**LuxuryStep Software Architecture Documentation**

**Summary**

**1.Conceptual Model**

**2.SAD:**

* **1.Product Overview**
* **2.Architectural Models**
* **6.Mapping Between Models**
* **7. Architectural Design Rationale**

**3.DDD:**

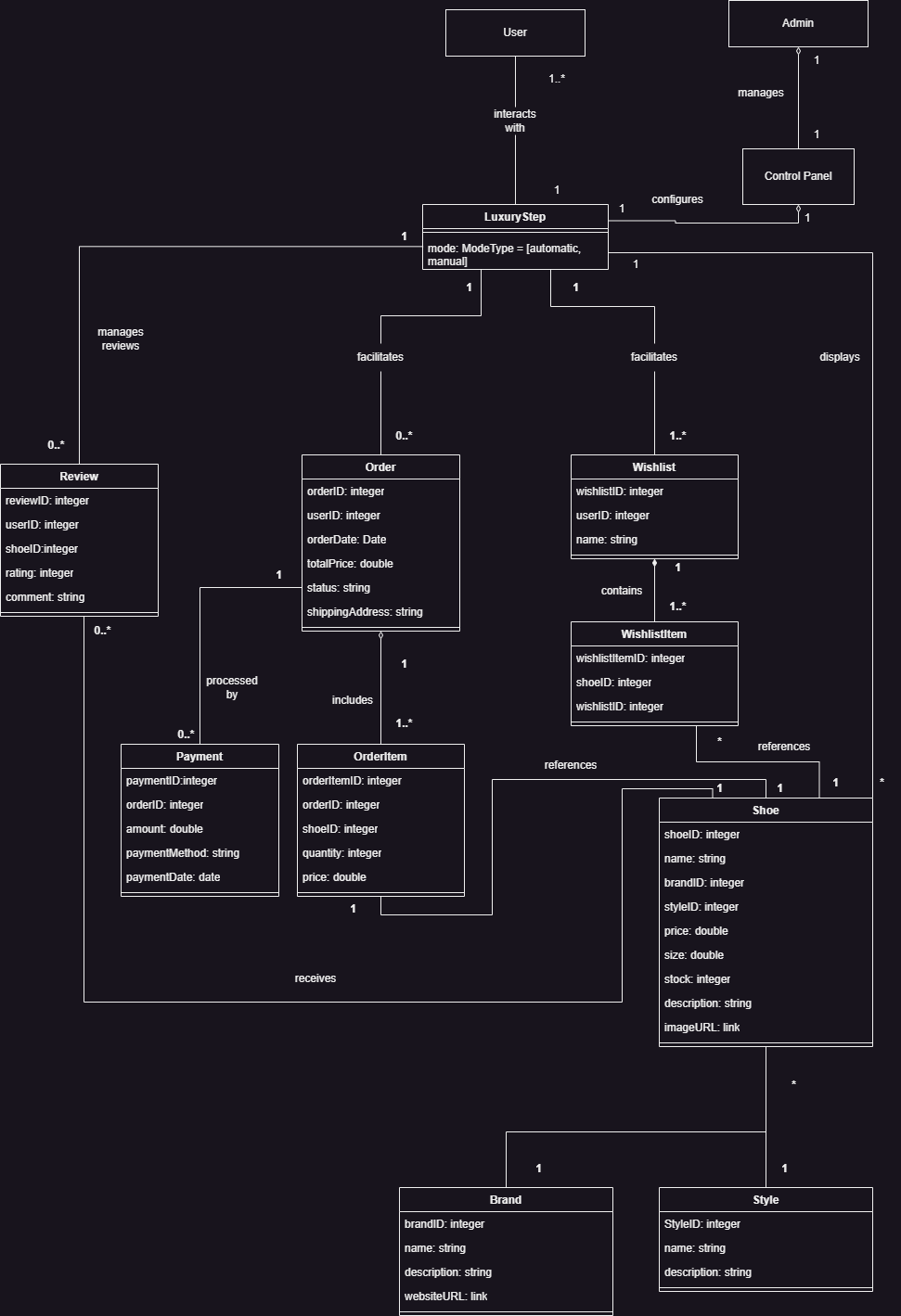
* **Introduction**
* **Mid-Level Design**
* **Low-Level Design**
* **Sequence Diagrams**
* **Design Rationale**

**4.Utility Tree**

**5.Design Patterns(Mid-Level & Low-Level explained thoroughly)**

* **Factory Design Pattern**
* **Singleton Design Pattern**
* **Lazy Loading (Caching Mechanism in Singleton Managers)**
* **Repository Design Pattern (Implicitly)**
* **Dependency Injection (Factory Integration)**
* **Active Record Pattern**

**1.CONCEPTUAL MODEL**

****

**2.SAD**

**1.0 Product Overview for Shoe Review**

**Product Vision**

The Shoe Review Website is a specialized e-commerce platform designed to provide a seamless experience for shoe enthusiasts and shoppers. It combines the functionality of a traditional e-commerce platform with a review and rating system, allowing users to browse, purchase, and review various types of shoes. The platform also supports personalized wishlist management and administrative controls for shoe review moderation.

