VIETNAM NATIONAL UNIVERSITY OF HOCHIMINH CITY
THE INTERNATIONAL UNIVERSITY
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

WEB APPLICATION DEVELOPMENT
IT093IU

FINAL REPORT

Topic: Game Shopping System

By Group 8386 PTPL – Member List

No	Name	ID
1	Đặng Đăng Khôi	ITCSIU22266
2	Lê Hoài Bảo	ITCSIU22259
3	Võ Hoàng Hiệp	ITCSIU22261

Instructor: Assoc. Prof. Nguyen Van Sinh
Lab instructor: MSc. Nguyen Trung Nghia

ACKNOWLEDGMENTS

We would like to express our deepest appreciation to **Assoc. Prof. Nguyen Van Sinh**, our instructor for theory classes, whose lectures and guidance provided the foundational knowledge necessary for this project. His expertise and feedback were invaluable in understanding the theoretical aspects of the system.

We are equally grateful to **MSc. Nguyen Trung Nghia**, our lab instructor, for his continuous support during the practical implementation phase. His technical assistance, code reviews, and problem-solving guidance helped us overcome numerous challenges and complete the development successfully.

We acknowledge the time and effort of both instructors dedicated to helping us throughout this project, and we are truly grateful for their mentorship.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This project implements a Game Shopping E-commerce Platform using a full-stack architecture: React.js frontend, Node.js/Express.js backend, and PostgreSQL database. The system provides 63 RESTful API endpoints organized into 11 functional modules: Authentication, Admin Management, User Profile, Reference Data, Games, Cart, Wishlist, Library, Orders, Payments, and Health Check.

The platform implements session-based authentication with email verification, supports advanced game search with full-text indexing, provides personalized recommendations based on user preferences, and includes a complete e-commerce workflow from cart to checkout and payment processing. The admin dashboard offers comprehensive management capabilities for games, orders, users, payments, and reviews.

Key technical implementations include RESTful API design following standard conventions, middleware-based authentication and authorization, database query optimization, error handling with standardized response formats, and responsive frontend design using React Context API for state management. The system demonstrates practical application of modern web development technologies and e-commerce platform design principles.

CHAPTER 1

INTRODUCTION

1.1. Background

Digital game distribution has grown with the shift to online platforms. E-commerce platforms for digital games need to provide browsing, purchasing, and management features. Users expect personalized recommendations, efficient search, secure transactions, and easy access to their game libraries. Administrators require tools to manage inventory, process orders, monitor payments, and maintain the platform.

Modern web technologies enable building scalable, responsive platforms that meet these needs. A well-designed e-commerce platform can improve user experience, streamline operations, and support business growth.

1.2. Problem Statement

Existing game e-commerce platforms often lack:

- Personalized recommendations based on user preferences
- Efficient search and filtering
- Seamless cart and checkout flows
- Integrated wishlist and library management
- Admin tools for managing games, orders, and users
- Secure authentication and payment processing

These gaps can reduce user satisfaction, complicate administration, and limit scalability. There is a need for a platform that addresses these areas while maintaining performance and usability.

1.3. Scope and Objectives

This project develops a Game Shopping E-commerce Platform with the following objectives:

Objectives:

1. Design and implement a full-stack web application with React.js frontend, Node.js/Express.js backend, and PostgreSQL database
2. Develop a RESTful API with 63 endpoints covering authentication, game management, cart, wishlist, orders, payments, and administration
3. Implement user authentication with email verification and session management

4. Create a game browsing system with search, filtering, and personalized recommendations
5. Build shopping cart and checkout workflows with payment integration
6. Develop wishlist and library management for users
7. Design an admin dashboard for managing games, orders, users, payments, and reviews
8. Implement a review and rating system for games
9. Ensure responsive design and user experience across devices
10. Demonstrate practical application of modern web development technologies and e-commerce principles

Scopes:

- User-facing features: registration, authentication, game browsing, cart, wishlist, orders, library, reviews
- Administrative features: game CRUD, order management, user administration, payment monitoring, review moderation
- Technical scope: RESTful API design, database schema design, frontend state management, authentication and authorization, payment processing integration

1.4. Customer Requirements

The platform must meet the following requirements:

User Requirements:

- Easy registration and secure login with email verification
- Intuitive game browsing with search, filters, and detailed game pages
- Personalized recommendations based on preferences
- Shopping cart with add/remove and checkout
- Wishlist to save games for later
- Order history and tracking
- Library to access purchased games
- Ability to review and rate owned games
- Profile management with preferences and billing addresses

Administrator Requirements:

- Dashboard with platform statistics and analytics
- Game management: add, edit, update prices and discounts

- Order management: view, track, and update order statuses
- User management: view users and their activities
- Payment monitoring: track transactions and manage payment statuses
- Review moderation: view and respond to user reviews
- System management: maintain platform data and configurations

Technical Requirements:

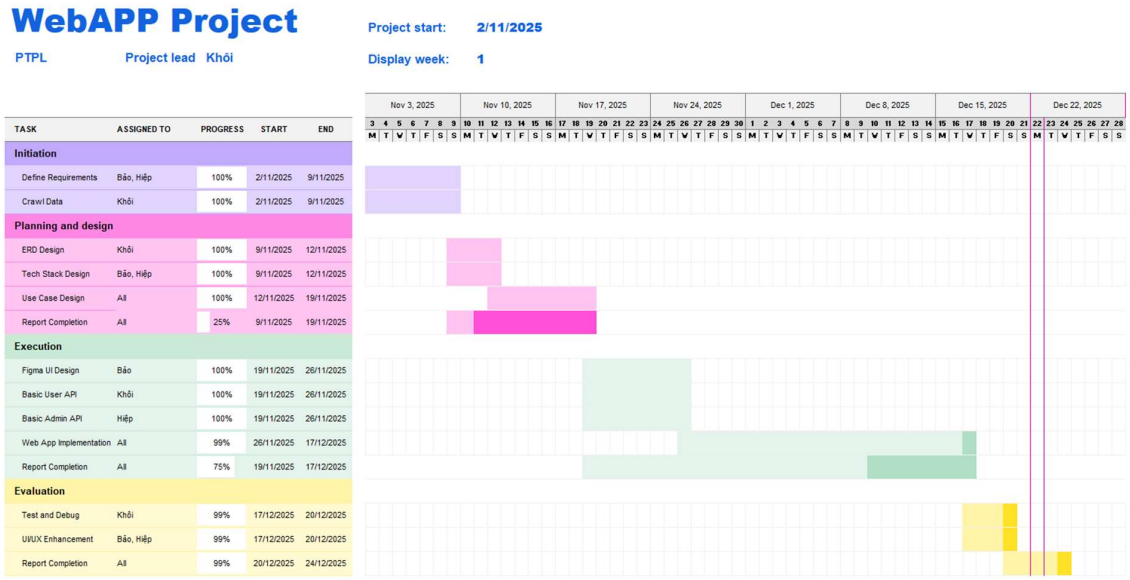
- Responsive design for desktop and mobile
- Secure authentication and authorization
- RESTful API with standardized error handling
- Database optimization for performance
- Scalable architecture for future growth
- Integration with payment processing systems

CHAPTER 2

PROJECT TIMELINE AND IN-CHARGE TABLE

2.1. Project Timeline

Our project start from November 2/11/2025 to 24/12/2025, within 2 months of implementation.



TASK	ASSIGNED TO	PROGRESS	START	END
Initiation				
Define Requirements	Bảo, Hiệp	100%	2/11/2025	9/11/2025
Crawl Data	Khôi	100%	2/11/2025	9/11/2025
Planning and design				
ERD Design	Khôi	100%	9/11/2025	12/11/2025
Tech Stack Design	Bảo, Hiệp	100%	9/11/2025	12/11/2025
Use Case Design	All	100%	12/11/2025	19/11/2025
Report Completion	All	25%	9/11/2025	19/11/2025
Execution				
Figma UI Design	Bảo	100%	19/11/2025	26/11/2025
Basic User API	Khôi	100%	19/11/2025	26/11/2025
Basic Admin API	Hiệp	100%	19/11/2025	26/11/2025
Web App Implementation	All	99%	26/11/2025	17/12/2025
Report Completion	All	75%	19/11/2025	17/12/2025
Evaluation				
Test and Debug	Khôi	99%	17/12/2025	20/12/2025
UI/UX Enhancement	Bảo, Hiệp	99%	17/12/2025	20/12/2025
Report Completion	All	99%	20/12/2025	24/12/2025

Figure 1: Project Timeline

2.2. In-Charge Table

Below is the In-Charge Table of our project.

Table 1: Individual responsibility and contribution

Name	Responsibility	Contribution
Đặng Đăng Khôi	System Design – Database Design	33.3%
Lê Hoài Bảo	Full Stack Web Design	33.3%
Võ Hoàng Hiệp	Full Stack Web Design	33.3%

CHAPTER 3

METHODOLOGY

3.1. Overview

The Game Shopping E-commerce Platform is a full-stack web application for digital game sales and management. It enables users to browse, purchase, and manage games, and provides administrators with tools to manage the platform. The system uses modern web technologies to deliver a responsive, scalable solution.

3.1.1. User requirement analysis

3.1.2. Functional Requirements

User:

Name	Detail
User Registration	New user can register a new account with their own references and verification code
User Authentication	User can login into the web page with their credentials, cookie and session save
User Profile Management	User can edit their own profile and save
Game Search and View	User can search any games, sort and filter games as their wish
Wishlist	User can add a game to their wishlist
Cart Management	User can add and remove items from the cart
Order Management	User can place order and buy games they want
Review Management	User can add a review for each game they owned and
Session and Cookie Handling	Session and Cookie of user references are saved for future usage.

Admin:

Name	**Detail**
Admin Authorization	Admin is authorized and accessible to admin page.
Admin Dashboard	Admin dashboard present the stats of the webpage
Admin Management	Admin can manage users, orders, payments, add new games and edit games.

3.1.3. Non-Functional Requirements

Name	Detail
Performance	Pages must load within 2 seconds under normal traffic.
Scalability	The system should handle up to 100,000 concurrent users without degradation.
Usability	The UI must be intuitive and provide a seamless navigation experience.
Security	Sensitive user data (e.g., passwords, payment information) must be encrypted.
Compatibility	The platform must work across modern browsers (Chrome, Firefox, Safari) and devices (iOS, Android).
Reliability	The system should have 99.9% uptime, ensuring minimal disruptions to users.
Maintainability	The codebase should follow clean coding practices and be easy to update.

3.3 System Design

3.3.1. Entity Relation Diagram

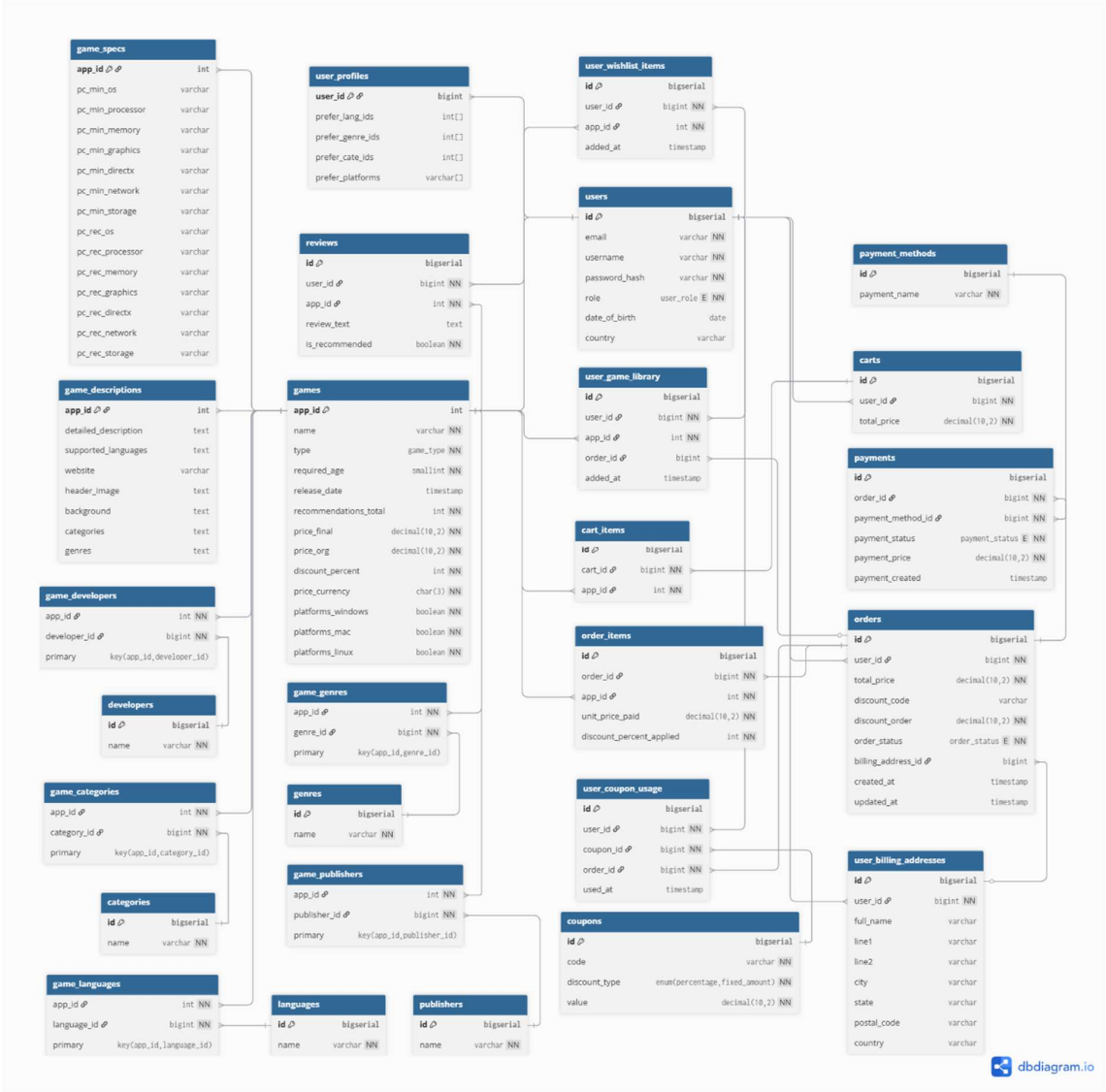


Figure 2: ER Diagram

3.3.2. Use Cases

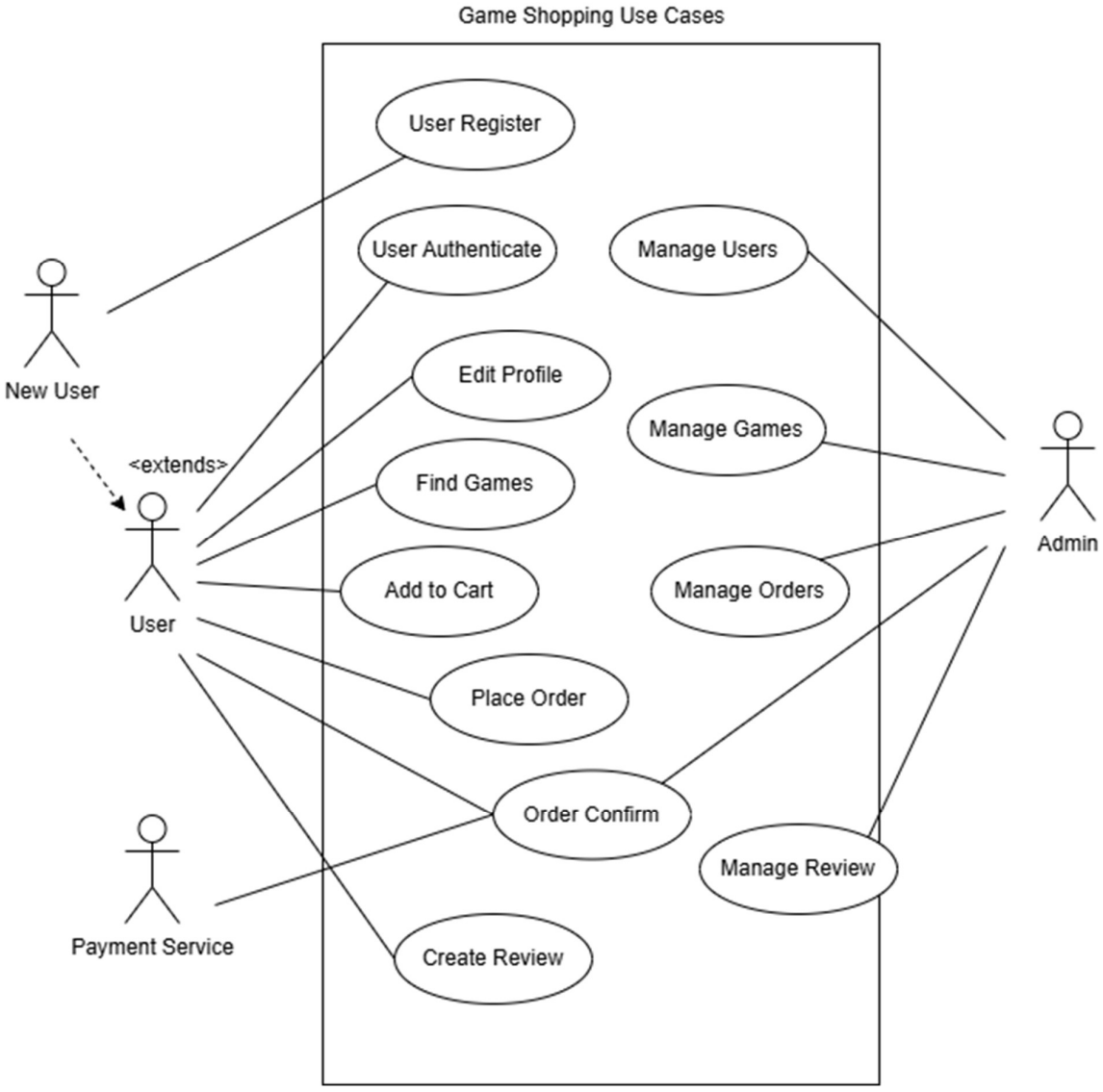


Figure 3: Use Case Diagram

3.1.3.1. Use Case Descriptions

- Actors:
  - New User: User who did not have or create account yet.
  - User: User who already have an account and have their own references.
  - Payment Service: Service provider that handle the payment of user.
  - Admin: User but have role as Admin, manage the system.

