoppingCartDto to pass data between the layers. This ensures that the presentation layer does not need to deal with the domain models directly, but rather with a simplified and potentially flattened view of the data that is tailored to the needs of the presentation layer.

- Domain Logic Coordination: The service layer coordinates complex business logic that involves multiple types of domain entities. For example, the AddProduct method involves operations on both the ShoppingCart and Product entities, as well as potentially creating a new ShoppingCartProduct.

- Service Operations: The operations provided by the service are focused on higher-level functionality that a client, such as a web application, might need. For example, Get, AddProduct, DeleteProduct, IncreaseQuantity, DecreaseQuantity, PrepareCheckout, CompleteCheckout, and PostProcessOrder are all operations that a client might directly invoke.

In summary, the Service Layer pattern in this code is implemented as a class that serves as an intermediary between the presentation layer and the data access layer. It provides business logic and operations related to shopping carts, handling transactions, and orchestrating the flow of data

between the application's components. This pattern promotes a clean separation of concerns and helps to keep the business logic organized and centralized.

7. Integration Requirements

- **Payment Gateway Integration:**
  - Integration with payment services-Stripe.
- **Email Service Integration:**
  - For user verification, notifications, and marketing.

Equal Contribution of 20% for each member of the team.