**Key Features:**

1. **Comprehensive Shoe Catalog**:
   * Offers a diverse selection of shoes categorized by brand, style, size, and price.
   * Detailed descriptions, stock information, and high-quality images for each product.
2. **Wishlist Management**:
   * Users can create and manage wishlists for desired shoes.
   * Ability to save shoes for future purchase or reference.
3. **Order Placement and Tracking**:
   * Allows users to add shoes to their cart, place orders, and track order status.
   * Features real-time stock updates and dynamic pricing calculations.
4. **User Reviews**:
   * Enables users to rate and write reviews for purchased shoes.
   * Helps other users make informed buying decisions based on reviews and ratings.
5. **Administrative Control Panel**:
   * Allows administrators to moderate shoe reviews and ensure the integrity of the review system.
   * Includes functionality for managing products and user feedback.

**Stakeholders:**

* **End Users**:
  + Primary users who browse, purchase, and review shoes.
  + Create personalized wishlists and place orders.
* **Shoe Brands**:
  + Partners promoting their products through the platform.
  + Benefit from user reviews and visibility.
* **Platform Administrators**:
  + Oversee and manage user interactions, reviews, and products.
* **Developers**:
  + Maintain the platform and enhance its features for scalability and usability.

**Target Market:**

* **General Consumers**: Individuals looking to purchase shoes online conveniently.
* **Shoe Enthusiasts**: People passionate about shoe collections, styles, and brands.
* **Retailers & Brands**: Seeking to enhance their digital presence and reach a broader audience.

**Technical Highlights:**

* **Dynamic Database Design**:
  + Relationships between users, reviews, orders, and shoes ensure consistent and reliable data management.
  + Modular architecture that allows for scalability and easy maintenance.
* **Interactive User Interface**:
  + Designed to be responsive and intuitive, offering a seamless browsing and shopping experience.
  + Integration of modern web technologies for enhanced performance.
* **Secure Transactions**:
  + Uses secure payment gateways and encrypted data storage to protect user information.

### 2. Architectural Models

This section provides the specification of architectural models for the Shoe Review Website, focusing on both static and dynamic aspects.

### 2.1.1 Layered Architecture Overview

The Shoe Review platform adopts a **Model-View-Template (MVT)** architecture, facilitated by Django REST Framework. This architecture ensures separation of concerns, enabling better modularity and maintainability:

* **Model**: Represents the database schema and handles business logic, such as entities for Shoe, Review, Order, and Wishlist.
* **View**: Processes incoming requests, interacts with models, and prepares responses.
* **Template**: Manages the presentation layer, delivering dynamic content rendered on the frontend.

This architecture provides clear boundaries between data management, business logic, and user interface, allowing for flexibility and scalability.

### 2.1.2 System Modules

The platform is structured into the following modules, each responsible for specific functionality:

**User Management**

**Key Functionalities**:

* **Authentication and Authorization**: Utilizes token-based mechanisms, such as **JWT** or **SimpleTokenAuthentication**.
* **User Profiles**: Includes management of user-specific data like wishlists and reviews.

**API Methods for User Management**

* **POST /api/users/register/**:
  + Registers a new user.
  + Validates fields like username, email, and password.
* **POST /api/users/login/**:
  + Generates an authentication token upon successful login.
  + Returns a JWT token or session token.
* **GET /api/users/cart/**:
  + Retrieves the logged-in user’s cart data.
  + Requires authentication via token in the header.
* **PUT /api/users/ cart /**:
  + Updates user cart information (e.g., name, email).

### ****Shoe Catalog Management****

**Key Functionalities**:

* **CRUD Operations**: Allows administrators to add, update, or delete shoe records.
* **Categorization**: Shoes are organized by brand, style, and category for user convenience.

#### ****API Methods for Shoe Catalog****

* **GET /api/shoes/**:
  + Fetches a list of all shoes.
  + Supports filters like brand, style, category.
  + Example Query: /api/shoes/?brand=Nike&style=streetwear
* **GET /api/shoes/<id>/**:
  + Retrieves details of a specific shoe by ID.
* **POST /api/shoes/**:
  + Adds a new shoe (**admin-only access**).
* **PUT /api/shoes/<id>/**:
  + Updates an existing shoe record.
* **DELETE /api/shoes/<id>/**:
  + Deletes a shoe record (**admin-only access**).

### ****Review Management****

**Key Functionalities**:

* Users can submit reviews only for purchased shoes.
* Validation mechanisms ensure authenticity by linking reviews to completed orders.

#### ****API Methods for Review Management****

* **GET /api/reviews/**:
  + Fetches all reviews, optionally filtered by **shoe\_id** or **user\_id.**
* **POST /api/reviews/**:
  + Allows users to submit a review.
* **PUT /api/reviews/<id>/**:
  + Allows users to edit their own review(technically they cant but they can leave a new review after deleting their previous review).
* **DELETE /api/reviews/<id>/**:
  + Deletes a review.

### ****Order Processing****

**Key Functionalities**:

* Handles shopping cart management, order placement, and stock updates.
* Ensures secure payment processing and order tracking.

#### ****API Methods for Order Processing****

* **POST /api/orders/**:
  + Places a new order.
* **GET /api/orders/<id>/**:
  + Retrieves details of a specific order, including items and status.
* **PUT /api/orders/<id>/**:
  + Updates the status of an order (**admin-only**).
* **DELETE /api/orders/<id>/**:
  + Cancels an order(**admin-only**, after the user procceds to checkout they cannot cancel the order).

### ****Wishlist System****

**Key Functionalities**:

* Users can create and manage multiple wishlists.
* Shoes can be added to wishlists or moved to the shopping cart.