1. User Register

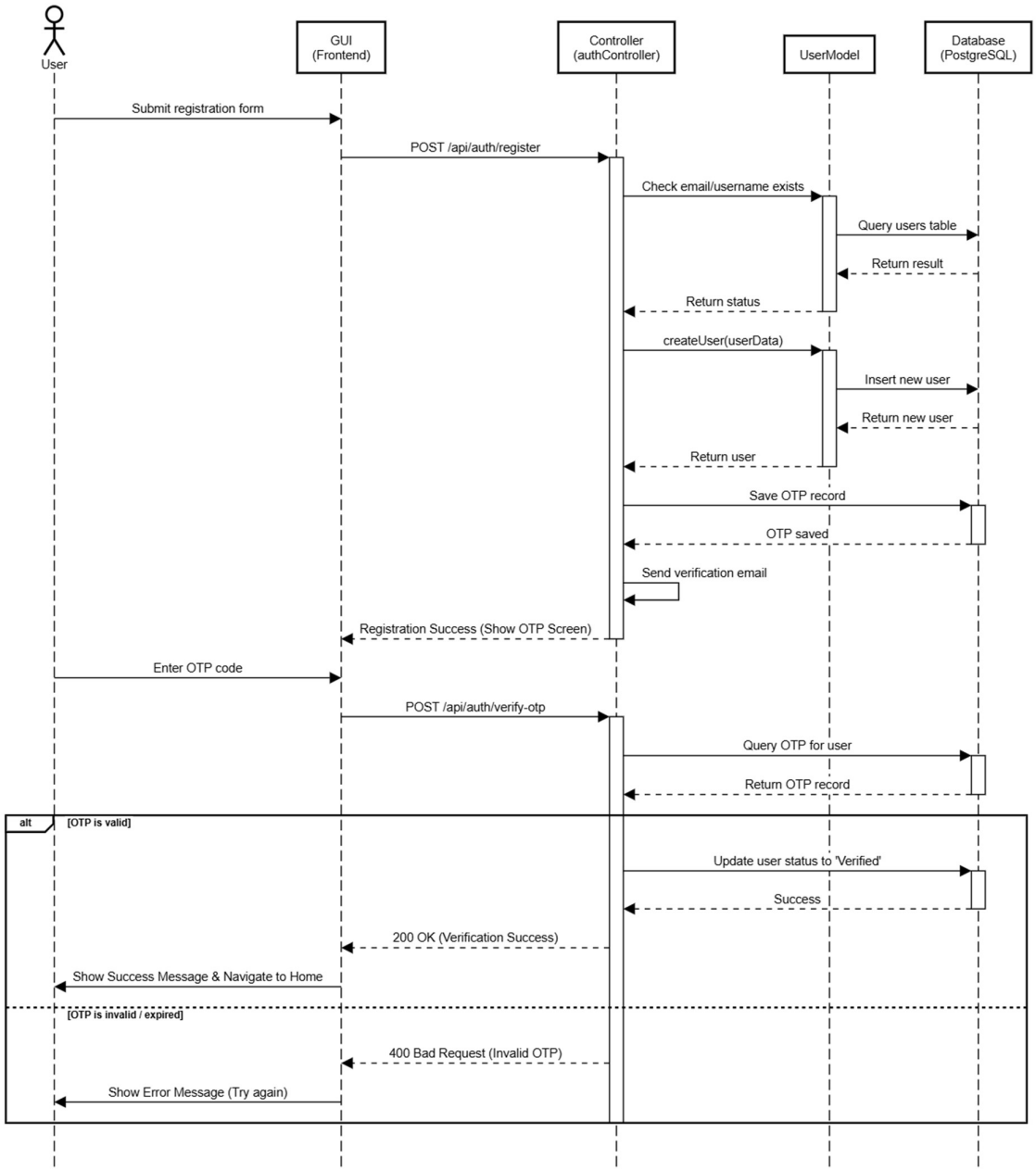


Figure 4: User Registration Sequence Diagram

Description:

- **Input Action**: User submits the registration form and later enters the received OTP.
- **Flow:** The Controller checks for duplicate emails in the Database, creates a new user, and sends a generated OTP via email. Once the user submits the code, the Controller validates it against the database record and updates the user's status to "Verified."
- **Output Action:** Successful: User is verified and navigated to the Home Page. Failure Message: "Email already exists" or "Invalid OTP."

2. User Authenticate

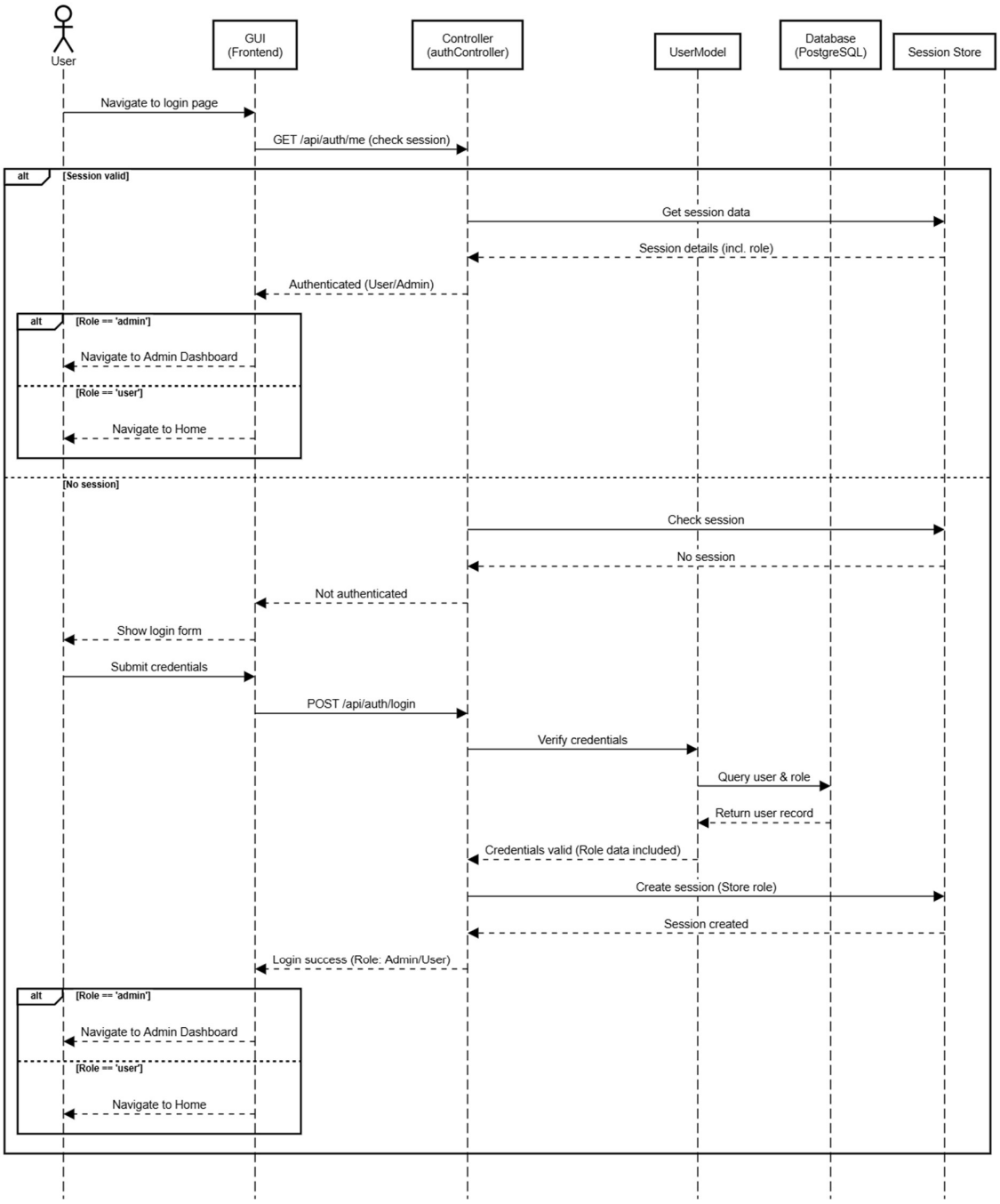


Figure 5: User Authenticate Sequence Diagram

Description:

- **Input Action:** User navigates to the login page or submits their username and password.
- **Flow:** The system first checks for an existing session; if found, it identifies the user's role to determine the destination. If no session exists, the Controller verifies the

submitted credentials against the Database, retrieves the user's role, and creates a new session.

- **Output Action:** Successful: Admins are redirected to the Admin Dashboard (/admin), while Regular Users are sent to the Home Page (/home). Failure Message: "Invalid credentials" or "Session expired."

3. Edit Profile

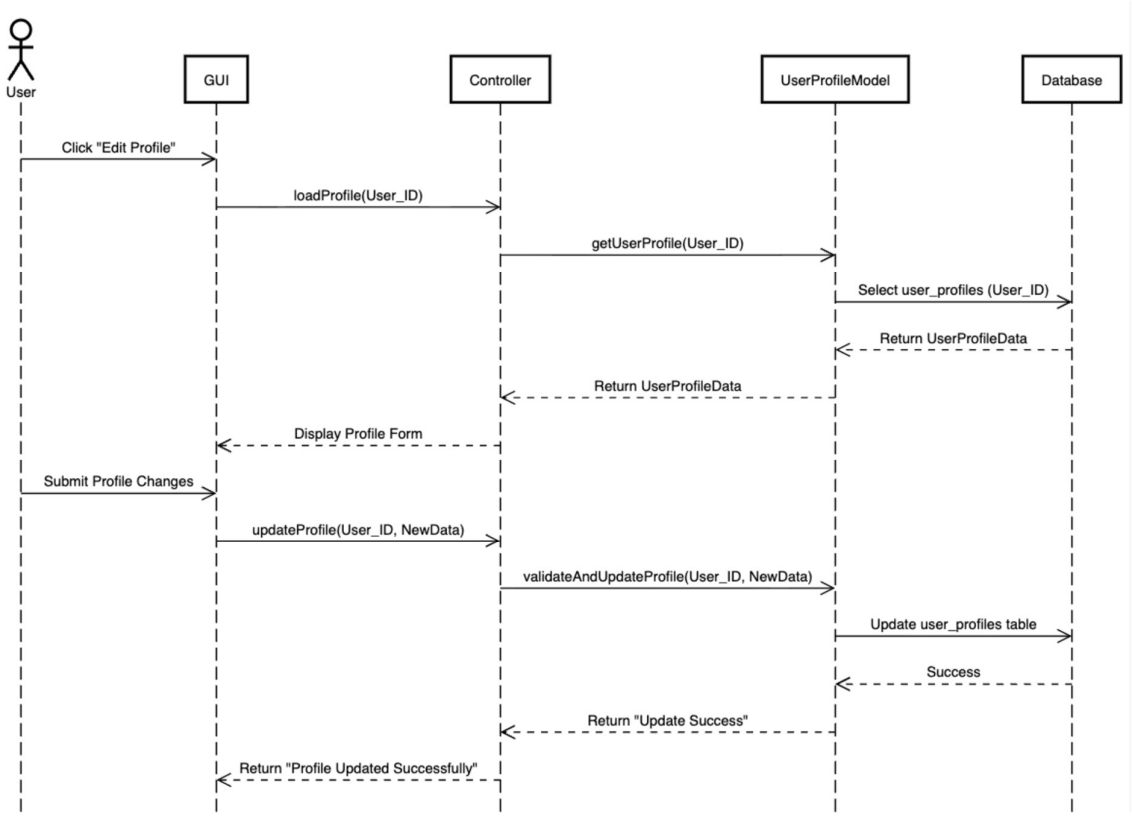


Figure 6: User - Edit Profile Sequence Diagram

Description:

- **Input Action:** User clicks "Edit Profile" and later submits a form with updated personal details.
- **Flow:** The Controller fetches current profile data from the Database to pre-fill the form for the user. Once the user submits changes, the UserProfileModel validates the new data and executes an update in the Database.
- **Output Action:** Successful: "Profile Updated Successfully" message is displayed. Failure Message: "Validation failed" or "Unable to save profile changes."

4. Find Games

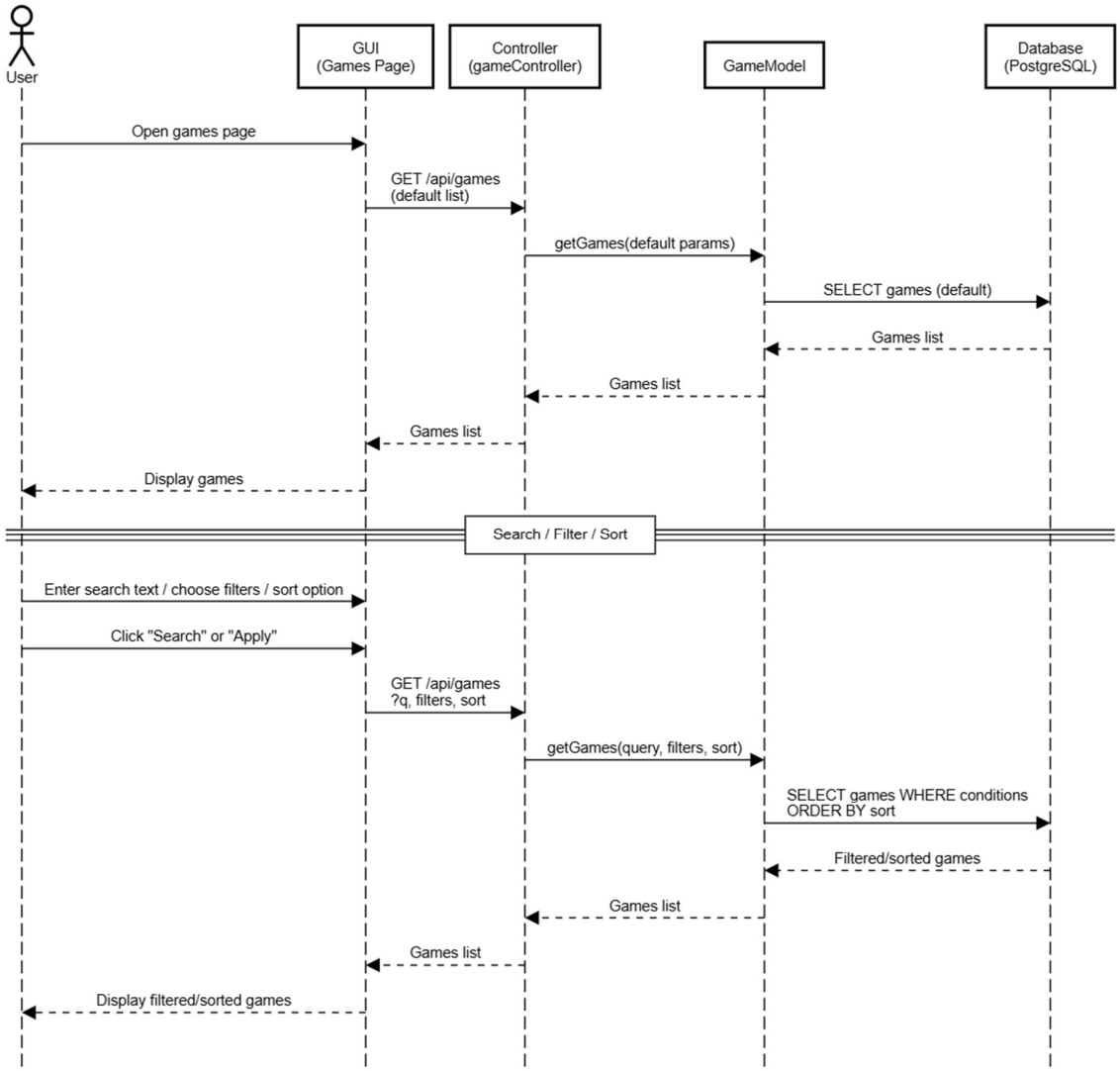


Figure 7: User- Find games Sequence Diagram

Description:

- **Input Action:** User opens the games page or enters search terms, filters, and sorting preferences.
- **Flow:** The GUI initially requests a default list of games, which the Controller fetches from the Database. When the user applies specific search criteria or filters, the Controller instructs the GameModel to execute a refined query to pull only matching results.
- **Output Action**: Successful: The GUI dynamically updates to display the filtered or sorted list of games. Failure Message: "No games found matching your criteria."

5. Add To Cart

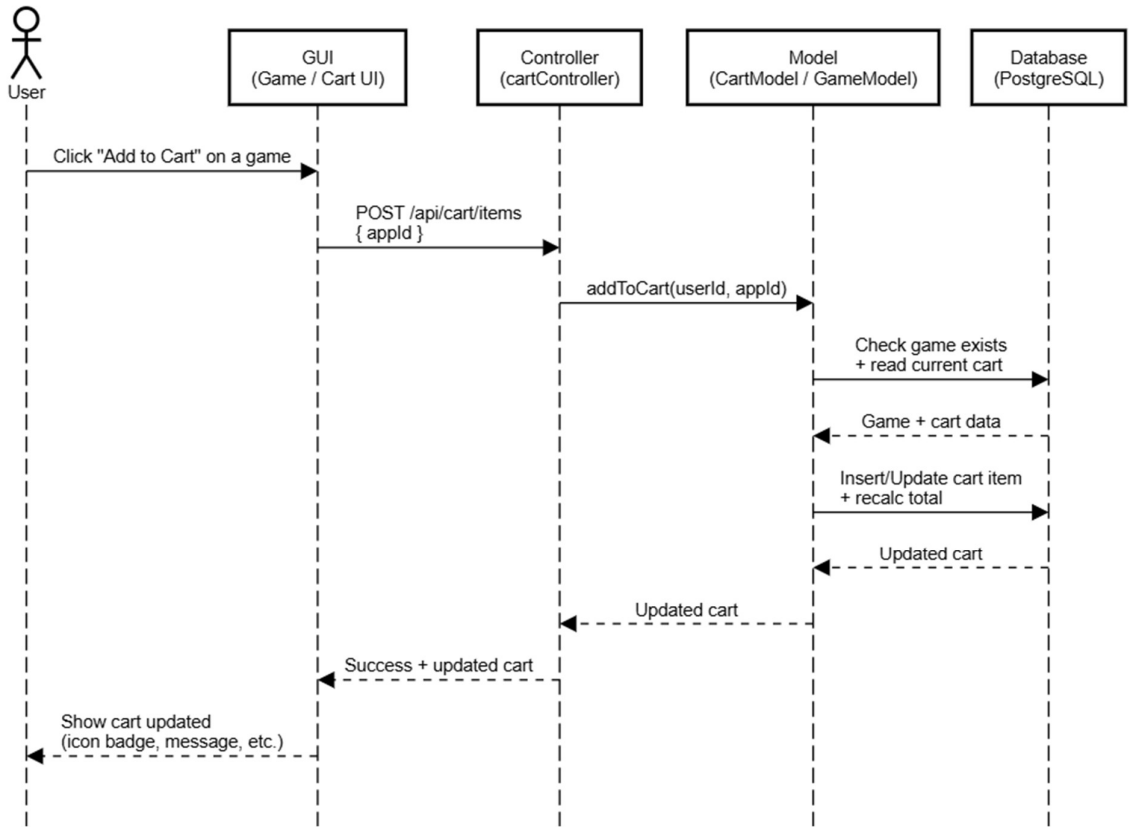


Figure 8: User - Add to Cart Sequence Diagram

Description:

- **Input Action:** User clicks the "Add to Cart" button on a specific game.
- **Flow:** The Controller verifies the game's availability and the user's current cart status via the Database. The Model then inserts the new item or updates the quantity, automatically recalculating the total price before sending the updated cart data back.
- **Output Action:** Successful: The GUI displays a confirmation message and updates visual elements like the cart icon badge. Failure Message: "Item could not be added to cart" or "Game out of stock."

