Court Kart: E-Commerce Platform

Design and Implementation Document

HADJ ARAB Adel

May 13, 2025

Abstract

This document presents the design and implementation of Court Kart, an e-commerce platform specializing in basketball equipment and apparel. The system is built using PHP, MySQL, and JavaScript, following the Model-View-Controller (MVC) architectural pattern. Key features include user authentication, product management, shopping cart functionality, order processing, and an administrative interface. The database implementation leverages advanced MySQL features including stored procedures and triggers to ensure data integrity and business rule enforcement. This document provides comprehensive details about the system architecture, database design, and implementation of required features.

Contents

1 Introduction

Court Kart is a comprehensive e-commerce platform specializing in sports equipment and apparel, with a primary focus on basketball products. The application follows modern web development patterns with a clean separation of concerns, using PHP for server-side processing, MySQL for database management, and JavaScript for client-side interactivity.

1.1 Project Overview

The Court Kart platform facilitates online shopping for basketball enthusiasts, offering features such as product browsing, detailed product views, user account management, shopping cart functionality, and secure checkout process. The platform also includes an administrative interface for managing products, orders, and users.

1.2 Technologies Used

The platform is built using the following technologies:

- **Backend:** PHP 8.1 with custom MVC framework
- **Database:** MySQL 8.0
- **Frontend:** HTML5, CSS3, JavaScript (ES6+)
- **Version Control:** Git
- **Additional Libraries:** Font Awesome for icons

Figure 1: Technology Stack Architecture

2 System Architecture

2.1 Architectural Pattern

Court Kart implements the Model-View-Controller (MVC) architectural pattern to ensure separation of concerns and maintainability:

- **Model:** Handles data logic and database interactions, encapsulating business rules and data access
- **View:** Renders the user interface using PHP templates with embedded HTML and CSS
- **Controller:** Processes user input, coordinates data flow between Model and View components
- **Core Components:** Provides fundamental functionality like routing, database connections, session management, and security

Figure 2: Model-View-Controller Architecture Diagram

2.2 Directory Structure

The application follows a well-organized directory structure that separates different concerns:

Figure 3: Project Directory Structure

```
/court-kart-store
  /config           # Application configuration files
    database.php    # Database connection settings
    app.php         # Application settings
  /src              # Application source code
    /Controllers    # Request handlers
    /Models         # Data models and business logic
    /Core           # Framework components
      Router.php    # URL routing system
      Database.php  # Database connection and query builder
      View.php      # Template rendering engine
      Session.php   # Session management
      Middleware.php # Request filtering
    /Services       # Business logic services
    /Helpers        # Utility functions
  /views            # UI templates
    /layouts        # Page templates and partials
    /shop           # Shop-related views
    /cart           # Shopping cart views
    /checkout       # Checkout process views
    /admin          # Admin interface views
    /auth           # Authentication views
    /errors         # Error pages
  /public           # Publicly accessible files
    /assets         # Static resources
      /css          # Stylesheets
      /js           # JavaScript files
      /images       # Image files
      /fonts        # Font files
  /routes           # Route definitions
    web.php         # Web routes
  /sql              # Database schema and scripts
    schema.sql      # Database tables
    triggers.sql    # Database triggers
    procedures.sql  # Stored procedures
  /docs             # Documentation files
    report.tex      # This design document
    images/         # Documentation images
```

Listing 1: Detailed Directory Structure

2.3 Request Lifecycle

The following sequence diagram illustrates the lifecycle of a typical request in the Court Kart application:

Figure 4: Request Lifecycle Sequence Diagram

3 Database Design

3.1 Entity-Relationship Diagram

The database schema is designed to support all e-commerce functionality while maintaining data integrity and relational constraints:

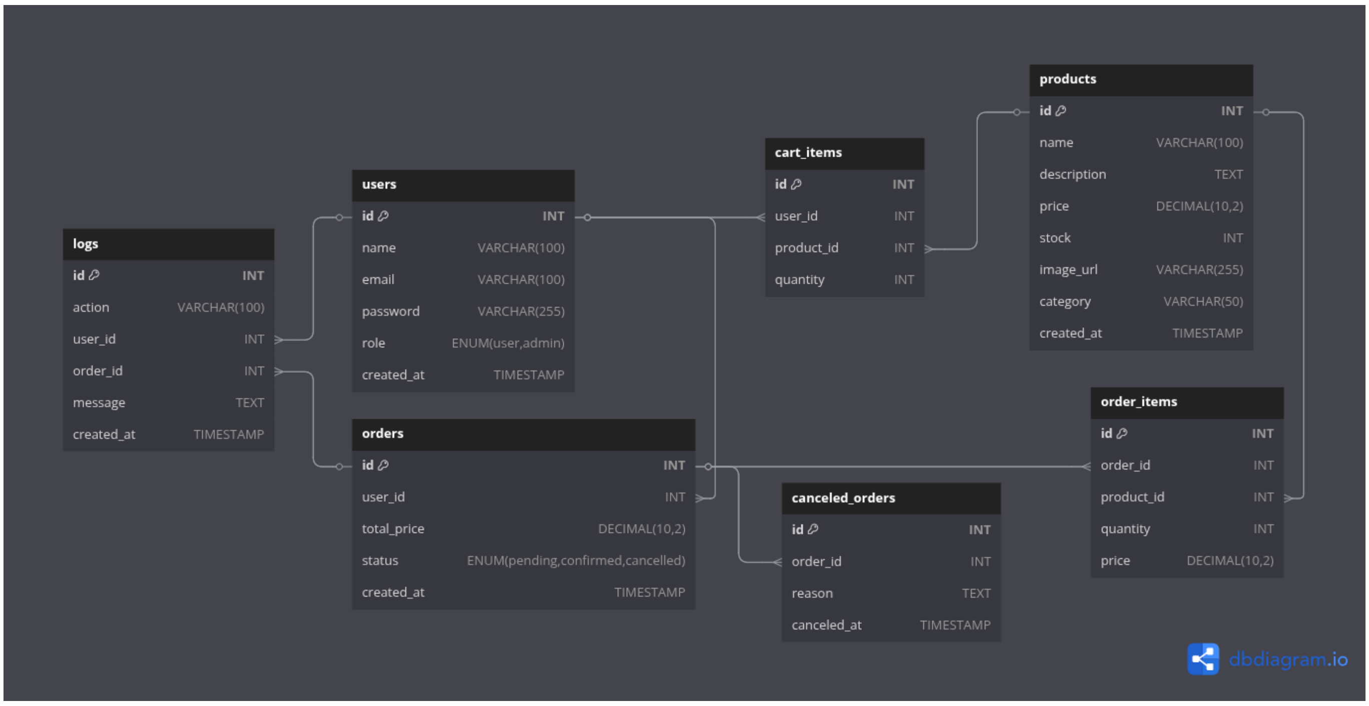

Figure 5: Entity-Relationship Diagram

3.2 Database Schema

Court Kart's database consists of several interconnected tables designed to support all e-commerce functionality:

Table 1: Database Schema Overview

Table Name	Primary Key	Description
users	id (INT)	Stores user account information including authentication data, profile details, and role designation (admin or regular user)
products	id (INT)	Contains comprehensive product details including name, description, pricing, stock levels, category, and image references
categories	id (INT)	Stores product categories for hierarchical organization of the product catalog
cart_items	id (INT)	Links users to products in their shopping cart with quantity information
orders	id (INT)	Maintains order headers with status tracking, user references, and timestamps
order_items	id (INT)	Contains line items associated with each order, including product information, quantity, and price at time of purchase
canceled_orders	id (INT)	Records canceled orders with reason information and timestamps for business analytics
logs	id (INT)	Comprehensive system activity tracking for auditing and debugging purposes