#### ****API Methods for Wishlist System****

* **GET /api/wishlists/**:
  + Retrieves all wishlists for the authenticated user.
* **POST /api/wishlists/**:
  + Creates a new wishlist.
* **PUT /api/wishlists/<id>/**:
  + Updates a wishlist’s name or details.
* **DELETE /api/wishlists/<id>/**:
  + Deletes a wishlist.

### ****Other Technical Details****

* **Validation**:
  + Uses serializers in DRF to handle data transformation and validation.
* **Permissions**:
  + Admin-specific endpoints require **Is\_Staff** permission.
  + Regular endpoints use **IsAuthenticated** or **AllowAny**.
* **Error Handling**:
  + Provides descriptive error messages ("**Failed to load data**").

### 2.1.3 Interaction with Other Layers

The platform layers interact in the following manner:

* **Templates** query the **Views**, which use **Models** to fetch and process data.
* The **API Layer** built using Django REST Framework allows frontend applications to communicate with the backend, supporting cross-platform compatibility.

### 2.1.4 Additional Dynamic Models

#### Login and Authentication Workflow:

* **Actors**: User, System.
* **Steps**:
  1. User submits login credentials.
  2. System validates credentials and generates a secure token (e.g., JWT).
  3. User accesses restricted resources using the token.

#### Error Handling Workflow:

* **Actors**: System.
* **Steps**:
  1. Detects issues such as stock unavailability, payment failures, or API timeouts.
  2. Logs errors and provides users with meaningful error messages.
  3. Automatically retries or suggests alternative solutions (e.g., contacting support).

### 2.2.1 Separation of Concerns

The use of the MVT architecture ensures that:

* **Models** manage the data layer, ensuring data consistency and integrity.
* **Views** handle business logic independently of the templates.
* **Templates** render dynamic content, focusing solely on the presentation layer.

### 2.2.2 Justifications for MVT

MVT was chosen because:

* It integrates seamlessly with Django REST Framework for building robust APIs.
* It enables rapid development using reusable components and conventions.

### 2.2.3 Deployment Design

* **Cloud Hosting**: The platform is deployed on AWS, which supports horizontal scaling for peak traffic.
* **Security**: Encryption methods (e.g., HTTPS and secure payment gateways) are used to protect sensitive data.
* **Load Balancing**: Ensures consistent performance during high traffic periods by distributing requests across multiple servers.

##### **2.2.4 Component Diagram**

This diagram highlights the modular architecture of the system.

* **Frontend**:
  + User Interface for browsing, managing wishlists, and placing orders.
  + Admin Interface for moderating reviews and managing products.
* **Backend**:
  + API layer for handling requests from the frontend.
  + Database interaction to manage and query entities.
* **Database**:
  + Centralized storage for all entities: Users, Reviews, Shoes, Orders, Wishlists.

**2.3 Simulation Layer Overview**

| **Component** | **Functionality** | **Interactions** | **Purpose** |
| --- | --- | --- | --- |
| **User Interaction Simulation** | Simulates user actions like browsing shoes, adding items to wishlist/cart, and placing orders. | - Mimics real user flows (e.g., adding items to wishlist, submitting reviews). - Interacts with Order, Review, and Wishlist entities without affecting live data. | Test user interactions and validate workflows without affecting live data. |
| **Stock Management Simulation** | Simulates real-time stock updates when items are added to cart or purchased. | - Decreases stock count when an item is purchased. - Verifies sufficient stock before purchase. - Simulates stock depletion scenarios. | Prevent overselling by simulating stock management under high traffic. |
| **Order Placement Simulation** | Simulates the order placement workflow (from cart to purchase). | - Simulates adding products to an order. - Processes payment, calculates total, creates Order and OrderItem records. - Updates stock accordingly. | Ensure correct order processing and stock updates during purchase. |
| **Review Simulation** | Simulates the review submission process for purchased shoes. | - Creates and links reviews to users and shoes. - Tests validation rules (e.g., reviewing only purchased items). | Validate the review submission workflow for authenticity and accuracy. |
| **Database Simulation** | Simulates interaction with a mock database. | - Mock data used for testing rather than live data. - Simulates creation, update, and deletion of entities like User, Order, Review, and Shoe. | Safely test database operations without affecting live data. |
| **Error and Exception Handling Simulation** | Simulates various system errors and exceptions. | - Injects error conditions like stock issues, payment failures, and network errors. - Verifies system’s graceful handling of errors. | Test system response to errors and ensure it behaves appropriately under failure conditions. |