6. Manage Users

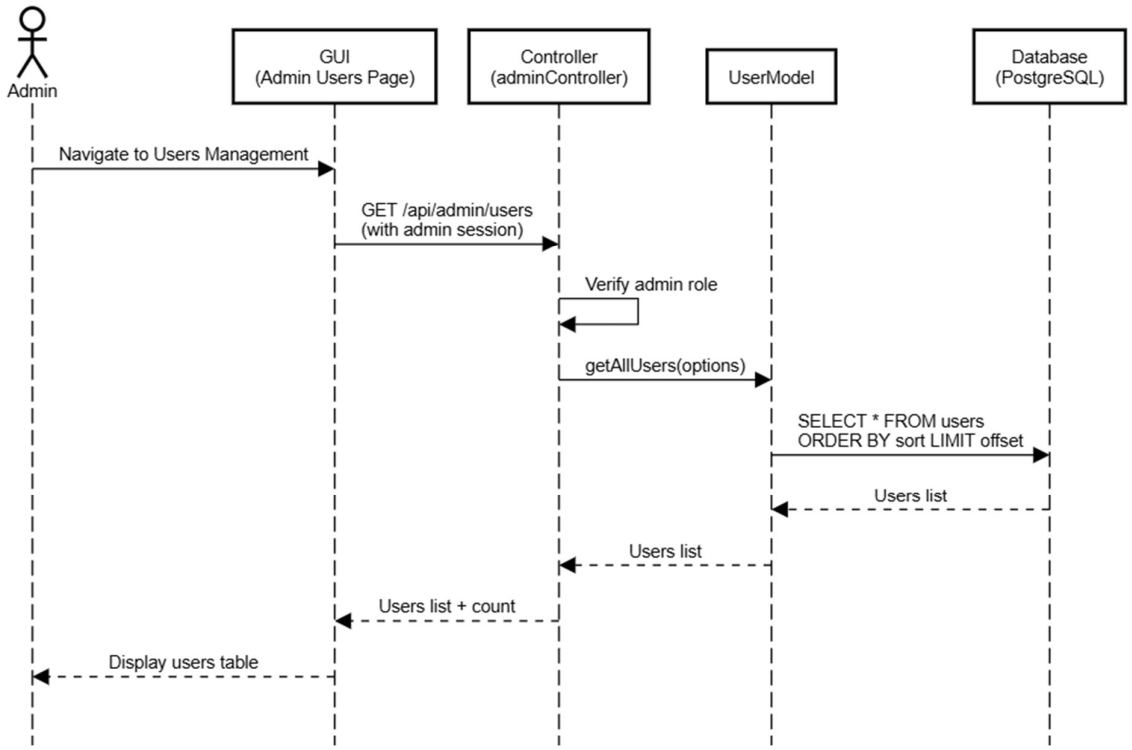


Figure 9: Admin - Manage User Sequence Diagram

Description:

- **Input Action:** Admin navigates to the User Management section in the dashboard.
- **Flow:** The Controller verifies that the user has administrative permissions before requesting the user list. Once authorized, the UserModel retrieves all user records from the Database, applying specific sorting and pagination rules to ensure the data is manageable.
- **Output Action:** Successful: The GUI renders a comprehensive table displaying all user information and the total record count. Failure Message: "Access Denied" or "Unable to load user list."

7. Manage Games

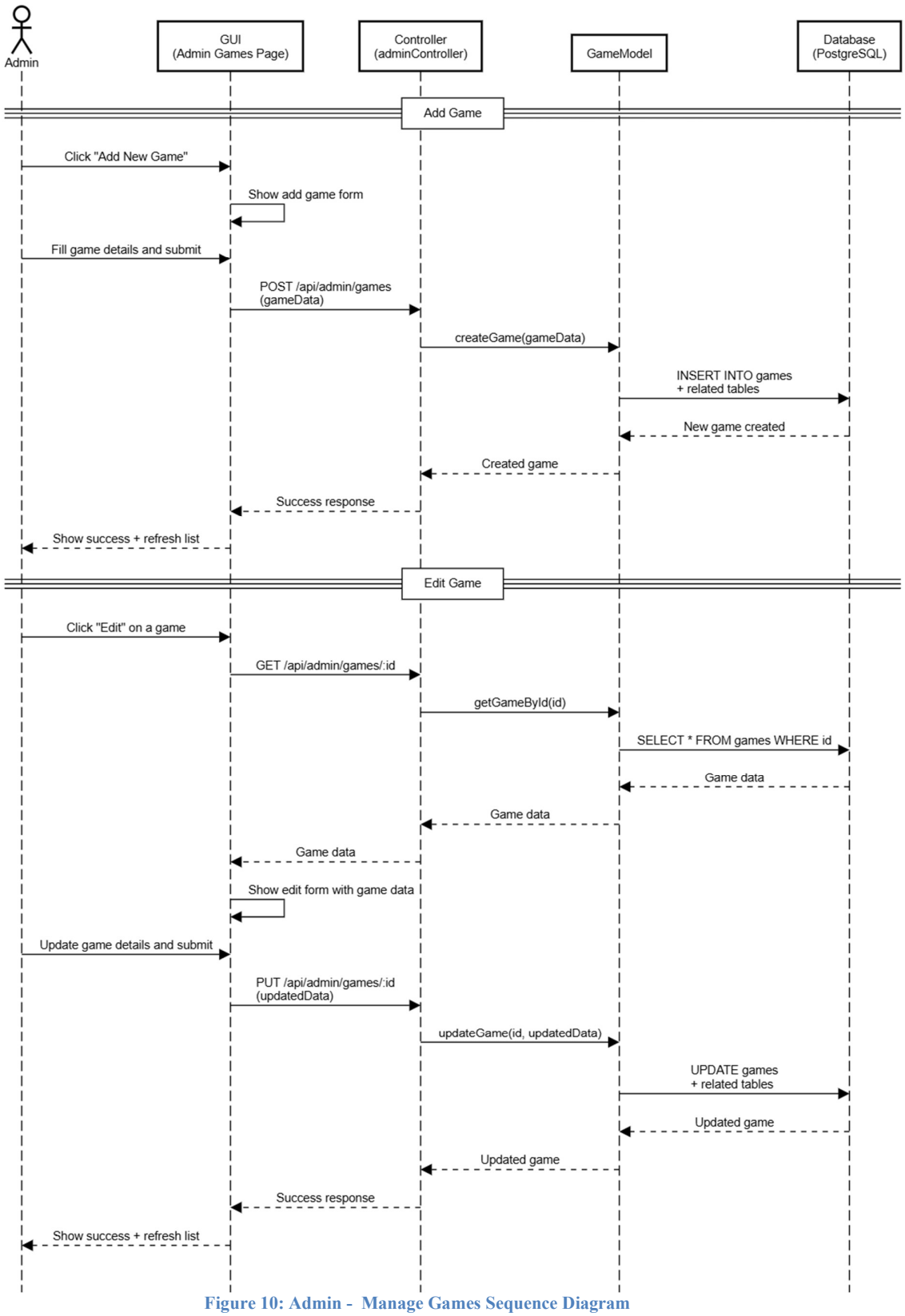


Figure 10: Admin - Manage Games Sequence Diagram

Description:

- **Input Action:** Admin submits a new game form or saves changes to an existing game.

- **Flow:** For new entries, the Controller instructs the GameModel to insert data across related tables in the Database. For edits, the Controller first fetches current game data to pre-fill the form; once submitted, it directs the GameModel to execute an update command in the Database.
- **Output Action:** Successful: A success notification appears, and the GUI refreshes to display the updated game list. Failure Message: "Error creating game" or "Failed to update game details."

8. Place Order

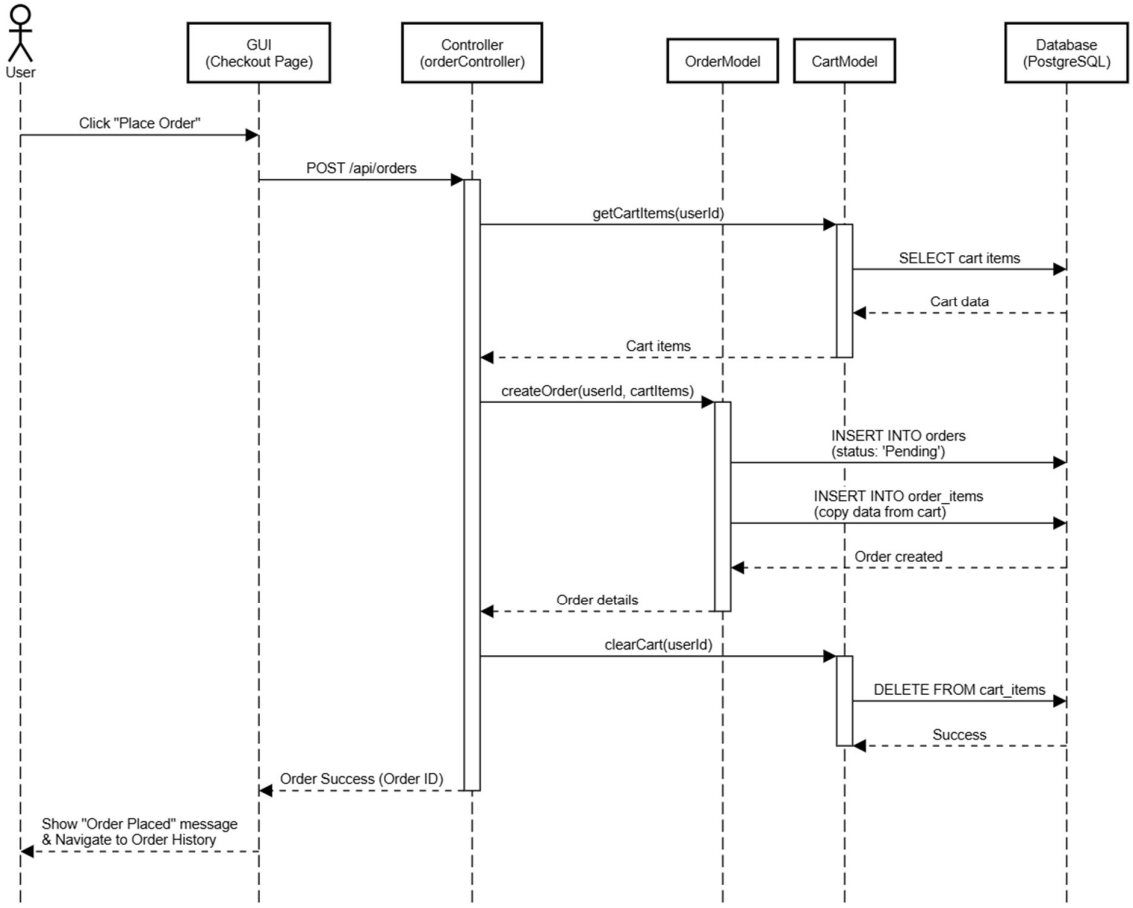


Figure 11: User - Place Order

Description:

- **Input Action:** User clicks the "Place Order" button on the checkout page.
- **Flow:** The Controller retrieves all current items from the user's cart and passes them to the OrderModel to generate a new transaction. The system saves the order to the Database with a "Pending" status and then clears the user's shopping cart.

- **Output Action:** Successful: The GUI displays an "Order Placed" message and navigates the user to their Order History. Failure Message: "Failed to process order" or "Cart is empty."

9. Manage Orders

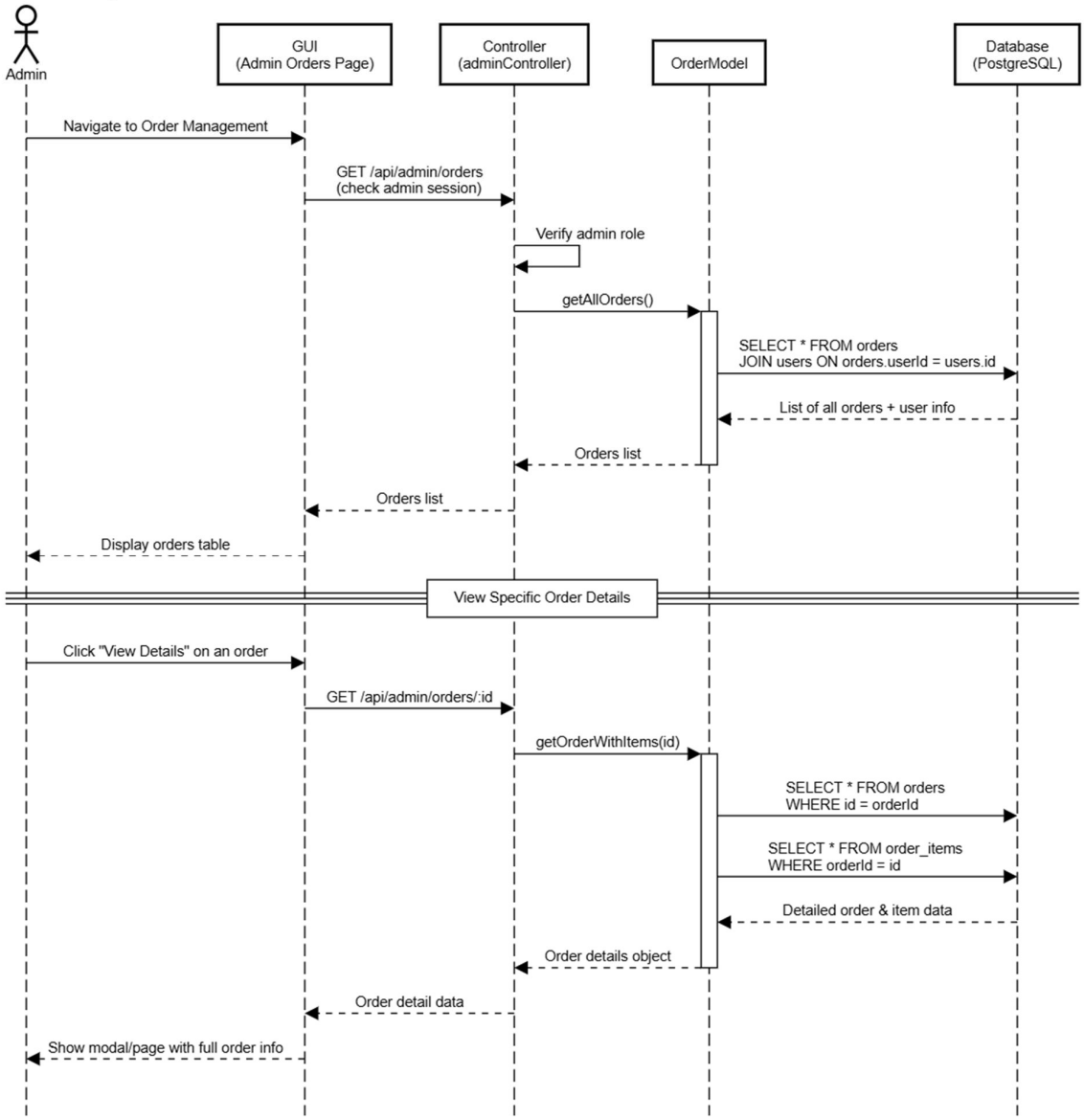


Figure 12: Admin - Manage Order

Description:

- **Input Action:** Admin navigates to the Order Management section or selects a specific order to inspect.
- **Flow:** The **Controller** verifies administrative permissions before fetching a complete list of all transactions from the **Database**. When a specific order is selected, the

system retrieves a deep-dive breakdown, including user information and every individual item purchased.

- **Output Action: Successful:** The **GUI** renders a comprehensive table of all orders or a detailed modal showing specific purchase information. **Failure Message:** "Access Denied" or "Unable to load order details."

10. Create Review

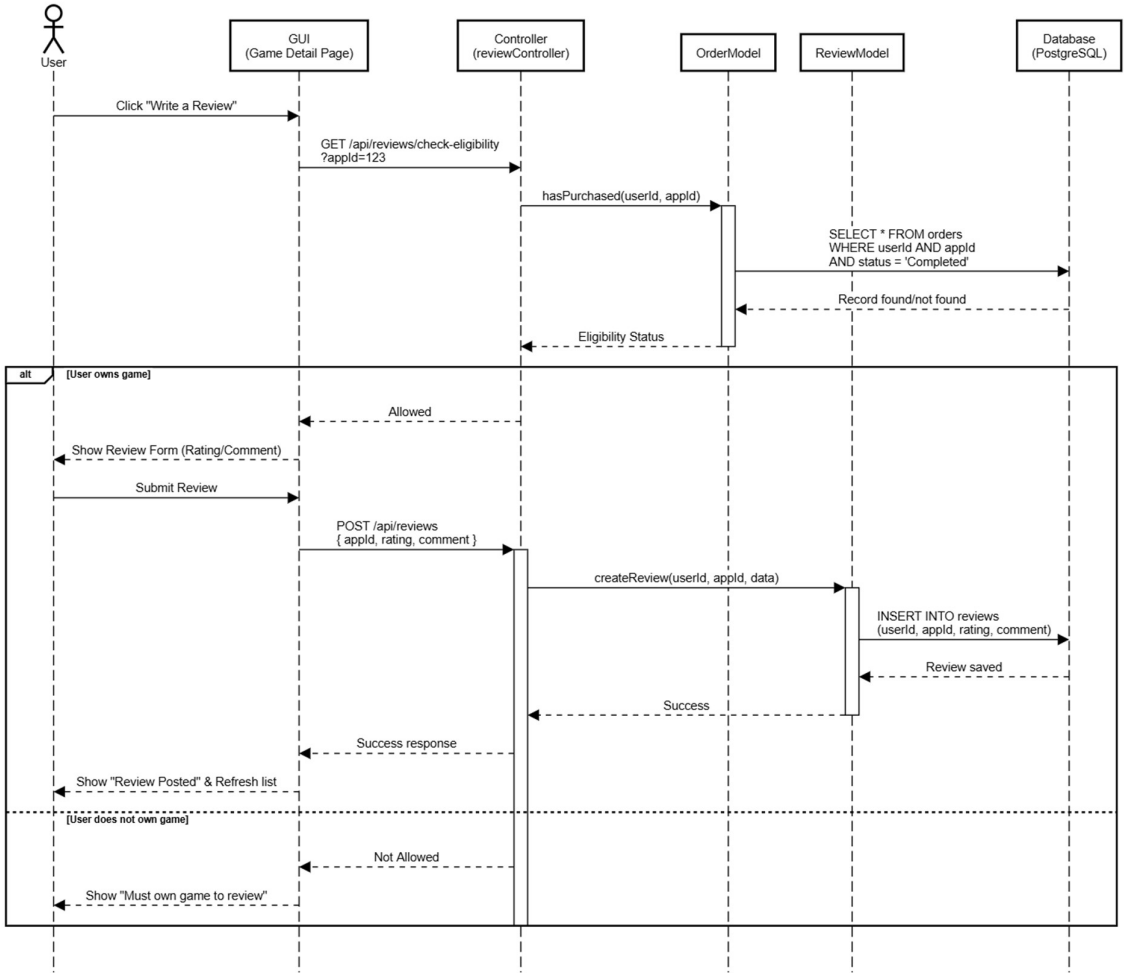


Figure 13: User - Create Review

Description:

- **Input Action:** User selects a game they own and submits a rating along with a written comment.
- **Flow:** The **Controller** first checks the **OrderModel** to verify the user has a completed purchase for that game in the **Database**. If eligible, the **ReviewModel** saves the new rating and comment, linking them to the user's account and the specific game.