3.3 Table Relationships

The database implements the following relationships to maintain data integrity:

- **users → cart_items:** One-to-many (A user can have multiple items in their cart)
- **users → orders:** One-to-many (A user can place multiple orders)
- **products → cart_items:** One-to-many (A product can be in multiple users' carts)
- **products → order_items:** One-to-many (A product can be in multiple orders)
- **orders → order_items:** One-to-many (An order contains multiple products)
- **orders → canceled_orders:** One-to-one (A canceled order references exactly one order)
- **categories → products:** One-to-many (A category contains multiple products)

3.4 Detailed Table Specifications

users

```sql
CREATE TABLE users (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(255) NOT NULL UNIQUE,
    password VARCHAR(255) NOT NULL,
    role ENUM('user', 'admin') DEFAULT 'user',
    profile_image VARCHAR(255) NULL,
    remember_token VARCHAR(100) NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

products

```sql
CREATE TABLE products (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    description TEXT NOT NULL,
    price DECIMAL(10,2) NOT NULL,
    stock INT NOT NULL DEFAULT 0,
    category VARCHAR(50) NOT NULL,
    image_url VARCHAR(255) NOT NULL,
    is_new BOOLEAN DEFAULT FALSE,
    discount INT DEFAULT 0,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

orders

```sql
CREATE TABLE orders (
    id INT PRIMARY KEY AUTO_INCREMENT,
    user_id INT NOT NULL,
    total_price DECIMAL(10,2) NOT NULL,
    status ENUM('pending', 'confirmed', 'shipped', 'delivered', 'cancelled') DEFAULT 'pending',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);
```

order_items

```sql
CREATE TABLE order_items (
    id INT PRIMARY KEY AUTO_INCREMENT,
    order_id INT NOT NULL,
    product_id INT NOT NULL,
    quantity INT NOT NULL,
    price DECIMAL(10,2) NOT NULL,
    FOREIGN KEY (order_id) REFERENCES orders(id) ON DELETE CASCADE,
    FOREIGN KEY (product_id) REFERENCES products(id) ON DELETE NO ACTION
);
```

4 Stored Procedures Implementation

The database includes several stored procedures to handle complex business logic directly at the database level. This approach enhances performance, reduces network traffic, and centralizes business rule enforcement.

4.1 GetOrderDetails

This stored procedure retrieves comprehensive information about a specific order, including all ordered items, product details, and pricing information.

```sql
DELIMITER $$

CREATE PROCEDURE GetOrderDetails(IN p_order_id INT)
BEGIN
    -- Get order items with product details
    SELECT
        oi.order_id,
        oi.product_id,
        p.name AS product_name,
        oi.quantity,
        oi.price AS unit_price,
        (oi.quantity * oi.price) AS subtotal,
        p.image_url
    FROM
        order_items oi
    JOIN
        products p ON oi.product_id = p.id
    WHERE
        oi.order_id = p_order_id;
END$$

DELIMITER ;
```

Listing 2: GetOrderDetails Stored Procedure

Usage Example:

```sql
CALL GetOrderDetails(123);
```

4.2 FinalizeOrder

This stored procedure handles the checkout process by confirming the order and removing items from the customer's cart.

```sql
DELIMITER $$

CREATE PROCEDURE FinalizeOrder(IN p_order_id INT, IN p_user_id INT)
BEGIN
    DECLARE v_order_exists INT;

    -- Start transaction to ensure atomicity
    START TRANSACTION;

    -- Check if order exists and belongs to the user
    SELECT COUNT(*) INTO v_order_exists
    FROM orders
    WHERE id = p_order_id AND user_id = p_user_id;

    IF v_order_exists = 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Order not found or does not belong to this user';
        ROLLBACK;
    ELSE
        -- Update order status to 'confirmed'
        UPDATE orders
        SET status = 'confirmed'
        WHERE id = p_order_id;

        -- Clear cart items for this user
        DELETE FROM cart_items
        WHERE user_id = p_user_id;

        -- Log the order finalization
        INSERT INTO logs (action, user_id, order_id, message)
        VALUES ('ORDER_FINALIZED', p_user_id, p_order_id, CONCAT('Order #', p_order_id, ' has been
    finalized'));

        COMMIT;
    END IF;
END$$

DELIMITER ;
```