**2.4 Simulation Layer Interfaces**

| **Interface** | **Component** | **Description** | **Methods/Functions** | **Data Flow** |
| --- | --- | --- | --- | --- |
| **User Interaction Interface** | User Interaction Simulation | Provides methods to simulate user actions like browsing, adding items to wishlist, placing orders, and reviews. | - addToWishlist(shoeID, userID) - placeOrder(userID, cart) - submitReview(userID, shoeID, review) | - Input: shoeID, userID, cart, review - Output: Simulated user interaction (added to wishlist, order placed, review submitted) |
| **Stock Management Interface** | Stock Management Simulation | Interfaces with the stock management system to update stock levels in real-time during order placement. | - updateStock(shoeID, quantity) - checkStock(shoeID) - simulateStockDepletion(shoeID) | - Input: shoeID, quantity - Output: Updated stock levels after purchase or simulation |
| **Order Placement Interface** | Order Placement Simulation | Simulates order processing by adding products to the order, processing payments, and creating OrderItem records. | - createOrder(userID, cart) - calculateTotal(cart) - confirmOrder(orderID) | - Input: userID, cart, orderID - Output: Order and OrderItem records created, updated total |
| **Review Submission Interface** | Review Simulation | Interfaces with the review system, ensuring only purchases are reviewed and associating reviews with Shoe. | - submitReview(userID, shoeID, rating, comment) - validateReview(userID, shoeID) | - Input: userID, shoeID, rating, comment - Output: Created and linked Review record |
| **Database Interface** | Database Simulation | Simulates interactions with the database, creating and updating records for entities like User, Order, etc. | - createRecord(entity, data) - updateRecord(entity, recordID, data) - deleteRecord(entity, recordID) | - Input: entity, data, recordID - Output: Simulated database changes (insert, update, delete) |
| **Error/Exception Handling Interface** | Error and Exception Handling | Provides methods to simulate error conditions like stock issues, payment failures, and network issues. | - simulateStockError(shoeID) - simulatePaymentFailure(userID, orderID) - simulateNetworkError() | - Input: shoeID, userID, orderID - Output: Simulated error conditions for testing |

**2.5 Simulation Layer Module Responsibilities**

The Simulation Layer is designed to emulate user interactions and system behaviors within the Shoe Review platform. It ensures workflows and functionalities are thoroughly tested without impacting live data. This layer interacts with mock data and components to simulate real-world scenarios, validate workflows, and identify potential issues.

| **Module** | **Functionality** | **Interactions** | **Purpose** |
| --- | --- | --- | --- |
| **User Interaction Simulation** | Simulates actions such as browsing shoes, adding items to wishlists or carts, and placing orders. | - Mimics user actions like adding items to wishlist or cart. - Interacts with Wishlist, Cart, and Order modules. | Validate workflows involving user interactions and ensure seamless navigation. |
| **Stock Management Simulation** | Simulates real-time stock updates during order placements. | - Decreases stock levels when items are purchased. - Verifies sufficient stock before allowing purchases. - Handles stock depletion scenarios during high traffic. | Ensure stock management operates correctly under dynamic conditions. |
| **Order Placement Simulation** | Emulates the entire order placement workflow from cart addition to checkout and confirmation. | - Simulates creating orders and linked order items. - Processes payment simulations and calculates totals. - Updates stock in the Shoe entity. | Test order processing workflow for accuracy and efficiency. |
| **Review Submission Simulation** | Validates the review submission process for purchased shoes. | - Tests eligibility checks (e.g., only purchased items can be reviewed). - Associates reviews with corresponding users and shoes. | Ensure the review system operates authentically and prevents fake reviews. |
| **Database Simulation** | Simulates interactions with a mock database for testing purposes. | - Mock data is used for CRUD operations on entities:User, Shoe, Order, Review, and Wishlist. | Safely test database functionality without impacting production data. |
| **Error and Exception Handling Simulation** | Tests the system’s response to errors and exceptions. | - Injects error scenarios like stock unavailability, payment failures, and network errors. - Verifies the system’s ability to recover gracefully from errors. | Validate robust error handling to ensure system reliability under failure conditions. |

**Purpose of the Simulation Layer**

**The simulation layer is critical for:**

1. Workflow Validation: Ensures end-to-end workflows operate as expected.
2. Error Identification: Exposes potential issues before deployment.
3. Safe Testing: Protects live data by using mock simulations.
4. Performance Analysis: Tests the system’s ability to handle high traffic or complex scenarios, such as concurrent orders or multiple wishlist modifications.

**2.6 User Interface Layer Structure**

The **User Interface (UI) Layer** of the **Shoe Review Website** is responsible for providing an intuitive and responsive interface for both users and administrators. Below is the structure of the UI Layer, detailing its key components, functionality, and interactions.

**The User Interface Layer enables:**

* Customers to browse products, manage wishlists, place orders, and submit reviews.
* Administrators to moderate reviews, manage products, and oversee platform operations.

**2.7 Components of the UI Layer**

| **Component** | **Description** | **Key Features** | **Interactions** |
| --- | --- | --- | --- |
| **Homepage** | The landing page for users, showcasing featured products and categories. | - Display featured shoes, categories, and offers. - Provide search and navigation options. | - Interacts with the Product Management backend to fetch featured shoes and categories. |
| **Product Listing Page(Gallery)** | Displays a list of shoes based on categories, styles, or search queries. | - Supports sorting and filtering options. - Displays product thumbnails, prices, and ratings. | - Fetches products from the backend. - Links to the Product Details Page for more information. |
| **Product Details Page(Gallery)** | Provides detailed information about a selected shoe. | - Displays product images, description, price, stock status, and reviews. - Option to add to wishlist or cart. | - Fetches product details and reviews. - Interacts with the Wishlist and Cart components. |
| **Wishlist Page** | Displays the user’s wishlist and allows them to manage items. | - List of saved items. - Option to move items to cart or remove them from wishlist. | - Interacts with the Wishlist Management backend to fetch and update wishlist data. |
| **Cart Page** | Displays items added to the cart and allows users to proceed to checkout. | - Displays cart summary and total cost. - Option to remove items or update quantities. - Checkout button. | - Interacts with the Order Management system to initiate the checkout process. |
| **Checkout Page**  **(Payment)** | Allows users to confirm their order and provide payment and shipping details. | - Collects shipping address and payment details. - Displays order summary. - Submit order button. | - Sends order data to the backend for processing. - Updates stock in the Stock Management system. |
| **Review Submission Page** | Enables users to write and submit reviews for purchased products. | - Form to submit ratings and comments. - Displays a list of eligible products for review. | - Validates user’s eligibility to review. - Interacts with the Review Management system. |
| **Control Panel** | Provides an interface for administrators to manage reviews, products, and users. | - Displays admin tools for moderating reviews, adding products, and managing users. | - Interacts with all backend modules for data management. |