- **Output Action: Successful:** The review is posted, and the **GUI** refreshes to show the feedback. **Failure Message:** "You must own the game to write a review" or "Review submission failed."

11. Manage Review

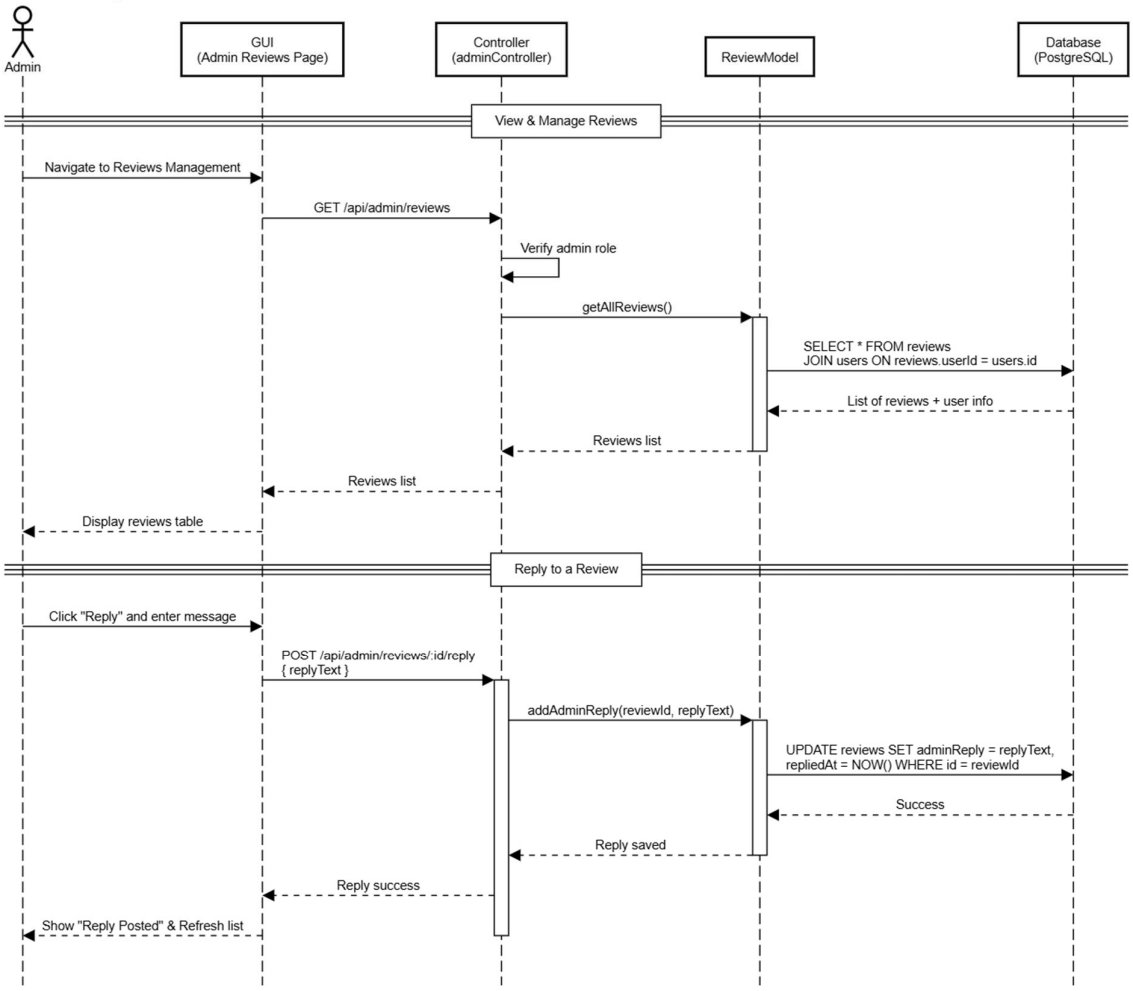


Figure 14: Admin - Manage Review

Description:

- **Input Action:** Admin browses the review list or types a response to a specific user review.
- **Flow:** The **Controller** verifies admin access and fetches all reviews from the **Database** via the **ReviewModel**. When the admin submits a reply, the system updates the original review record in the database with the admin's text and a timestamp.
- **Output Action: Successful:** The **GUI** displays the admin's reply attached to the user's review. **Failure Message:** "Access Denied" or "Unable to post reply."

12. Order Confirm

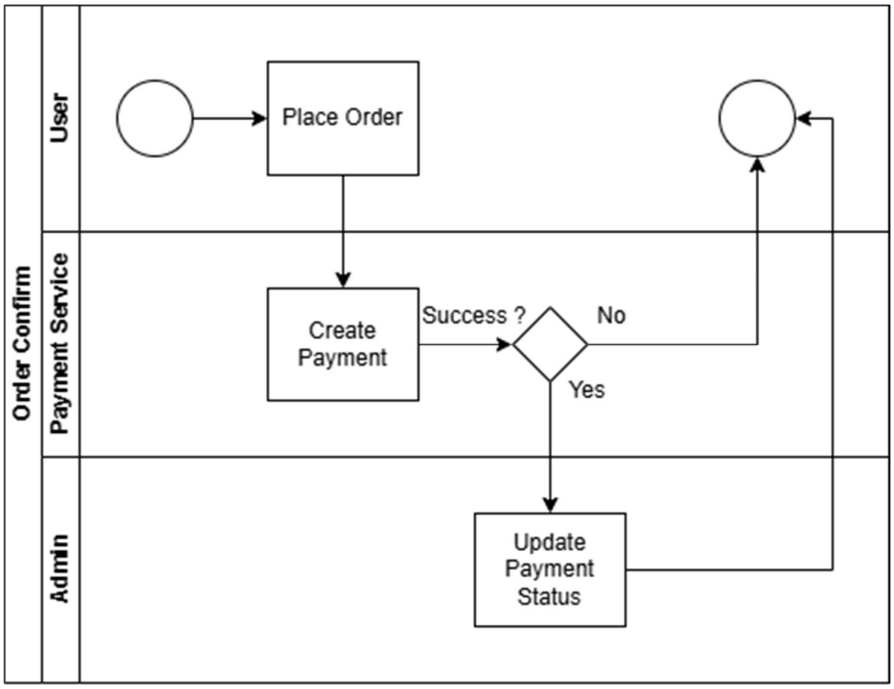


Figure 15: Order Confirm Activity Diagram

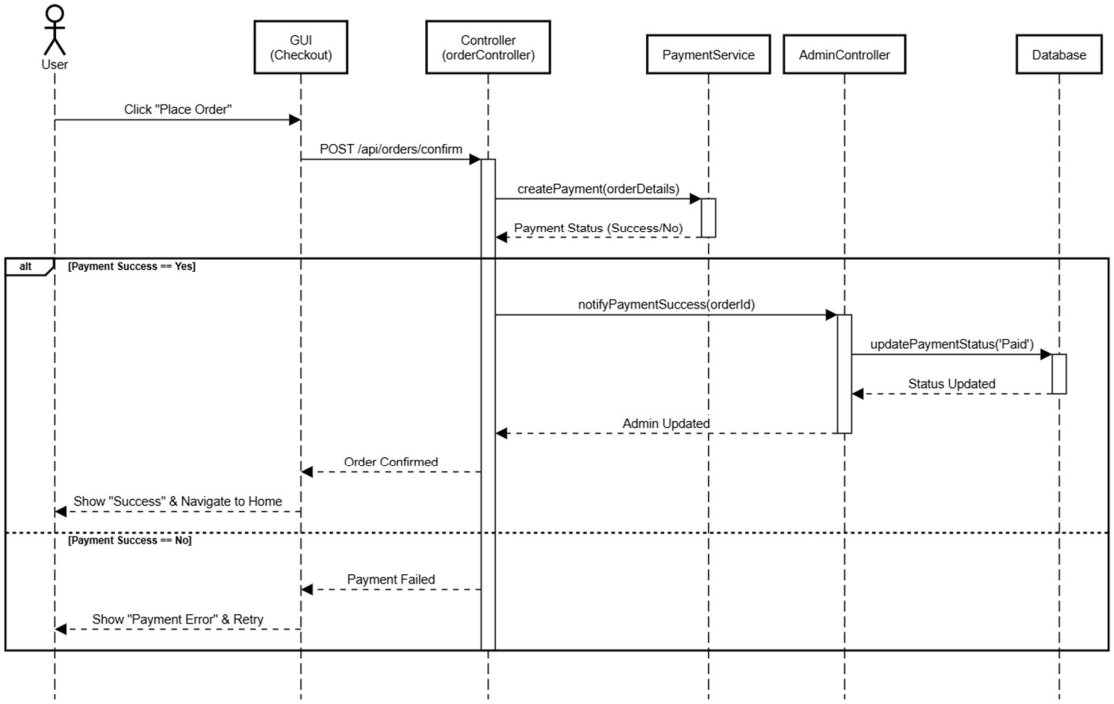


Figure 16: Order Confirm Activity Diagram

Description:

- **Input Action:** User clicks the "Place Order" button to finalize their purchase.
- **Flow:** The **Controller** initiates a request to the **PaymentService** to process the transaction. If the payment is successful, the **Admin** is notified to update the payment

status to "Paid" in the **Database**. If the payment fails, the process terminates and returns an error.

- **Output Action: Successful:** The **GUI** displays an "Order Confirmed" message and navigates the user home. **Failure Message:** "Payment Failed" or "Transaction Declined," prompting the user to retry

3.3.3. Technology Design

Frontend – UI:

React.js serves as the client-side presentation layer of the application. It is utilized to build a high-performance, single-page application (SPA) that provides a seamless user experience.

- Component-Based Architecture: The UI is broken down into reusable components (such as GameCards, Navbar, and Modals), making the codebase easier to maintain and scale.
- State Management: It handles dynamic data updates, ensuring that when a user adds a game to their cart or filters a list, the interface updates instantly without a full page reload.
- API Integration: React uses the Axios or Fetch API to communicate with the Node.js backend, sending user inputs and rendering the resulting JSON data.

Backend:

The server-side logic is built using Node.js and the Express.js framework, acting as the bridge between the user interface and the database.

- Asynchronous Environment: Node.js allows the server to handle multiple concurrent requests efficiently, which is vital for a platform with high user interaction.
- RESTful API Design: Express.js is used to create structured API endpoints (e.g., /api/auth, /api/games, /api/orders) that follow standard HTTP methods.
- Middleware & Security: This layer handles critical business logic, including JWT-based session authentication, role-based access control (Admin vs. User), and input validation to ensure data integrity.

Database:

PostgreSQL is the relational database management system (RDBMS) used for persistent data storage. It was chosen for its reliability and ability to handle complex data relationships.

- Relational Structure: It organizes data into structured tables with defined relationships (Foreign Keys), ensuring that orders are correctly linked to specific users and games.

- Data Integrity: PostgreSQL enforces strict data types and constraints, preventing inconsistent data entries, such as a user purchasing a game that does not exist.
- Scalable Queries: It supports advanced SQL querying, allowing the application to perform fast searches, complex filtering, and detailed administrative reporting.

MVC Model

In our system, the MVC (Model-View-Controller) architecture is used to separate concerns, making the application easier to organize, test, and maintain:

- Model (PostgreSQL & Models): This layer represents the data and business logic. The UserModel, GameModel, and OrderModel define how data is structured and handle all direct interactions with the PostgreSQL database.
- View (React.js GUI): This is the User Interface that the user sees and interacts with. It is responsible for displaying data provided by the Model and sending user inputs (like login credentials or search filters) back to the Controller.
- Controller (Node.js & Express.js): The authController, gameController, and others act as the "brains" or mediators. They receive requests from the GUI, process logic (such as verifying roles or generating OTPs), and instruct the Models to update or retrieve data.

By decoupling the data (Model), the interface (View), and the logic (Controller), we ensure that changes to the UI (React) do not require a complete rewrite of the backend logic or database structure.

System Architecture

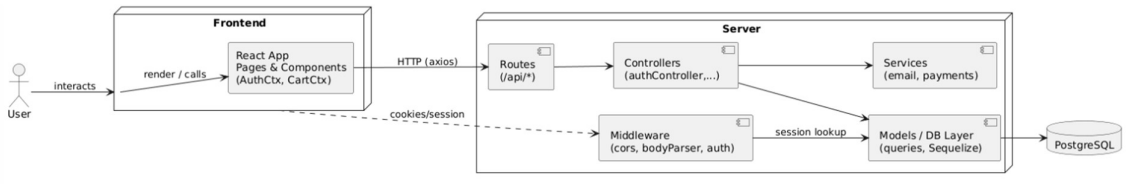


Figure 17: System Design Flow

1. Request-Response Cycle (MVC)

When a user performs an action, such as editing their profile:

1. **Input:** The user interacts with the **GUI**, which sends a request to a specific endpoint on the **Server**.
2. **Processing:** The **Controller** receives the request and coordinates with the **Model** (e.g., UserProfileModel) to fetch or validate data.
3. **Database Interaction:** The Model interacts with **PostgreSQL** to retrieve or update the relevant tables.

4. **Response:** Once the database confirms success, the result travels back through the Controller to the Frontend, which updates the interface for the user.

2. Service & Decision Logic

For complex operations like order confirmation, the system integrates external services and conditional logic:

- **Integration:** The **Controller** invokes internal or external **Services** (e.g., Payment or Email services).
- **Decision Making:** The system evaluates the outcome of these services (e.g., "Was payment successful?") to determine the next step.
- **State Update:** Upon a positive outcome, the system triggers an administrative update to the database (e.g., updating payment status) before finalizing the user's request.

3. Design Benefits

- **Scalability:** Each layer (Frontend, Server, Database) can be scaled or updated independently without disrupting the others.
- **Security:** Middleware on the server ensures that only authenticated users can access sensitive database layers.
- **Maintainability:** By separating the logic (Express) from the presentation (React), developers can modify the UI without changing how data is processed.

CHAPTER 4

IMPLEMENT AND RESULTS

4.1. In User

4.1.1 Home Page

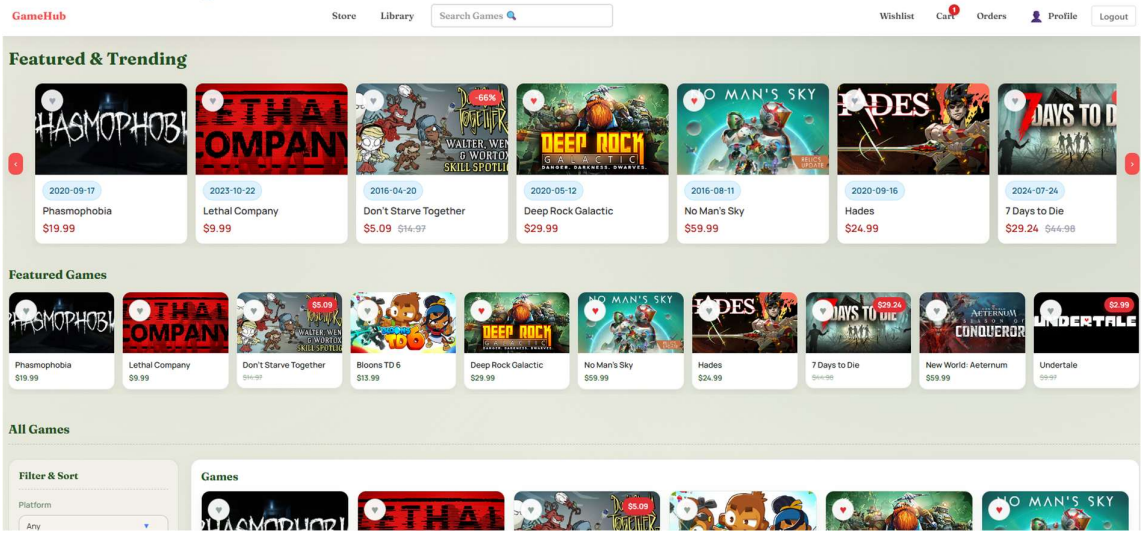


Figure 18: Home Page

Home: http://localhost:3001/: This page allows users to browse, search/filter, and manage wish list before viewing details or making a purchase. The page includes:

Featured & Trending Section Carousel: Displays recommended games (or popular if not logged in). Each card contains: Game image, Game title, Heart icon to add or remove game to/from wish list (red heart for game added, gray heart for game yet added), Release date, game final price with Discount badge (if there is any). When users click on one of the gamecards, the page will move to Game Detail page for that game. There are left and right arrows to scroll the game list and see more.

Filter & Sort Panel: At the All Games section users can choose games that suit their platform, genres, game categories, languages. Price range & sort allows users to adjust Min/Max price inputs, choose radio Asc or Desc before choosing Clear to reset filters or Sort by price to apply sort. Filters are automatically applied when changed, and the games will be displayed on the right column according to the filters selected. The column has an infinite scroll to automatically load more when scrolling near the end.