Listing 3: FinalizeOrder Stored Procedure

Usage Example:

```
CALL FinalizeOrder(123, 456);
```

4.3 GetCustomerOrderHistory

This stored procedure retrieves a customer's complete order history with detailed information.

```
DELIMITER $$

CREATE PROCEDURE GetCustomerOrderHistory(IN p_user_id INT)
BEGIN
    -- Get all orders for the user with item counts
    SELECT
        o.id AS order_id,
        o.created_at,
        o.total_price AS total,
        o.status,
        COUNT(oi.id) AS items_count,
        GROUP_CONCAT(p.name SEPARATOR ', ') AS product_names
    FROM
        orders o
    LEFT JOIN
        order_items oi ON o.id = oi.order_id
    LEFT JOIN
        products p ON oi.product_id = p.id
    WHERE
        o.user_id = p_user_id
    GROUP BY
        o.id
    ORDER BY
        o.created_at DESC;
END$$

DELIMITER ;
```

Listing 4: GetCustomerOrderHistory Stored Procedure

Usage Example:

```
CALL GetCustomerOrderHistory(456);
```

5 Database Triggers Implementation

Triggers are used to enforce business rules and maintain data integrity automatically when certain database events occur.

5.1 AfterOrderConfirmed

This trigger automatically updates product stock quantities when an order status changes to "confirmed".

```
DELIMITER $$

CREATE TRIGGER AfterOrderConfirmed
AFTER UPDATE ON orders
FOR EACH ROW
BEGIN
    DECLARE v_done INT DEFAULT 0;
    DECLARE v_product_id INT;
    DECLARE v_quantity INT;
    DECLARE cur CURSOR FOR
        SELECT product_id, quantity FROM order_items WHERE order_id = NEW.id;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_done = 1;

    IF OLD.status != 'confirmed' AND NEW.status = 'confirmed' THEN
        -- Log the order confirmation
        INSERT INTO logs (action, user_id, order_id, message)
        VALUES ('CHECKOUT', NEW.user_id, NEW.id, CONCAT('Order #', NEW.id, ' confirmed'));

        -- Update product stock for each item in the order
        OPEN cur;
        read_loop: LOOP
            FETCH cur INTO v_product_id, v_quantity;
            IF v_done THEN
                LEAVE read_loop;
            END IF;

            -- Decrease product stock
            UPDATE products
            SET stock = stock - v_quantity
            WHERE id = v_product_id;
        END LOOP;
        CLOSE cur;

        -- Log stock updates
        INSERT INTO logs (action, user_id, order_id, message)
        VALUES ('PRODUCT_UPDATE', NEW.user_id, NEW.id, CONCAT('Stock updated for order #', NEW.id));
    END IF;
END$$

DELIMITER ;
```

Listing 5: AfterOrderConfirmed Trigger

5.2 BeforeOrderItemInsert

This trigger prevents adding products to orders if the requested quantity exceeds available stock.

```
DELIMITER $$

CREATE TRIGGER BeforeOrderItemInsert
BEFORE INSERT ON order_items
FOR EACH ROW
BEGIN
    DECLARE available_stock INT;
    DECLARE v_user_id INT;

    -- Get available stock for the product
    SELECT stock INTO available_stock
    FROM products
    WHERE id = NEW.product_id;

    -- Get user ID for logging
    SELECT user_id INTO v_user_id
    FROM orders
    WHERE id = NEW.order_id;

    -- Check if we have enough stock
    IF NEW.quantity > available_stock THEN
        -- Log the stock violation attempt
        INSERT INTO logs (action, user_id, order_id, message)
        VALUES ('PRODUCT_UPDATE', v_user_id, NEW.order_id,
                CONCAT('Failed to add product #', NEW.product_id, ' to order #', NEW.order_id,
                       ': Requested ', NEW.quantity, ', Available ', available_stock));

        -- Raise an error to prevent the insert
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Cannot insert order item: requested quantity exceeds available stock';
    END IF;
END$$

DELIMITER ;
```