**3. Navigation Flow**

* **Homepage**:
  + Users can browse featured shoes or navigate to categories and search results.
* **Product Listing Page**:
  + Users select a shoe, which directs them to the **Product Details Page**.
* **Wishlist Page**:
  + Users can add items to the cart or remove them.
* **Cart Page**:
  + Users proceed to the **Checkout Page** for payment and order confirmation.
* **Admin Dashboard/Control Panel**:
  + Admins can navigate to tools for managing reviews, products, and users.

**4. Interaction with Other Layers**

The User Interface Layer communicates with the backend layers (Business Logic Layer/Views and Data Layer/Models) through API’s to:

1. **Fetch Data**: Retrieves product information, user wishlists, and reviews.
2. **Submit Actions**: Sends user actions such as adding items to cart, placing orders, and submitting reviews.
3. **Receive Responses**: Displays feedback to the user, such as successful order placement or review submission.

**6. Mapping Between Models**

This section provides the relationship between the static and dynamic models, illustrating how entities interact within the system during different workflows.

#### ****6.1 Entity Relationship Mapping****

| **Entity** | **Static Model Description** | **Dynamic Model Interaction** |
| --- | --- | --- |
| **User** | Represents customers and administrators. Tracks interactions, orders, and reviews. | The User interacts with various workflows such as placing orders, creating wishlists, and submitting reviews. |
| **Shoe** | Core entity with attributes such as name, price, size, and stock. Associated with brand and style. | Shoes are added to the Wishlist and Order records, and they are reviewed by Users. |
| **Wishlist** | Represents a collection of shoes a user intends to buy. | Users add shoes to their wishlist, and a WishlistItem is created linking the user and shoe. |
| **Review** | Links users to specific shoes and includes attributes like rating, comment, and review date. | Users submit reviews for shoes after purchase. Each review is linked to a User and a Shoe. |
| **Order** | Represents the order a user places, containing items linked to specific shoes. | Once the user confirms a purchase, an Order and related OrderItems are created, linking the user to the shoes they ordered. |

#### ****6.2 Mapping Dynamic Model Workflow to Entities****

| **Workflow** | **Entities Involved** | **Static Model Link** |
| --- | --- | --- |
| **User adds a shoe to the wishlist** | User, Shoe, Wishlist, WishlistItem | The User adds a Shoe to their Wishlist, creating a WishlistItem record. |
| **Order placement process** | User, Order, OrderItem, Shoe | The User places an Order, which contains multiple OrderItem records linked to specific Shoes. |
| **User submits a review for a shoe** | User, Shoe, Review | After purchasing a shoe, the User creates a Review linked to both the Shoe and User. |

### ****7. Architectural Design Rationale****

This section explains the critical design decisions made during the development of the Shoe Review Website. It covers design choices that were difficult to implement, crucial to system performance, and hard to change once made. These decisions are central to the system’s functionality, scalability, and maintainability.

#### ****7.1 Key Design Decisions****

##### **7.1.1 Separation of Concerns**

* **Decision**: The system was divided into distinct modules (User, Product Management, Order Management, and Review Management).
* **Rationale**: This separation ensures each module focuses on a specific set of related functions, making the system easier to maintain and extend. It also allows for better scalability, as each module can evolve independently. For example, the Order Management module can be optimized without impacting the Review Management module.
* **Challenges**: Initially, there was concern about potential tight coupling between modules, especially between Order and Review entities. However, maintaining clear module boundaries helps in isolating failures and scaling individual modules.

##### **7.1.2 Database Normalization and Entity Relationships**

* **Decision**: The system uses a normalized relational database model to store data, linking entities like User, Shoe, Review, Order, and Wishlist.
* **Rationale**: This approach reduces redundancy and ensures data consistency across the platform. Each entity is linked through foreign keys, which provides a structured way to manage and query data. For example, reviews are linked to both users and shoes, ensuring reviews are always associated with the correct entities.
* **Challenges**: One of the difficulties was ensuring the relationships between User, Shoe, and Review remained consistent when a shoe was removed or updated. This required careful management of foreign key constraints to prevent orphaned records.

##### **7.1.3 Real-Time Stock Allocation**

* **Decision**: The system supports dynamic, real-time stock updates when a user places an order.
* **Rationale**: By ensuring that the available stock is updated in real time, the system prevents users from purchasing out-of-stock items. This decision is essential for user trust and operational efficiency.
* **Challenges**: Managing concurrent access to stock data posed a significant challenge, especially during high traffic periods. Implementing an efficient locking mechanism to prevent race conditions in stock updates required additional effort and testing.

#### ****7.2 Crucial Design Decisions****

##### **7.2.1 Review System Linked to Purchase**

* **Decision**: Users can only submit reviews for shoes they have purchased through the platform.
* **Rationale**: This ensures that reviews are relevant and trustworthy, as users can only review items they have experience with. It prevents fake or biased reviews, which is crucial for maintaining the credibility of the platform.
* **Challenges**: The complexity of enforcing this rule was a significant challenge, especially when dealing with users who might try to manipulate the system. This required integration between the Order and Review systems to validate that a review could only be submitted after an order had been completed.