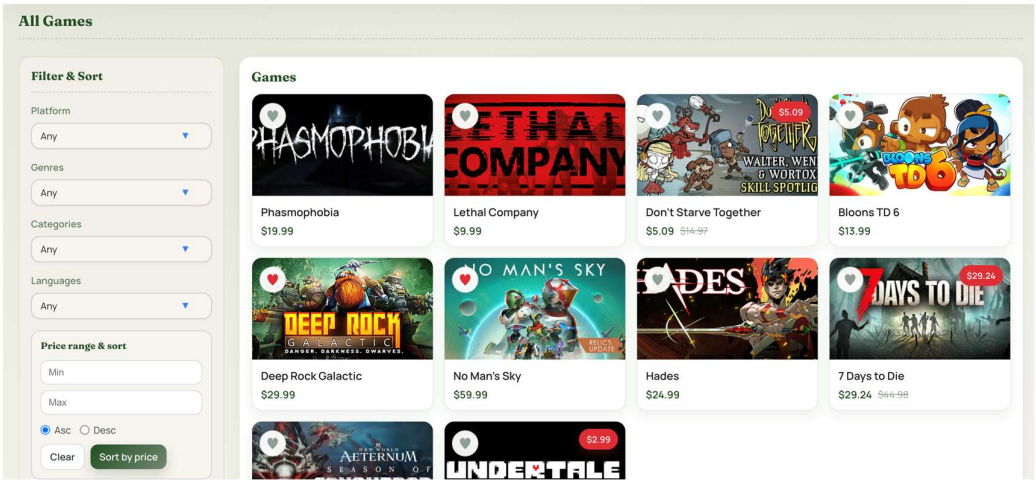


Figure 19: Sort and Filter section

4.1.2 Game Detail Page

Figure 20: Game Detail Page

Game Detail :http://localhost:3001/game/1966720 : Displays full game details, system requirements and game price, allows users to purchase, add to the wish list, and submit reviews.

Add to Cart: Users click **Add to Cart to** add game to cart (this will redirect to login if not authenticated).

Add to Wishlist: Users click **Add to Wishlist / Remove from Wishlist** to toggles wish list status for the game they are viewing (redirects to login if not authenticated).

Reviews: Users choose Recommended or Not Recommended review for the game and write/edit review in the Recommendation box. If the review is responded by the Developers, the response will be shown under the user review and beneath that, the reviews from other users can also be displayed.

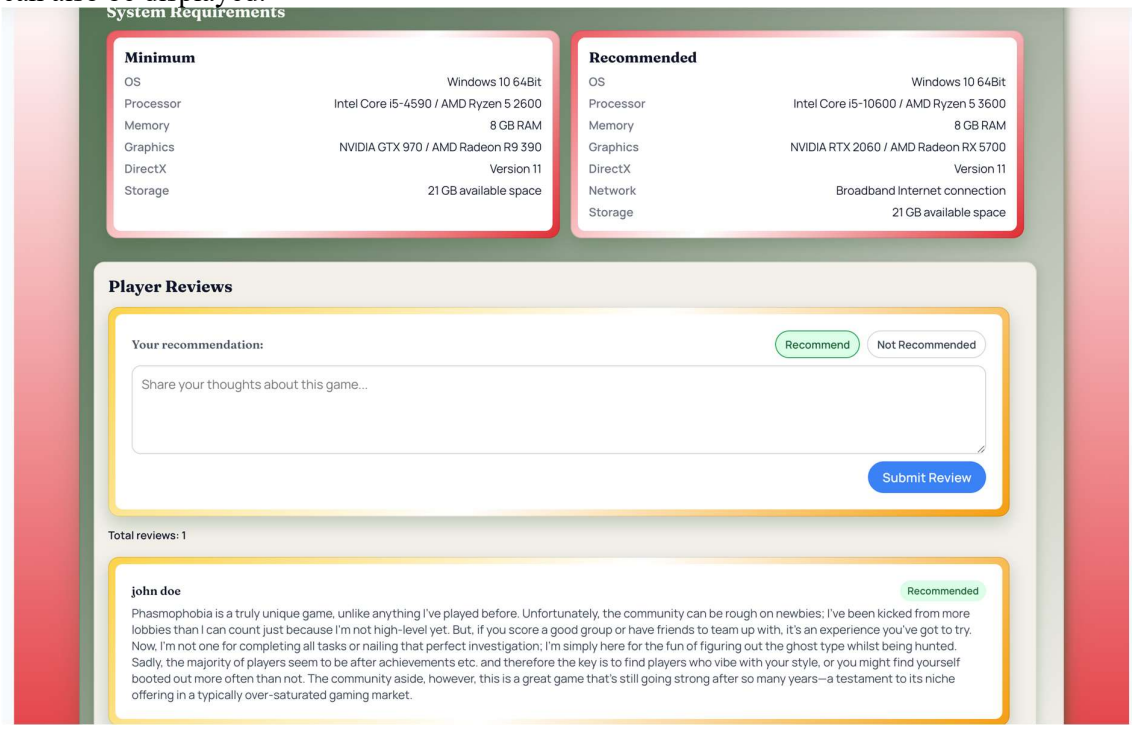


Figure 21: Add Game Reviews

4.1.3. Wishlist Page

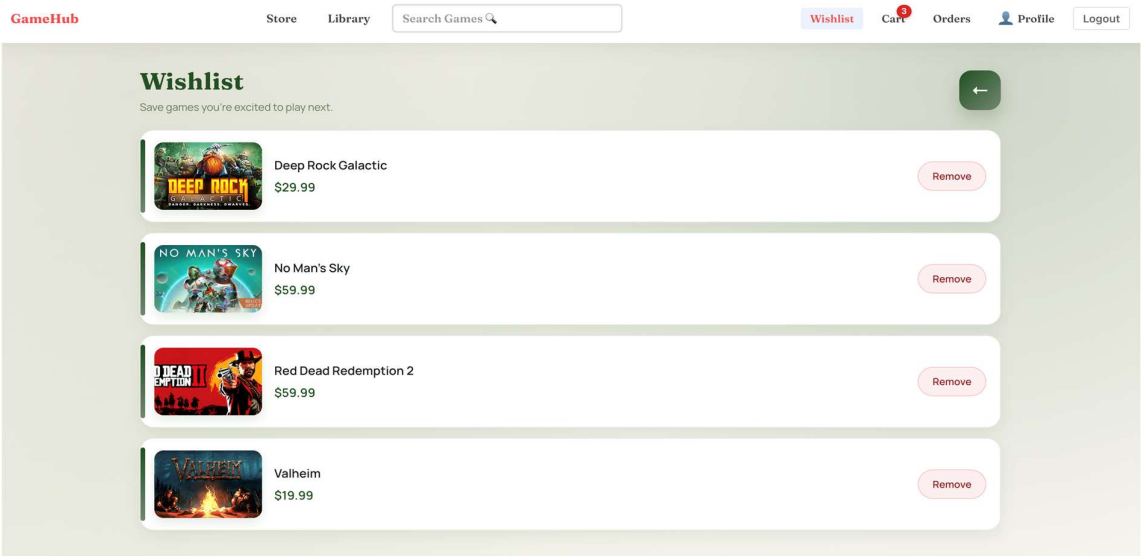


Figure 22: Wishlist Page

Wishlist: http://localhost:3001/wishlist : Displays games saved to the Wish list. Users can check the games they wish and their final price, they can also remove items if they want.

4.1.4. Library

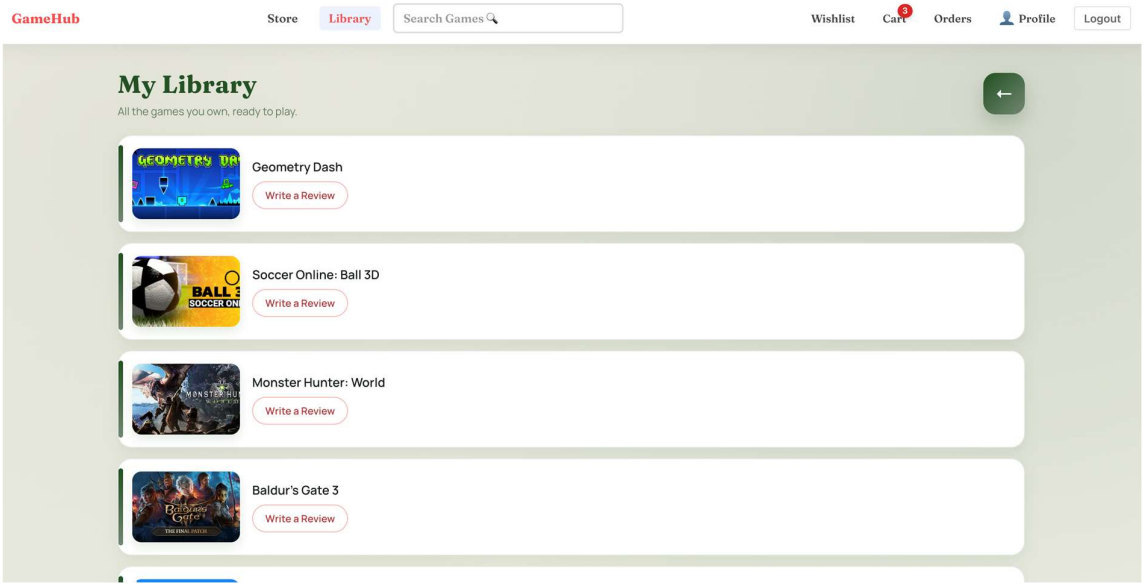


Figure 23: Library Page

Library: http://localhost:3001/library **:** Displays games the user owns and allows writing reviews. Protected route (requires authentication).

Write / Editing Review: User can click this button next to the game they own to choose Recommend status and write Review or update it.

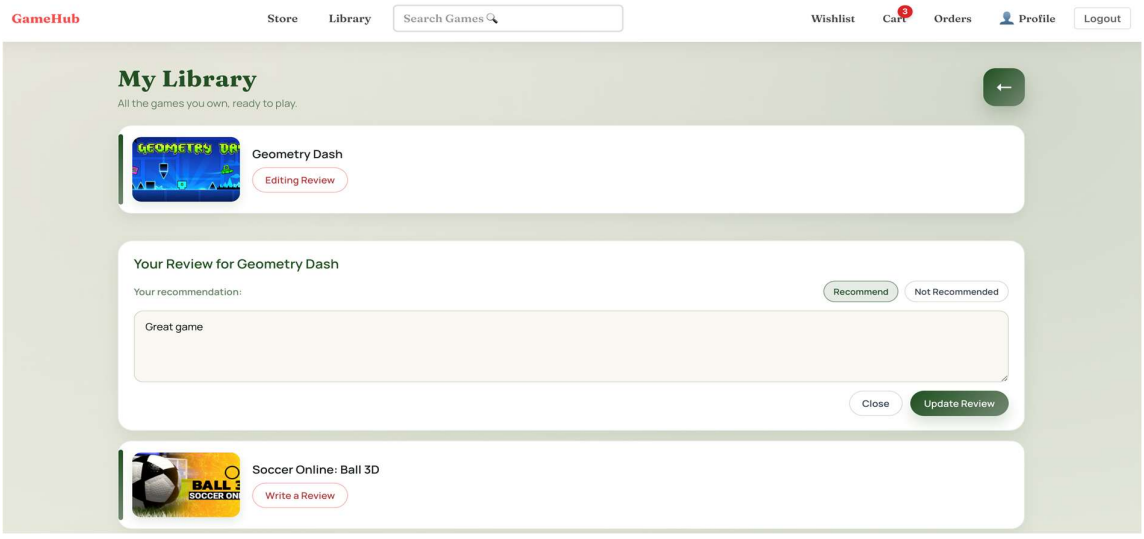


Figure 24: Write Review on Library Page

4.1.5. Cart

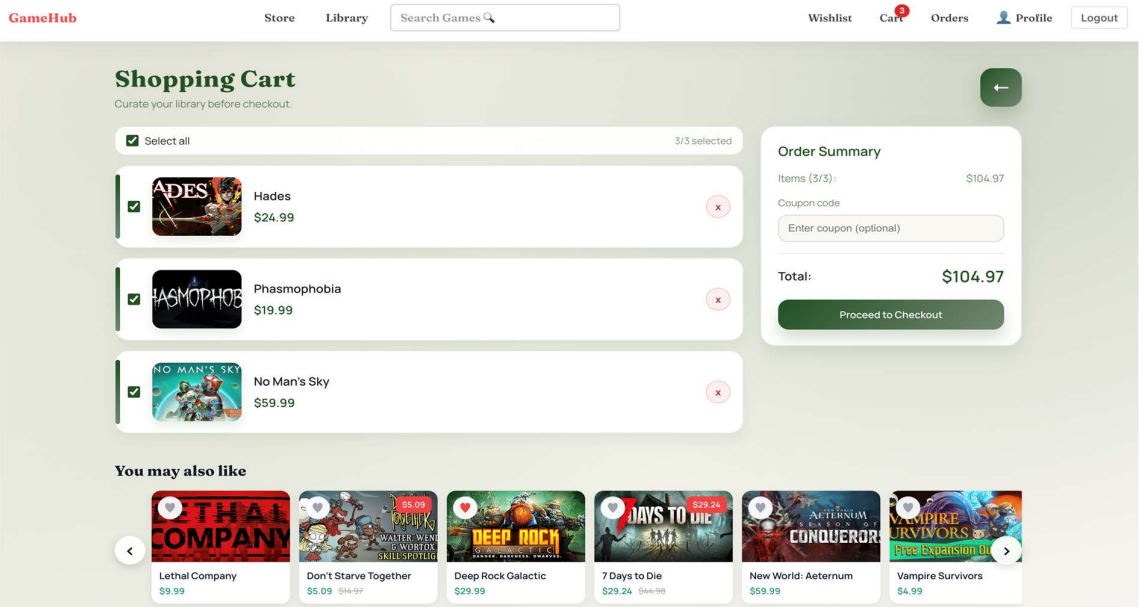


Figure 25: Shopping Cart Page

Cart: http://localhost:3001/cart**:** Displays items in the shopping cart. Users can select items, apply coupons, and proceed to checkout. Users may also check out some other games suggested by the system (based on the first cart item) below the Cart.

Once a game gets added to Cart, it will go to the waiting list in this page so users can double check what they mark to buy and its price by clicking on the checkbox on the left of the game, or choose Select all to buy all the game in buying list, and they can also remove the game from the list with the cross button next to the game.
At the Checkout state, the number of items together with their total price will be automatically displayed on the Order Summery, users can apply the coupon codes for discount if they have. Then, the final price will be printed out.

Proceed to Checkout: Users click this button after checking the final price from Order Summery to move to Checkout page.

4.1.6. Checkout Page

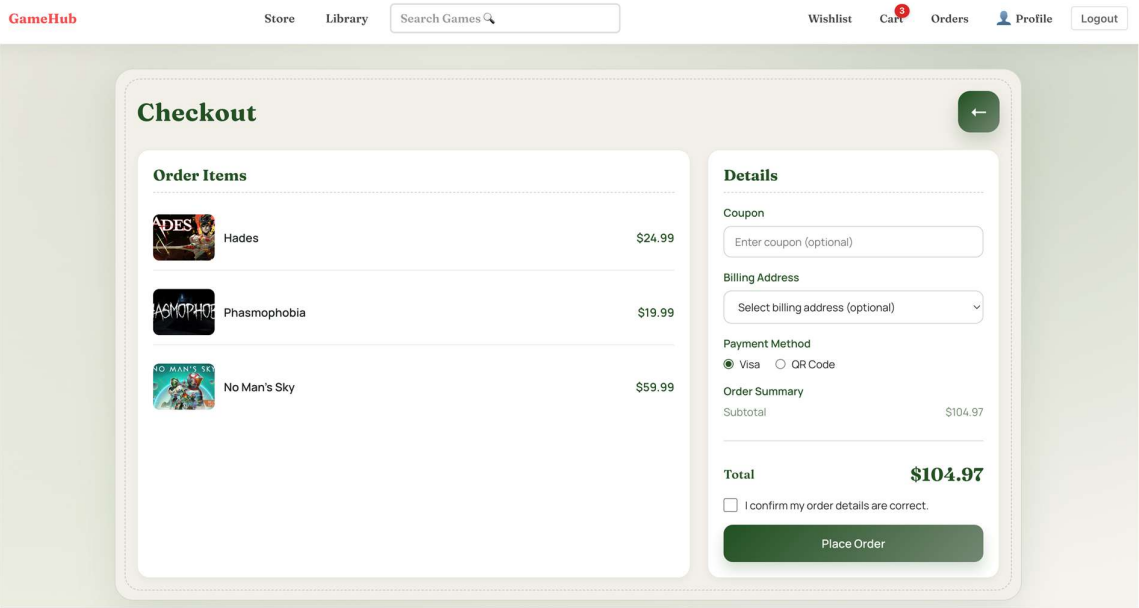


Figure 26: Checkout Page

Checkout: http://localhost:3001/checkout: Processes the order. Users can review items, apply coupons, select a billing address, choose a payment method, and place the order.

This page lists the unit price of the items being chosen. Users can apply coupon here if they have not applied to Cart's Order Summery, and input Billing address if they need bill delivery prior to picking a Payment Method.

Place Order: Users are required to check their bill price the last time and confirm in the checkbox before placing order.

4.4.7. Orders Page

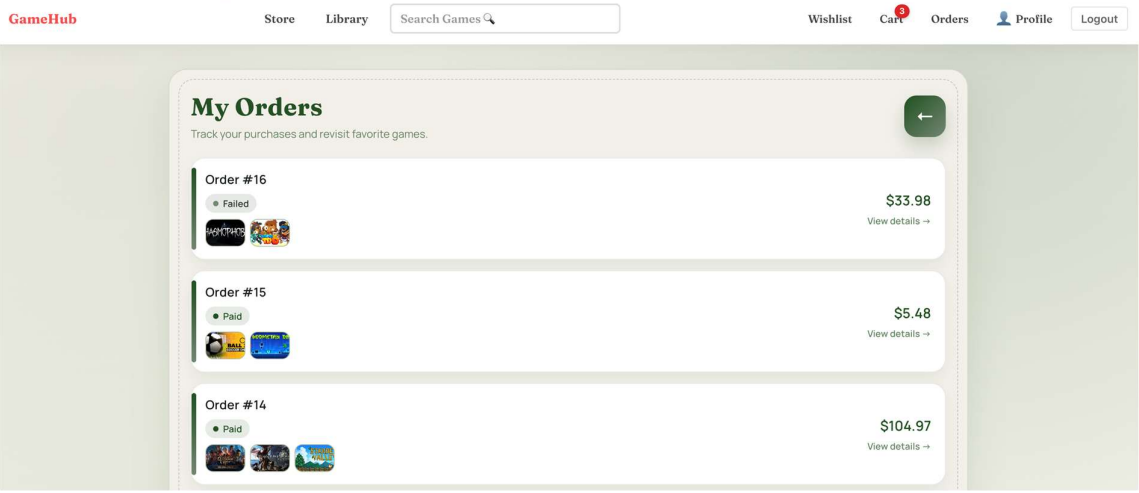


Figure 27: Orders Page

Orders Page: http://localhost:3001/orders**:** Lists order history. Users can view all orders' status and open details for any order. Clicking in View details will lead users to Order Details for more detailed content.

4.1.8. Order Details

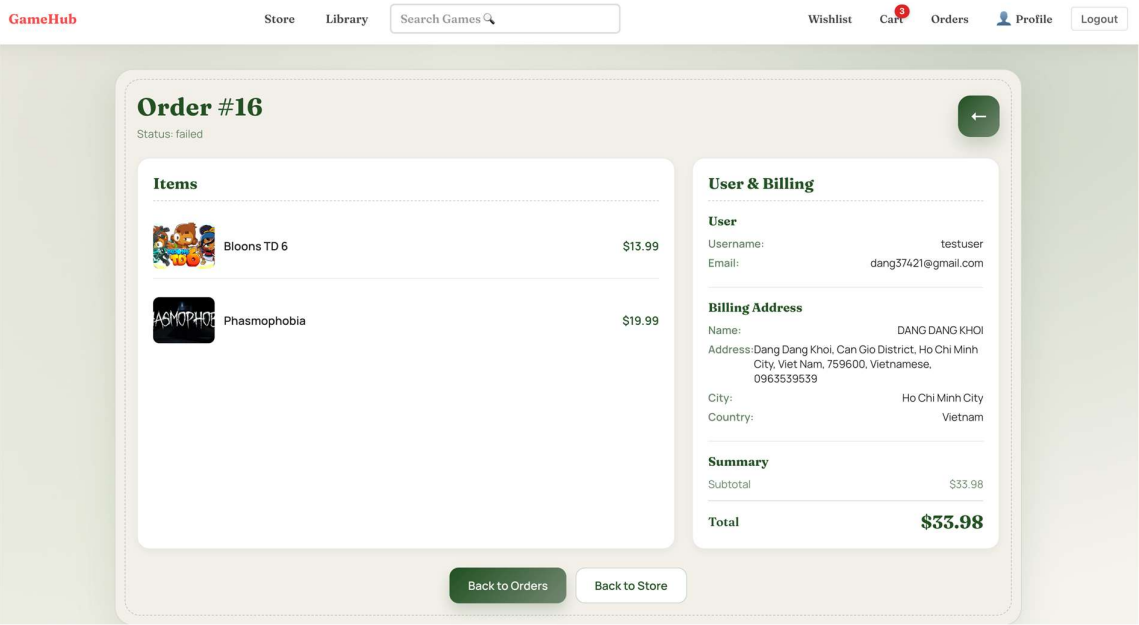


Figure 28: Order Detail Page

Order Detail Page: http://localhost:3001/orders/16 : Displays detailed information for a specific order. Users can view items, user info, billing address, and order summary. After viewing order details, the user can choose to go Back to Orders or to go Back to Store.

4.1.9. Profile

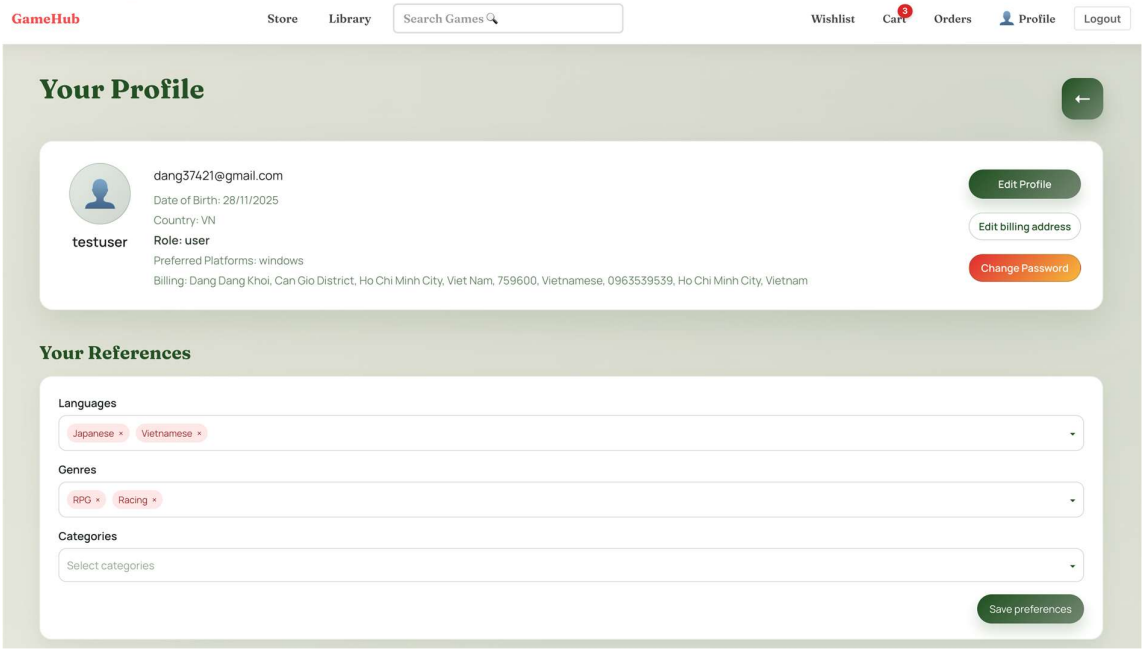


Figure 29: User Profile Page

Profile:: http://localhost:3001/profile: Manages user profile, preferences, billing address, owned games, wish list, and purchase history. Protected route (requires authentication).

User profile: Depicts Avatar, Name: Username or email prefix, some public User Info such as: Email, Date of Birth (DD/MM/YYYY or "Not set"), Country, Role (user/admin), Preferred Platforms (comma-separated), Billing Address (city, country, or "Not set").

Edit Profile: The user can click the Edit Profile button to switch to edit user info

Figure 30: Edit Profile

Edit billing address: User input into the fields: Full name, Country, Address line 1, Address line 2, City, State/Province, Postal code and press Save for updating.

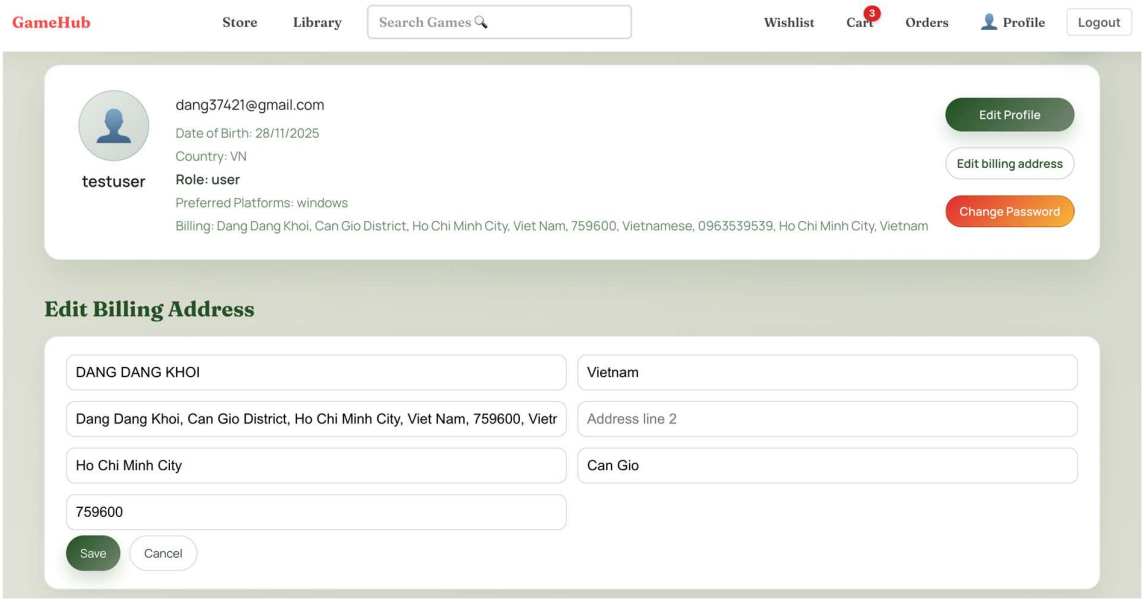


Figure 31: Edit Billing Address

Change Password button: Switches to edit mode, user needs to type old password first, then input new password and confirm it again.

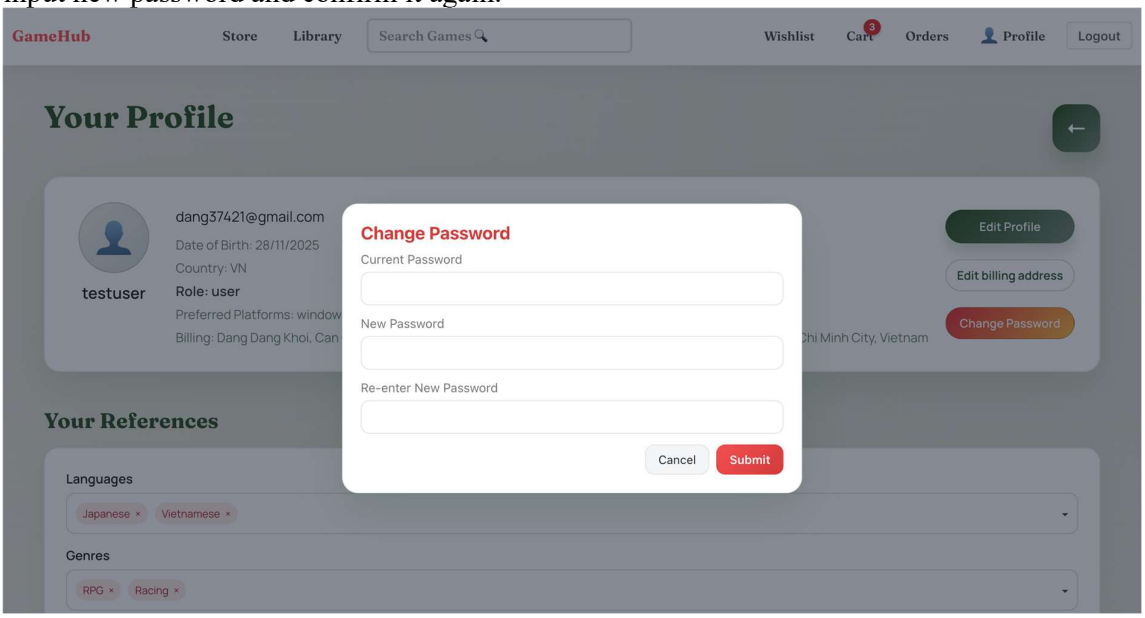


Figure 32: Change Password

User's Reference: User can edit their game references by selecting tags in the dropdowns for Languages, Genres, Categories.

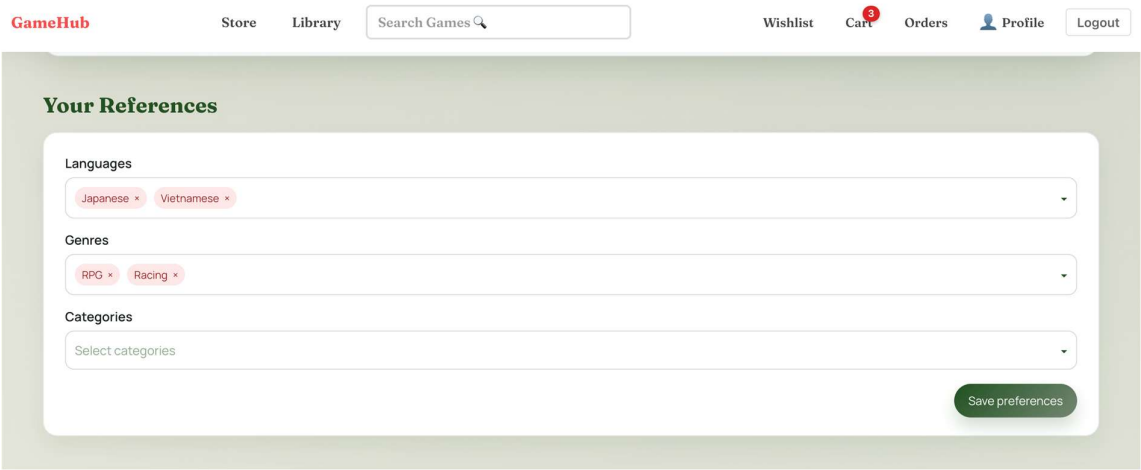


Figure 33: User Reference

Owned game & Wishlist and Purchase History: Game lists data like Owned game, Wishlist and Purchase History that the user has obtained are also displayed on Profile Page.

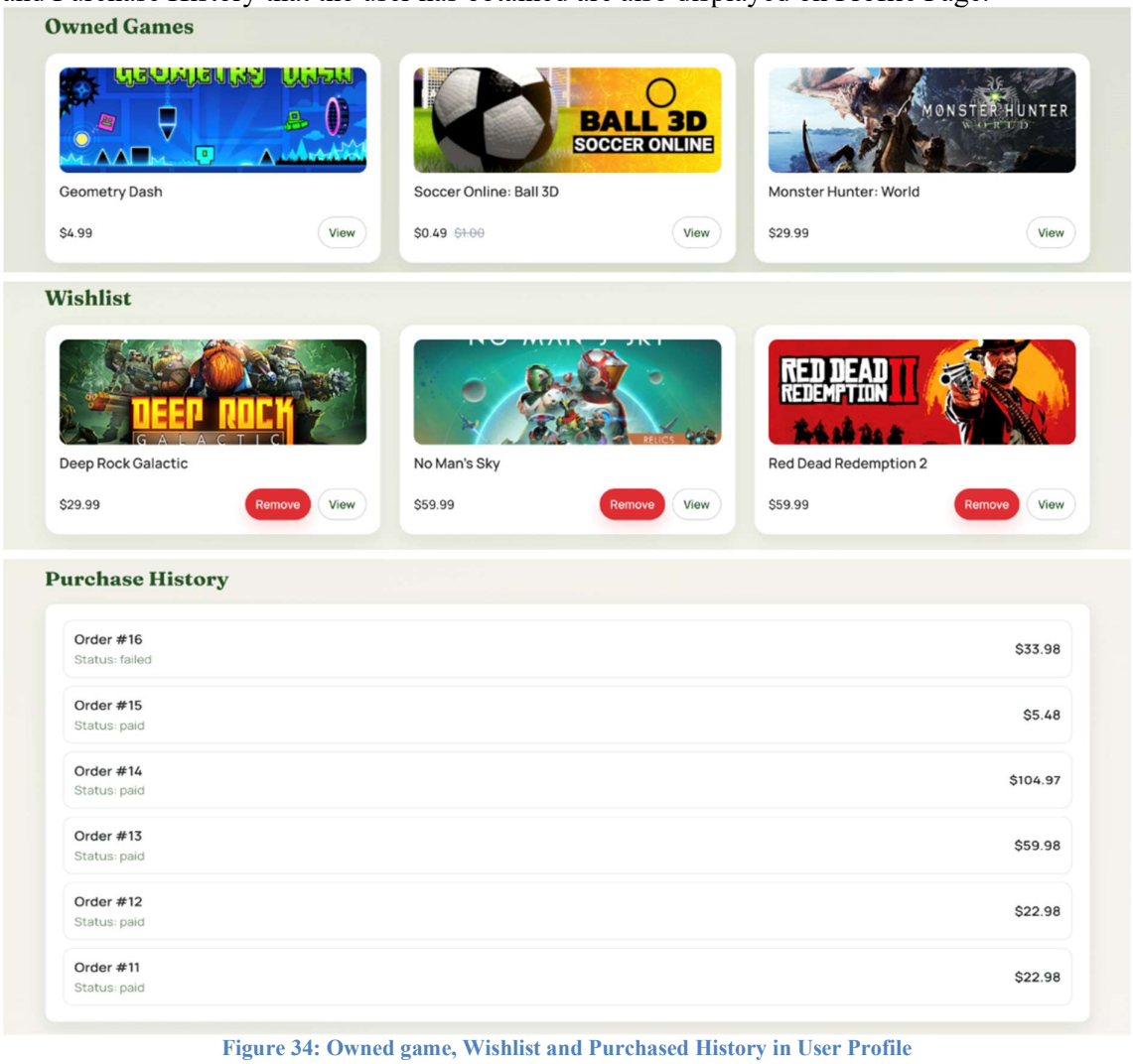


Figure 34: Owned game, Wishlist and Purchased History in User Profile

4.2. API of User

4.2.1. userService

Table 2: User API List

Method	Endpoint	Description
GET	api /user/profile	Update user profile username, email, dateOfBirth, country, preferences)
GET	api /user/billing-addresses	Get all saved billing addresses
GET	api /user/billing-addresses/:id	Get specific billing address by ID
POST	api /user/billing-addresses	Create new billing address
PUT	api /user/billing-addresses/:id	Update existing billing address
DELETE	api /users/billing-addresses/:id	Delete billing address (note: endpoint has /users instead of /user)
GET	api /wishlist	Get user's wishlist with pagination and sorting
POST	api /wishlist	Add game to wishlist (body: { appId })
DELETE	api /wishlist/:appId	Remove game from wishlist
GET	api /user/library	Get user's owned games library with pagination and sorting

4.2.2. Authentication API

Table 3: Authentication API List

Method	Endpoint	Description
POST	/auth/register	Register a new user (sends OTP verification email).
POST	/auth/login	Authenticate user and create session (email or username).
GET	/auth/me	Get current authenticated user information.
POST	/auth/logout	Destroy current user session and log out.
POST	/auth/verify	Verify user email using OTP code.
POST	/auth/resend-otp	Resend OTP verification code to email.

4.2.3. Reference API

Table 4: Reference API List

Method	Endpoint	Description
GET	/reference/languages	Get all available languages.
GET	/reference/languages/:id	Get a language by ID.
GET	/reference/categories	Get all game categories.
GET	/reference/categories/:id	Get a category by ID.
GET	/reference/genres	Get all game genres.
GET	/reference/genres/:id	Get a genre by ID.