##### **7.2.2 Flexible Wishlist System**

* **Decision**: Users can create multiple wishlists, and each wishlist can contain multiple shoes.
* **Rationale**: This gives users flexibility in managing their desired products. For example, a user might have one wishlist for summer shoes and another for formal shoes. This decision enhances user experience and increases engagement.
* **Challenges**: A key challenge was designing a system that allowed multiple, dynamically manageable wishlists for each user while maintaining performance and avoiding data redundancy.

#### ****7.3 Difficult and Crucial Decisions****

##### **7.3.1 Security and User Data Protection**

* **Decision**: The platform employs encryption for sensitive user data (e.g., passwords, payment information) and uses secure authentication methods (OAuth, token-based authentication).
* **Rationale**: Protecting user privacy and ensuring secure transactions is paramount. Given the platform's role in handling financial transactions and personal information, security was a priority throughout the design process.
* **Challenges**: Ensuring that the system remained secure against evolving threats required continuous monitoring and periodic updates to the security protocols. The difficulty was in designing a solution that balanced user convenience with robust security measures.

##### **7.3.2 Cross-Platform Compatibility**

* **Decision**: The platform is designed to be fully responsive, ensuring a consistent experience across mobile, tablet, and desktop devices.
* **Rationale**: Given the wide variety of devices users may access the platform on, ensuring that the system is compatible with all screen sizes is crucial for user experience. This decision is vital in attracting a broad audience.
* **Challenges**: Implementing a truly responsive design required careful testing across multiple browsers and devices, which added complexity to the design process.

**3.DDD**

# **1. Introduction**

The Shoe Review system is designed to facilitate the management of user reviews, wishlists, orders, and shoe inventory. This document provides a comprehensive design, detailing the system's structure, its main components, and their interactions. The goal is to offer a robust, scalable, and user-friendly platform for shoe enthusiasts.

# **2. Mid-Level Design**

This section describes the key classes in the system, their attributes, methods, and relationships.



## **Class Descriptions**

### User

\*\*Description\*\*: Represents a system user who can write reviews, place orders, and manage wishlists.

\*\*Attributes\*\*:

* - userId: Integer
* - name: String
* - email: String
* - password: String

\*\*Methods\*\*:

* - register()
* - login()

### Wishlist

\*\*Description\*\*: Represents a collection of shoes that a user saves for future reference.

\*\*Attributes\*\*:

* - wishlistId: Integer
* - userId: Integer
* - name: String

\*\*Methods\*\*:

* - addItem(shoeId: Integer)
* - removeItem(shoeId: Integer)

### Order

\*\*Description\*\*: Represents a purchase containing one or more items.

\*\*Attributes\*\*:

* - orderId: Integer
* - userId: Integer
* - orderDate: Date
* - totalPrice: Double

\*\*Methods\*\*:

* - placeOrder()
* - calculateTotal()

### Shoe

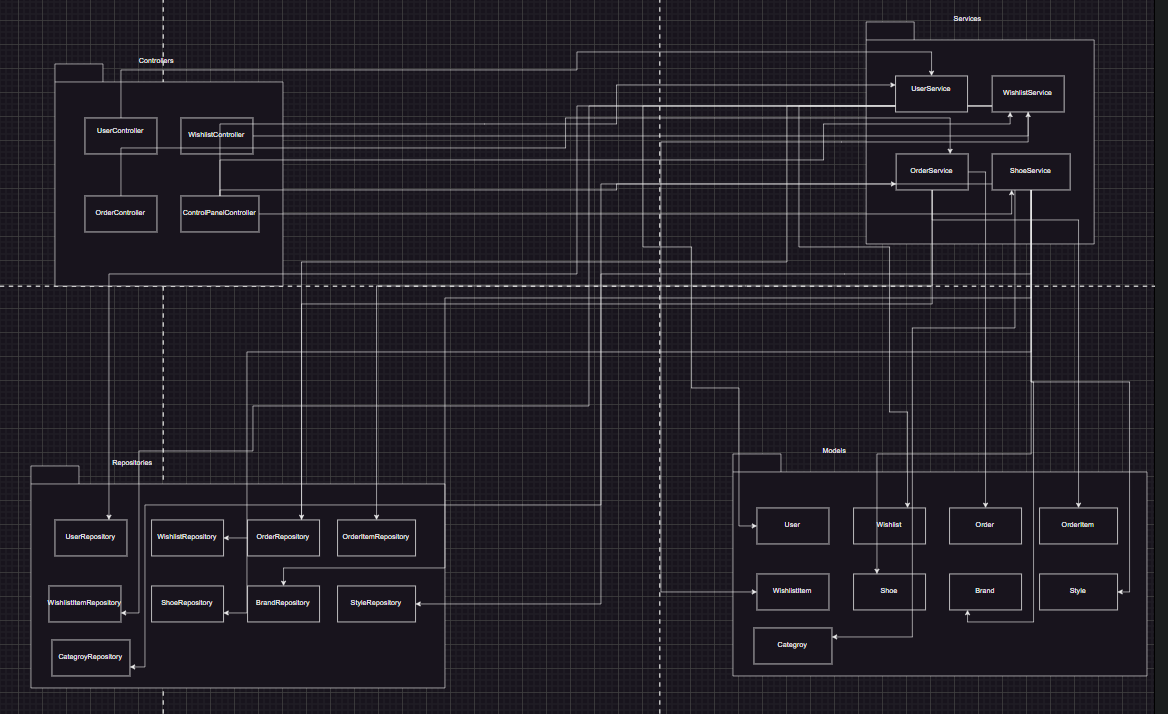
\*\*Description\*\*: Represents a shoe product available for purchase or wishlist.

\*\*Attributes\*\*:

* - shoeId: Integer
* - name: String
* - brandId: Integer
* - styleId: Integer
* - categoryId: Integer
* - price: Double
* - stock: Integer
* - description: String

# **3. Low-Level Design**

The Low-Level Design organizes the system into logical packages based on the Model-View-Controller (MVC) architecture. The Package Diagram below illustrates the organization and dependencies between these packages.



## **Package Responsibilities**

### Controllers Package

Manages user requests and delegates tasks to the Services layer.

Components:

* - UserController
* - WishlistController
* - OrderController
* - ControlPanelController

### Repositories Package

Interacts with the database to perform CRUD operations.

Components:

* - UserRepository
* - WishlistRepository
* - WishlistItemRepository
* - OrderRepository
* - ReviewRepository
* -PaymentRepository
* - OrderItemRepository
* - ShoeRepository
* - BrandRepository
* - StyleRepository

### Models Package

Represents the core entities in the system.

Components:

* - User
* - Wishlist
* - WishlistItem
* - Order
* - OrderItem
* - Shoe
* - Brand
* - Style
* -Payment
* -Review

**Services Package**

Implements business logic and processes data received from Views.

Components:

• - UserService

• - WishlistService

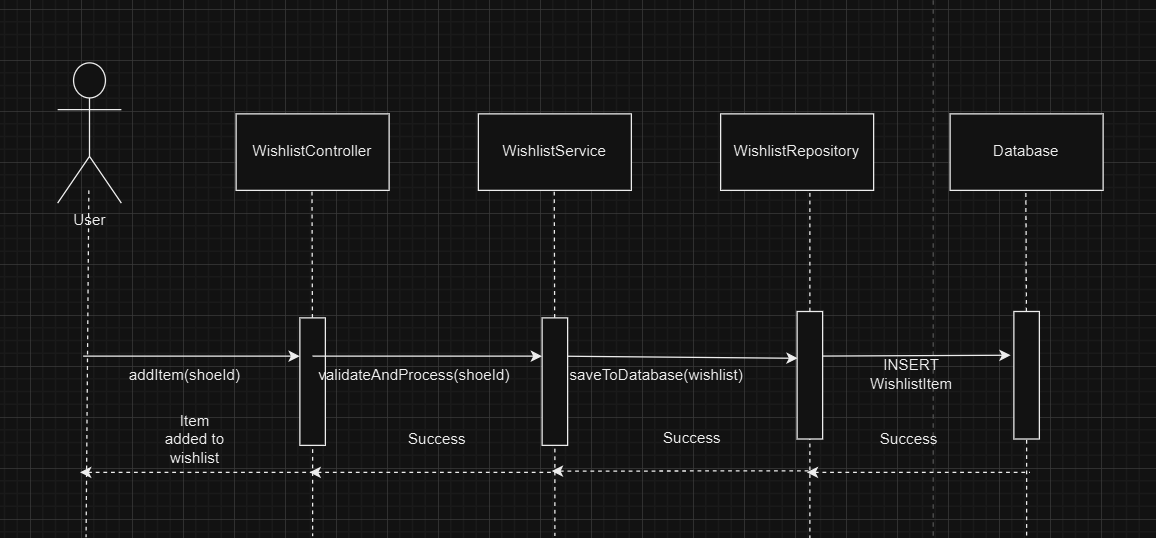
• - OrderService

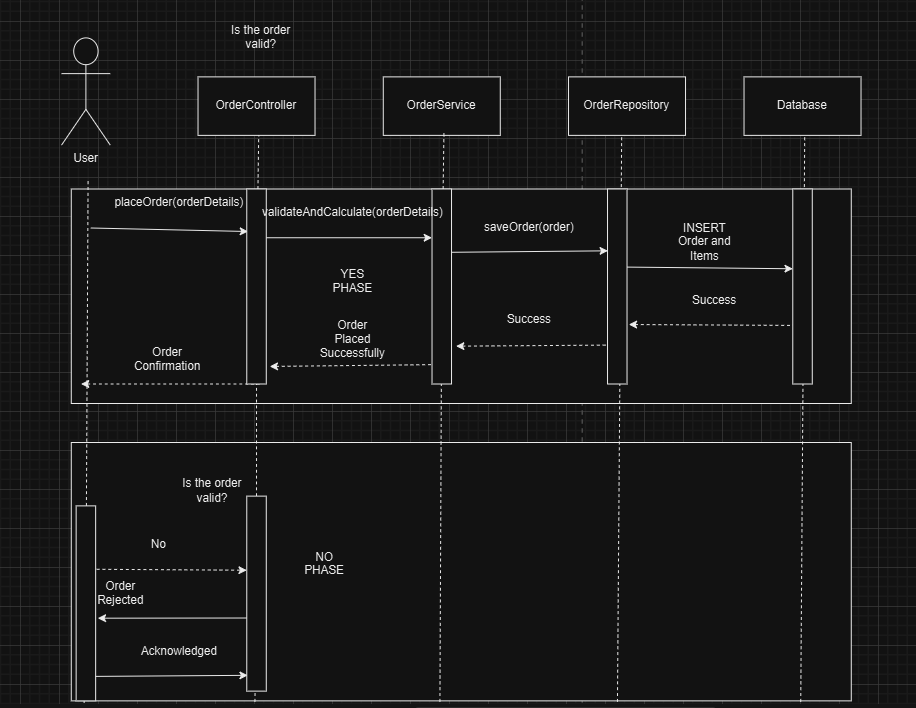
• - ShoeService

**4. Sequence Diagrams**

Sequence diagrams illustrate the interactions between system components during key workflows.