4.2.4. Games API

Table 5: Games API List

Method	Endpoint	Description
GET	/games	Paginated list of games with filters and sorting.
GET	/games/:appId	Get detailed information about a game.
GET	/games/search	Full-text search for games
GET	/games/search/autocomplete	Autocomplete suggestions for game search.
GET	/games/featured	Get featured/promoted games.
GET	/games/recommended	Personalized recommendations (optional auth).
GET	/games/discounted	Personalized recommendations (optional auth).
GET	/games/newest	Newest games added to the platform.
GET	/games/genre/:genreId	List games by genre.
GET	/games/category/:categoryId	List games by category.
GET	/games/:appId/reviews	List reviews for a game.
GET	/games/:appId/review/me	Get current user's review for a game.

4.2.5. Cart API

Table 6: Cart API List

Method	Endpoint	Description
GET	/cart	Get current user's shopping cart.
GET	/cart/count	Get number of items in cart.
POST	/cart/items	Add a game to the cart.
DELETE	/cart/items/:appId	Remove a game from the cart.
DELETE	/cart	Clear the cart.
GET	/cart/validate-coupon	Validate a coupon code.
POST	/cart/checkout	Checkout the cart and create an order.

4.2.6. Wishlist API

Table 7: Wishlist API List

Method	Endpoint	Description
GET	/wishlist	Get user's wishlist.
POST	/wishlist	Add a game to wishlist.
DELETE	/wishlist/:appId	Remove a game from wishlist.
GET	/wishlist/check/:appId	Check if a game is in wishlist.

4.2.7. Library

Table 8: Library API List

Method	Endpoint	Description
GET	/library	Get user's owned games (library).
POST	/library/review	Create or update a review for an owned game.

4.2.8. Orders API

Table 9: Order API List

Method	Endpoint	Description
GET	/orders	Get user's order history.
GET	/orders/:id	Get details for a specific order.

4.2.9. Payments API

Table 10: Payment API List

Method	Endpoint	Description
POST	/payments	Create a payment transaction for an order.

4.2.10. Health Check API

Table 11: Health Check API

Method	Endpoint	Description
GET/	/health	Check server health/status.

4.3. In Admin

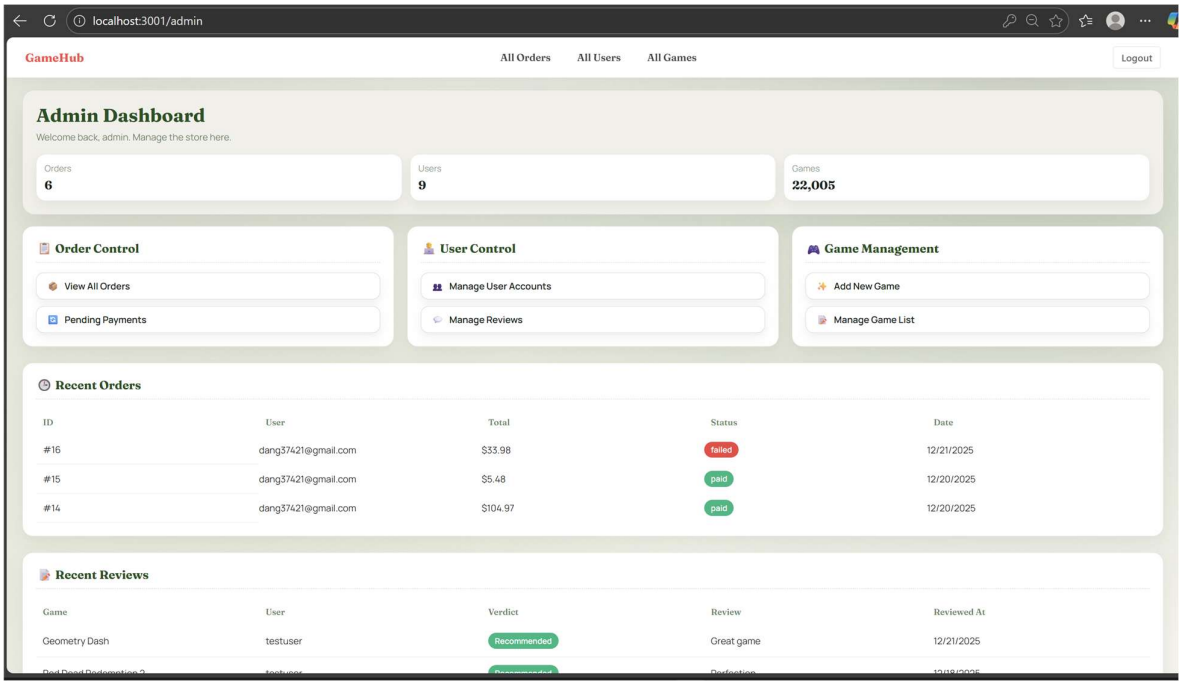


Figure 35: Admin dashboard

URL: http://localhost:3001/admin/dashboard

The **Admin Dashboard** provides administrators with a centralized overview of the system, displaying key statistics such as total orders, users, and games, along with recent orders and reviews to support quick monitoring and decision-making.

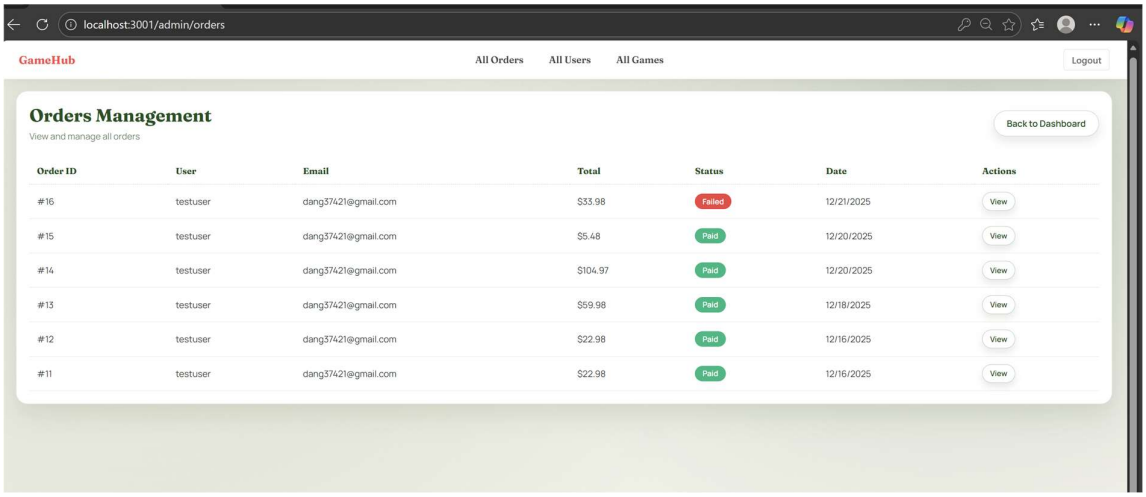


Figure 36: Order Management

URL: http://localhost:3001/admin/orders

The **Orders Management** page allows administrators to view and track all orders in the system, showing essential information such as order details, payment status, and order date to support efficient transaction monitoring.

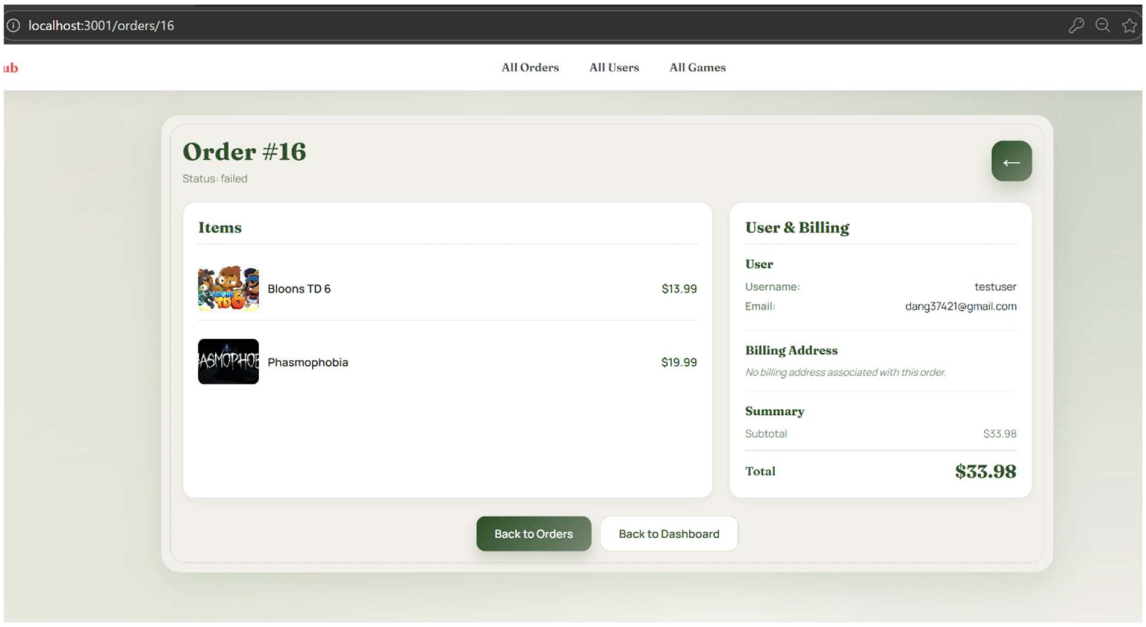


Figure 37: Detail of order

URL: http://localhost:3001/admin/orders/{orderId}

The **Order Detail** page displays comprehensive information about a selected order, including purchased items, user information, order status, and total cost, enabling administrators to review transactions in detail.

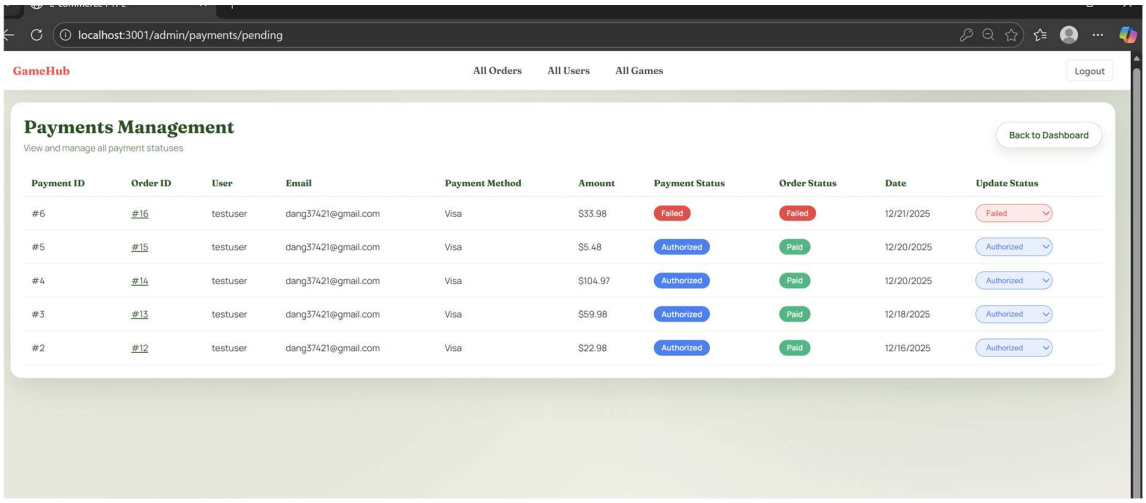


Figure 38: Management payments

URL: http://localhost:3001/admin/payments

The **Payments Management** page enables administrators to monitor and update payment transactions, helping resolve failed or pending payments and maintain financial accuracy within the system.

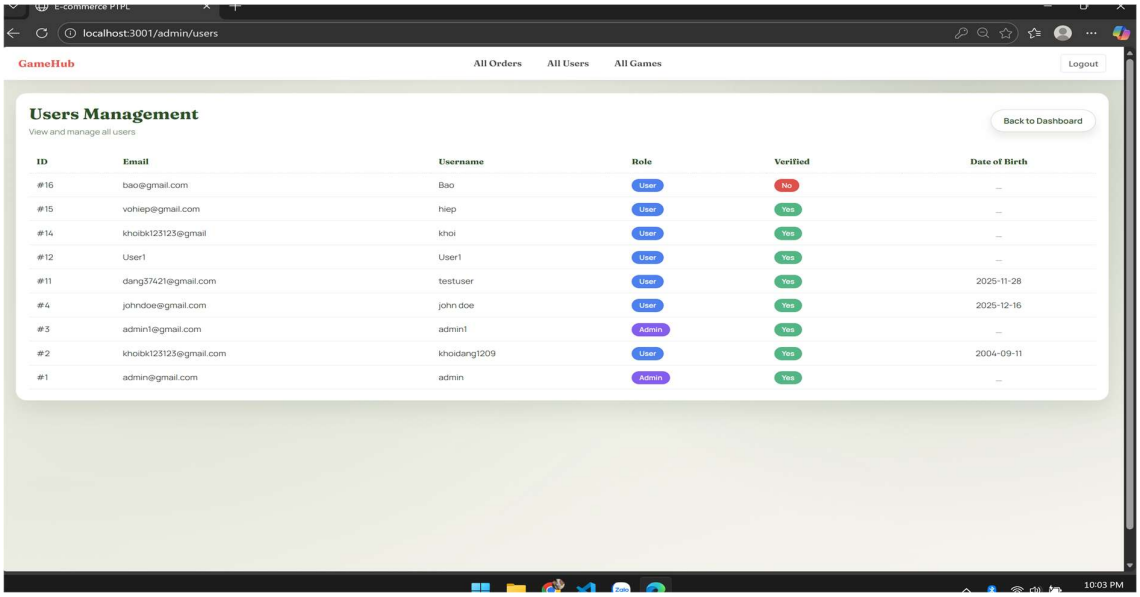


Figure 39: Management users

URL: http://localhost:3001/admin/users

The **Users Management** page allows administrators to manage registered user accounts by viewing user information, roles, and verification status to ensure proper access control.

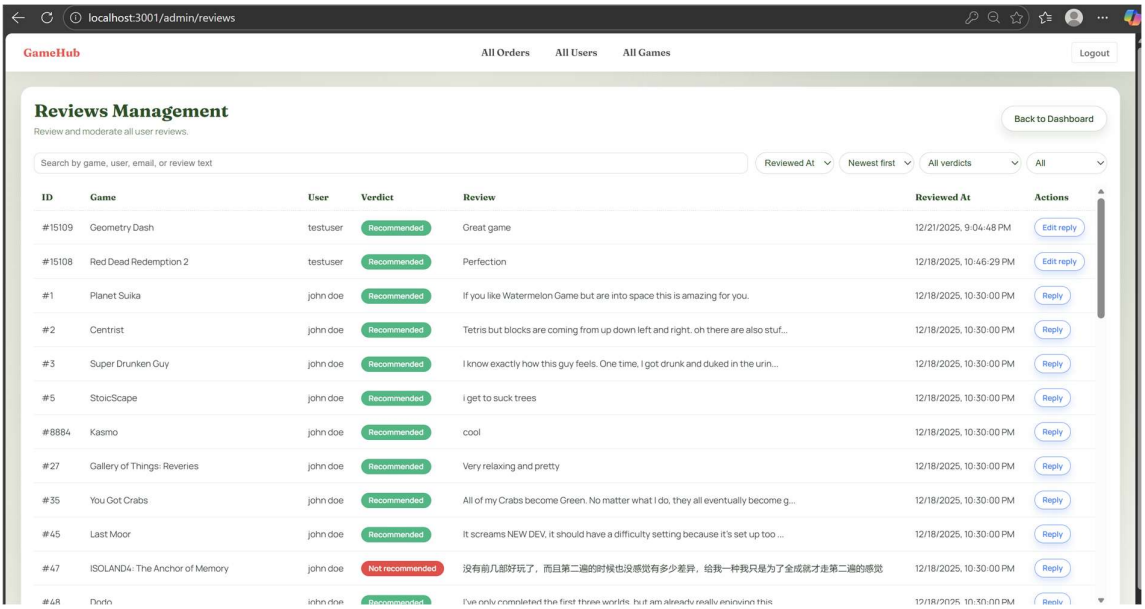


Figure 40: Management reviews

URL: http://localhost:3001/admin/reviews

The **Reviews Management** page supports moderation of user reviews by allowing administrators to view, filter, and respond to feedback in order to maintain content quality.

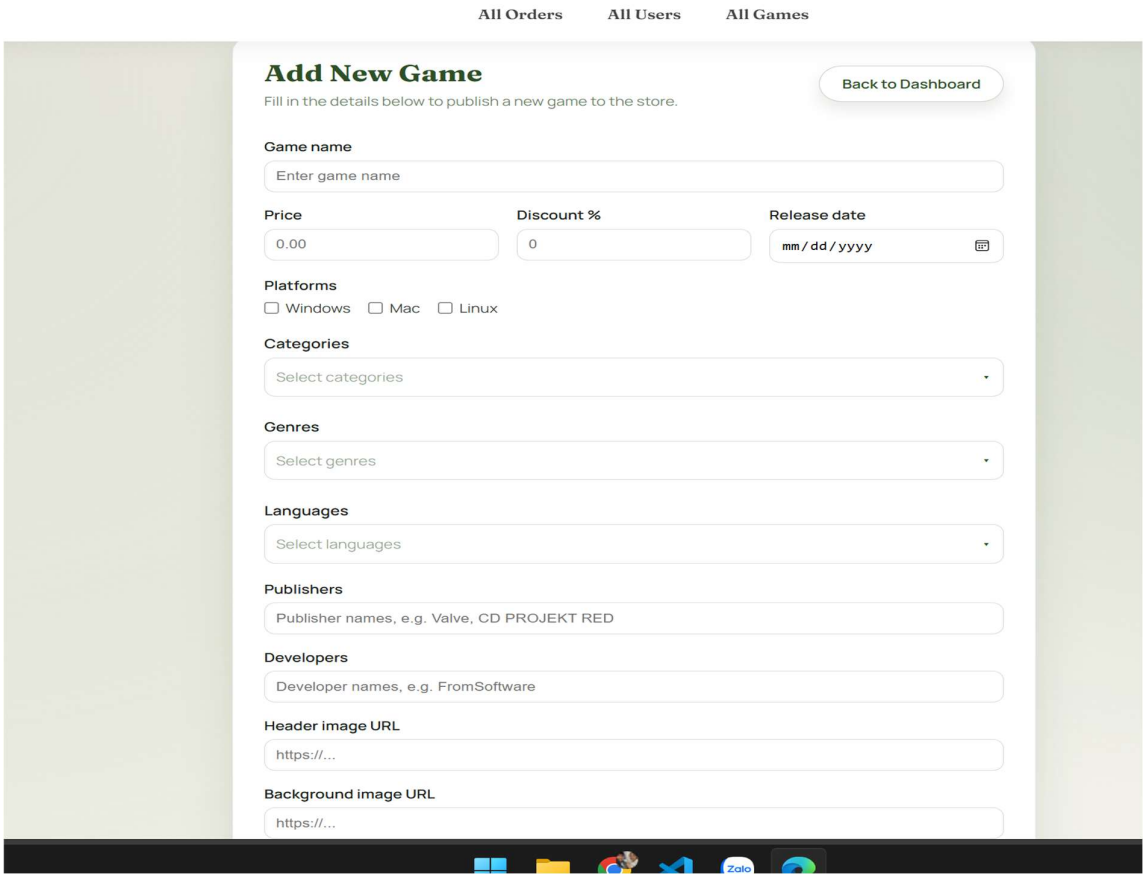


Figure 41: Add new game

URL: http://localhost:3001/admin/games/add

The **Add New Game** page enables administrators to add new games to the platform by entering complete game information before publishing them to the store.