**4.1 Sequence Diagram WishList**



**4.2 Sequence Diagram OrderService**

# **5. Design Rationale**

The design of the Shoe Review system was guided by the following principles and considerations:

- Separation of concerns through the MVC architecture, ensuring maintainability and scalability.

- Flexibility in managing user preferences with distinct Wishlist and WishlistItem classes.

- Aggregation and composition relationships between entities for accurate data representation.

**4.UTILITY TREE**

**Usage**

* **Search Shoes** (H): Allow users to search for shoes by name, brand, style, or category.
* **Write Reviews** (M): Enable users to provide feedback on purchased shoes.
* **Purchase Shoes** (H): Ensure a seamless checkout and payment process.
* **Wishlist Management** (M): Allow users to add or remove shoes from their wishlist.

**Performance**

* **Fast Page Load** (H): Optimize loading times for search and product pages.
* **Recommendation System** (M): Recommend popular or related shoes to users based on tags or categories.

**Reliability**

* **Secure Transactions** (H): Provide encrypted and secure payment methods.

**Modifiability**

* **Discount System** (M): Easily add or update discount rules (“Buy 1, Get 1 Free”).
* **UI Customization** (M): Allow admins to customize the interface for promotions.

**Reusability**

* **Product API** (M): Use APIs to integrate with external systems for inventory updates or recommendations.
* **Review Analytics** (M): Provide summarized insights from user reviews for marketing purposes.

**Adaptability**

* **Multi-Currency Support** (L): Enable the platform to handle multiple currencies & payment methods.
* **Scalable Infrastructure** (M): Ensure the system supports growing users and inventory.

**Security**

* **Authentication System** (H): Ensure secure logins with password encryption and user roles.
* **Order Tracking** (M): Securely manage and display order history to users.

**5.Design Patterns**

**1. Factory Design Pattern**

The Factory Pattern is used to encapsulate the creation logic of objects, promoting loose coupling between client code and the classes being instantiated. In our project, various factory classes (UserFactory, BrandFactory, ShoeFactory, etc.) centralize object creation for models like User, Brand, Shoe, and others. This pattern improves testability, code reuse, and readability by abstracting and standardizing the instantiation process.

**Usage in Code:**

* Simplifies object creation with default values and ensures objects are instantiated consistently.
* Supports optional parameters and logic before saving objects to the database.

**2. Singleton Design Pattern**

The Singleton Pattern ensures that a class has only one instance and provides a global access point to it. This pattern is used in our ManagerSingleton classes (ReviewManagerSingleton, WishlistManagerSingleton, ShoeManagerSingleton) to manage shared resources like cache and ensure consistent behavior throughout the application.

**Usage in Code:**

* Provides a single shared instance of managers to coordinate operations like create, read, update, and delete (CRUD).
* Implements caching to optimize database queries by reducing redundant lookups.
* Prevents multiple instances of manager classes, ensuring centralized control.

**3. Lazy Loading (Caching Mechanism in Singleton Managers)**

Lazy Loading is a design pattern where object initialization is deferred until the object is actually needed. Our singleton managers implement a caching mechanism, storing objects in memory to avoid repetitive database queries. This improves application performance and reduces database load.

**Usage in Code:**

* Caches frequently accessed objects (reviews, wishlist items, shoes) in \_cache dictionaries.
* Fetches data from the cache if available, falling back to the database only when necessary.
* Enhances application responsiveness by reducing latency.

1. **Repository Design Pattern (Implicitly)**

The Repository Pattern separates the business logic from data access logic by providing a centralized way to interact with the data layer. While not explicitly named, our manager classes function as repositories by abstracting data operations on models.

**Usage in Code:**

* Encapsulates CRUD operations for models like Wishlist, Shoe, and Order.
* Provides methods like create\_, get\_, update\_, and delete\_ for each model, ensuring consistent data access across the application.

**5. Dependency Injection (Factory Integration)**

Dependency Injection is a design principle where dependencies (like related models) are provided to a class rather than being hardcoded. This principle is indirectly applied in the factory classes, where dependent objects (Brand for Shoe, Order for OrderItem) are passed as parameters to the factory methods.

**Usage in Code:**

* Factories like ShoeFactory and OrderItemFactory accept dependencies (Brand, Order) as parameters, promoting modularity and flexibility.
* Makes unit testing easier by allowing mock dependencies to be injected.

**6. Active Record Pattern**

The Active Record Pattern is a design pattern where objects are tied directly to database rows, and database operations (save, delete) are methods on those objects. In this code, Django ORM implements the Active Record Pattern, and factories utilize it to manage persistence.

**Usage in Code:**

* Each factory calls methods like .save() on model instances to persist data.
* Simplifies database interactions by abstracting SQL queries.