The form includes fields for game name, price, discount, release date, supported platforms, categories, genres, languages, publishers, and developers, as well as image URLs for visual display. This page enables admins to efficiently create and manage game listings, ensuring that all necessary details are provided before a game is made available on the platform.

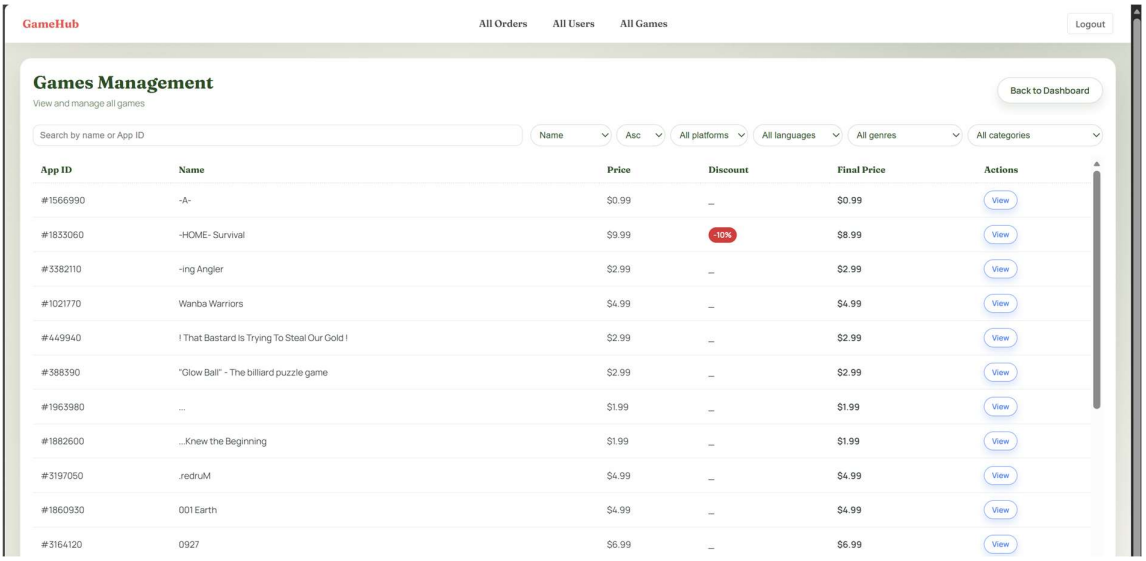


Figure 42: Management Game

URL: http://localhost:3001/admin/games

The **Games Management** page allows administrators to manage the game catalog by viewing, searching, and updating game information, including pricing and discounts.

Games are listed in a table showing key details such as app ID, game name, original price, discount, and final price, along with an action button to view detailed information. The page provides search, sorting, and filtering options by name, platform, language, genre, and category, helping admins quickly locate specific games

4.4. API of Admin

Table 12: Admin API List

Method	Endpoint	Description
POST	/api/admin/login	Admin login (creates session)
GET	/api/admin/stats	Dashboard totals: orders/users/games
GET	/api/admin/recent-orders	Recent orders for dashboard
GET	/api/admin/reviews/recent	Recent reviews for dashboard
GET	/api/admin/orders	List all orders (supports limit/offset/sort)
GET	/api/admin/users	List all users (supports limit/offset/sort)
GET	/api/admin/reviews	List reviews (search/filter/sort)
PUT	/api/admin/reviews/:id/reply	Create/update admin reply to review
GET	/api/admin/games	List games (search/filter/sort)
POST	/api/admin/games	Create new game
PUT	/api/admin/games/:id	Update game
GET	/api/admin/games/:id	Get game detail
GET	/api/admin/payments	List all payments

GET	/api/admin/payments/pending	List pending payments
PUT	/api/admin/payments/:id/status	Update payment status + order status

CHAPTER 5
TEST CASE

Project Name	GameHub Website
Priority	High
Description	End-to-end functional testing for all pages, ensuring proper behavior, integration, and responsiveness.
Test Objective	Validate UI components, backend integration, response message
Test Execution Date	22/12/2025

Use Case 1:
Name: Register new user
Identifier UC1
Inputs:
1. Registration information (username, email, password, optional profile fields)
Outputs:
1. Welcome message and redirect to profile/dashboard [If success]
2. Registration page and error message (validation or duplicate) [If fail]
Basic Course

Table 13: Register New User Test Case

Actor: User	System
1. Open registration page	1.1. Display the thread creation section
2. Enter registration information and submit	
3. Submit	3.1. Validate input (email format, password strength, unique username/email) 3.2. If registration success, create user record, send verification (if enabled), show success message and redirect. 3.3. If fail, display error message and keep form filled for correction.

Precondition
1. No account exists with the provided email/username.
Post condition
1. New user record created in `users` table and an optional verification token is stored.
User story: As a new user, I want to register an account so I can participate in the hub and buy games.

Use Case 2:
Name: Login
Identifier UC2
Inputs:
1. Credentials (username or email, password)
Outputs:
1. Access to protected pages and success message [If success]
2. Login page and error message [If fail]

Basic Course

Table 14: Login Test Case

Actor: Visitor / User	System
1. Open login page	1.1. Display login form
2. Enter credentials and submit	
3. Submit	3.1. Validate credentials against stored password_hash 3.2. If authentication success, create session, set session cookie, and redirect to intended page with success message. 3.3. If fail, show error message (invalid credentials) and allow retry.

Precondition
1. User account exists and is not locked.
Post condition
1. Session created and stored; user is authenticated for subsequent requests.
User story: As a registered user, I want to log in so that I can access my account and perform actions like buying games.

Use Case 3:
Name: Edit profile
Identifier UC3
Inputs:
1. Updated profile fields (display name, avatar, country, date_of_birth, preferences)
Outputs:
1. Updated profile displayed and success message [If success]
2. Profile edit page and error message [If fail]
Basic Course

Table 15: Edit Profile Test Case

Actor: User	System
1. Navigate to profile settings	1.1. Display current profile data in editable form
2. Update desired fields and submit	
3. Submit	3.1. Validate inputs (e.g., image type/size for avatar) 3.2. If update success, save changes to `user_profiles` / `users`, return success message and display updated profile. 3.3. If fail, show error message and keep input for correction.

Precondition
1. User is authenticated.
Post condition
1. Profile data updated in database and reflected in the UI.
User story: As a user, I want to update my profile information so other users see my latest details.

Use Case 4:
Name: Change password
Identifier UC4
Inputs:
1. Current password, new password, new password confirmation
Outputs:
1. Success message and enforced re-login (optional) [If success]
2. Password change page and error message [If fail]
Basic Course

Table 16: Change Password Test Case

Actor: User	System
1. Open change-password form in account settings	1.1. Display form fields
2. Enter current password and new password, confirm new password	
3. Submit	3.1. Verify current password matches stored hash 3.2. Validate new password strength and confirmation 3.3. If change success, update `password_hash` in DB, optionally invalidate existing sessions, and display success message. 3.4. If fail (wrong current password or weak new password), display error message.

Precondition
1. User is authenticated and knows current password.
Post condition
1. User password updated in database; sessions may be invalidated per security policy.
User story: As a user, I want to change my password to improve account security or recover from exposure.

Use Case 5:
Name: Add to cart
Identifier UC4
Inputs:
1. Game identifier (`app_id`)
Outputs:
1. Item appears in cart and success message [If success]
2. Product page and error message [If fail]
Basic Course

Table 17: Add to Cart Test Case

Actor: User	System
1. View game detail page	1.1. Display 'Add to cart' control and availability
2. Click 'Add to cart'	
3. Submit	3.1. Verify game exists and is purchasable

	3.2. If cart exists for user/session, add item to `cart_items`; otherwise create cart and add item. Update cart totals. 3.3. If success, show cart summary and success message; if fail, show error (out-of-stock, DB error).

Precondition
1. Game exists in `games` table
Post condition
1. `cart_items` updated and `carts.total_price` recalculated; item visible in user's cart session.
User story: As a customer, I want to add a game to my shopping cart so I can purchase it later.

CHAPTER 6

DISCUSSION AND EVALUATION

6.1. Discussion

This project implements a full-stack e-commerce platform focused on digital game distribution. The system is organized as a traditional web application using a React single-page application (SPA) frontend and an Express backend with PostgreSQL as the primary datastore. Key design goals were simplicity, clear separation of concerns (presentation, controller, data), and ease of local development.

Architecture and design decisions:

- Frontend: React with context APIs (AuthContext, CartContext, WishlistContext) to manage client-side state and reduce round-trips for common UI interactions. React was chosen for its ecosystem and developer productivity.
- Backend: Express for lightweight routing and middleware composition; controllers are organized per feature (auth, games, cart, orders). The server exposes RESTful endpoints under the /api/* prefix and performs server-side session management.
- Sessions: Server-side sessions stored in PostgreSQL using connect-pg-simple and express-session. Storing sessions server-side simplifies session invalidation and reduces client-side token handling complexity.
- Data layer: PostgreSQL stores games (keyed by Steam app_id), game descriptions/specs, users, carts, orders, payments, and supporting reference tables. The schema includes normalized many-to-many relationships for genres/developers/languages, and specialized tables for user libraries and wishlist items.
- Authentication & security: Passwords are hashed with bcrypt; session cookies are used with a configurable SESSION_SECRET. CORS is configured to allow common dev origins and the FRONTEND_URL environment variable.

Challenges and trade-offs

- Server-side sessions vs JWT: Server-side sessions simplify revocation and session storage but require a persistent session store and add server-side state that must be scaled or shared across instances (solved by centralizing in Postgres). JWT would reduce server state but complicate revocation and require more careful token lifecycle management.
- Query layer: The project mixes direct SQL helpers and Sequelize models. Raw queries can be more performant and predictable for complex queries (e.g., full-text search),

while Sequelize increases developer ergonomics for simple CRUD and associations. The trade-off is maintainability vs performance in hotspots.

- Data import & freshness: The project imports game data from CSVs. This is easy for bootstrapping but can introduce stale data and requires processes for updating, normalizing, and maintaining referential integrity (developer/publisher/genre link tables).
- Scalability: The current architecture is horizontally scalable for stateless API logic, but sessions and DB connection pools need proper configuration when scaling to multiple backend instances. Database performance at scale will require indexes (some are included) and potential read-replicas or caching for high-read endpoints (e.g., listings/search).

6.2. Comparison

This section compares key architectural choices against plausible alternatives and highlights why the chosen approach fits the project goals.

1. **Session management**

- Chosen: Server-side sessions in PostgreSQL (connect-pg-simple).
    - Pros: Easy session invalidation, centralized storage, less exposure of auth tokens to clients, straightforward cookies-based auth for browser clients.
    - Cons: Server-side state to manage; DB becomes critical for auth; requires shared session store across instances.
- Alternative: JWT (stateless tokens)
    - Pros: Truly stateless servers, easier horizontal scaling for pure API services.
    - Cons: Revocation complexity, potential long-lived tokens if not implemented carefully, need to secure tokens on client.

2. **Data layer: Raw SQL + Sequelize**

- Chosen: Combination — raw query helpers for complex queries and some Sequelize models for convenience.
    - Pros: Raw SQL provides fine-grained control and performance; Sequelize simplifies modeling relations and migrations.
    - Cons: Mixed styles increase cognitive load and require developers to know both patterns.
- Alternative: Full ORM (Sequelize only) or Full query builder (Knex/pg)

- Full ORM simplifies consistency and model-level constraints, while a query builder plus raw SQL is often preferred for complex, optimized queries.

3. **Frontend state management**

- Chosen: React Context + local component state.
  - Pros: Minimal dependencies, easy to understand; suitable for small-to-medium apps.
  - Cons: Context can cause unnecessary re-renders for large trees and may need memoization or state libs for heavy use.
- Alternative: Redux/MobX
  - Useful for very large apps with complex cross-cutting state; adds boilerplate and learning curve.

4. **API patterns: Monolith REST vs Microservices**

- Chosen: Single monolithic backend (feature-organized).
  - Pros: Easier local development and deployment, low operational overhead.
  - Cons: A monolith needs proper boundaries as it grows; complicates independent scaling of different services (e.g., payments vs catalog).
- Alternative: Microservices separation (catalog, auth, payments)
  - Makes scaling and independent deployment easier but increases operational complexity and infrastructure.

6.3. Evaluation

The application meets the goals of an MVP e-commerce platform: core flows (authentication, browsing, cart, checkout, order recording and user library) are implemented, the data model is normalized and expressive, and developer ergonomics (Express controllers, clear routes, React contexts) make iterating on features straightforward. Using PostgreSQL as both the primary datastore and session store simplifies deployment and session revocation. For a small-to-moderate user base the architecture is practical and maintainable; however, production readiness requires targeted improvements in security, testing, and scaling.

Limitations:

1. Security / SSL and cookie hardening

- Limitation: Development settings currently allow insecure cookies (httpOnly/secure=false) and CORS is permissive. TLS is not enforced by the app.
- Mitigation: Enforce TLS (HTTPS) at ingress, set cookie.secure = true and httpOnly = true in production, tighten sameSite, restrict CORS to trusted origins, rotate SESSION_SECRET, and run regular dependency vulnerability scans.

2. Session scaling and persistence
    - Limitation: Server-side sessions live in PostgreSQL — good for correctness but can become a bottleneck at scale and requires connection pooling and centralized state.
    - Mitigation: Use a dedicated session cache (Redis) for low-latency session reads/writes or ensure robust DB pooling; alternatively use sticky sessions behind a load balancer if Redis is not available.
3. Database scaling and connection pooling
    - Limitation: Single Postgres instance and default pool settings may exhaust connections under load; heavy read traffic (catalog/search) can overload DB.
    - Mitigation: Tune PG pool sizes, add read replicas for read-heavy workloads, introduce caching (Redis or CDN) for listings/search, and use indexes and query optimizations for hotspots.
4. Load balancing & horizontal scaling
    - Limitation: Running multiple backend instances requires shared session store and careful management of DB connections and static assets.
    - Mitigation: Deploy behind a load balancer (e.g., Nginx, cloud LB), centralize sessions (Redis) or use sticky sessions, and add autoscaling + health checks (use /api/health).

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1. Conclusion

This Game Shop project delivers a practical, well-structured MVP for a digital games e-commerce platform. The stack (React frontend, Express backend, PostgreSQL) and the modular code organization enabled the rapid development of core flows, including user authentication with OTP verification, dynamic game catalogs, and a secure checkout process. By utilizing a decoupled architecture, the system ensures that the presentation layer remains independent of the business logic, allowing for seamless updates and maintenance.

The implementation of the Model-View-Controller (MVC) pattern provided a clear separation of concerns:

- The View (React) manages complex frontend states through context-driven logic, ensuring a responsive user experience.
- The Controller (Node/Express) secures the application through role-based access control and middleware, effectively managing the "Security Handshake" required for user verification.
- The Model (PostgreSQL) maintains a normalized database schema that ensures data integrity across games, orders, and user profiles.

Beyond the core functionality, the system's design emphasizes extensibility. The normalized database structure and RESTful API design make it straightforward to integrate future features such as advanced analytics, a community forum, or multi-currency payment gateways. Ultimately, this project serves as a robust foundation for a scalable e-commerce solution, demonstrating a professional approach to full-stack development and system integration.

7.2. Future work

1. Security Hardening

- Encrypted Communications: Implementation of secure TLS/SSL termination to encrypt all data in transit between the client and the server.
- Secure Session Management: Hardening of cookie settings (e.g., SameSite, Secure, and HttpOnly flags) to prevent Cross-Site Scripting (XSS) and Request Forgery (CSRF) attacks.

- Enhanced Authentication: Transitioning from basic OTP to Two-Factor Authentication (2FA) via authenticator apps and implementing more granular activity logs for administrative oversight.

2. Operational & DevOps Excellence

- Standardized Data Patterns: Refining the Model layer to use standardized data access patterns, ensuring consistent performance and easier debugging across the codebase.
- CI/CD Pipeline: Establishing Automated Testing (Unit, Integration, and E2E) and Continuous Integration/Continuous Deployment (CI/CD) pipelines to ensure code quality and rapid, reliable releases.
- Monitoring & Health Checks: Implementing a Load Balancer with integrated health checks to monitor server status and automatically redirect traffic away from failing nodes.

3. Scalability & Performance Optimization

- Distributed Session Storage: Moving session management from local memory to a dedicated Redis cache, allowing the application to scale horizontally across multiple server instances.
- Database Scaling: Optimizing PostgreSQL performance through DB connection pooling and the implementation of read replicas to handle high volumes of concurrent queries without bottlenecking the primary database.
- Global Content Delivery: Integrating a Content Delivery Network (CDN) to serve game assets (images, videos, and binaries) with lower latency to a global user base.

REFERENCES

1. *PostgreSQL*. (2025, December 22). PostgreSQL. https://www.postgresql.org/
2. *Node.js — Run JavaScript everywhere*. (n.d.). https://nodejs.org/en
3. *React*. (n.d.). https://react.dev/
4. *Nodemailer | Nodemailer*. (n.d.). https://nodemailer.com/
5. *Steam Store*. (n.d.). https://store.steampowered.com/
6. Brian. (2024, March 29). *How to write an ecommerce Website Requirements Document: Tips, tricks, and best practices*. DEV Community. https://dev.to/brianporter/how-to-write-an-ecommerce-website-requirements-document-tips-tricks-and-best-practices-4c4d
7. *Ecommerce website design: Examples and tips (2026)*. (2025, December 10). Shopify. https://www.shopify.com/blog/best-ecommerce-sites