Listing 6: BeforeOrderItemInsert Trigger

5.3 AfterOrderCancelled

This trigger restores product stock quantities when an order is canceled.

```
DELIMITER $$

CREATE TRIGGER AfterOrderCancelled
AFTER UPDATE ON orders
FOR EACH ROW
BEGIN
    DECLARE v_done INT DEFAULT 0;
    DECLARE v_product_id INT;
    DECLARE v_quantity INT;
    DECLARE cur CURSOR FOR
        SELECT product_id, quantity FROM order_items WHERE order_id = NEW.id;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_done = 1;

    IF OLD.status != 'cancelled' AND NEW.status = 'cancelled' THEN
        -- Log the order cancellation
        INSERT INTO logs (action, user_id, order_id, message)
        VALUES ('ORDER_CANCEL', NEW.user_id, NEW.id, CONCAT('Order #', NEW.id, ' canceled'));

        -- Restore stock for each product in the order
        OPEN cur;
        read_loop: LOOP
```

```
            FETCH cur INTO v_product_id, v_quantity;
            IF v_done THEN
                LEAVE read_loop;
            END IF;

            -- Increase product stock
            UPDATE products
            SET stock = stock + v_quantity
            WHERE id = v_product_id;
        END LOOP;
        CLOSE cur;

        -- Log stock restoration
        INSERT INTO logs (action, user_id, order_id, message)
        VALUES ('PRODUCT_UPDATE', NEW.user_id, NEW.id, CONCAT('Stock restored for order #', NEW.id))
    ;
    END IF;
END$$

DELIMITER ;
```

Listing 7: AfterOrderCancelled Trigger

5.4 LogCanceledOrder

This trigger records detailed information about canceled orders in a dedicated history table.

```
DELIMITER $$

CREATE TRIGGER LogCanceledOrder
AFTER UPDATE ON orders
FOR EACH ROW
BEGIN
    IF OLD.status != 'cancelled' AND NEW.status = 'cancelled' THEN
        -- Insert into canceled_orders table
        INSERT INTO canceled_orders (order_id, reason, canceled_at)
        SELECT NEW.id, 'Order was canceled by user or admin', NOW()
        FROM dual
        WHERE NOT EXISTS (
            SELECT 1 FROM canceled_orders WHERE order_id = NEW.id
        );

        -- Log the cancellation record
        INSERT INTO logs (action, user_id, order_id, message)
        VALUES ('ORDER_CANCEL', NEW.user_id, NEW.id, CONCAT('Order #', NEW.id, ' cancellation
    recorded'));
    END IF;
END$$

DELIMITER ;
```

Listing 8: LogCanceledOrder Trigger

6 Key System Features

6.1 User Interface

The Court Kart user interface is designed with a focus on usability, accessibility, and responsive design to provide an optimal shopping experience across different devices.

Figure 6: Court Kart User Interface Sample

Key UI features include:

- Responsive design that automatically adapts to different screen sizes
- Intuitive navigation with clearly labeled menu options
- High-quality product images with zoom functionality
- Consistent color scheme and typography for brand identity
- Accessible design with proper contrast and semantic HTML
- Interactive elements with appropriate visual feedback

6.2 User Authentication and Authorization

The platform implements a comprehensive authentication and authorization system:

- Secure user registration with email verification
- Password hashing using PHP's password_hash() function
- Role-based access control (user and admin roles)
- Session management with CSRF protection
- "Remember Me" functionality using secure cookies
- Account management features (profile updates, password changes)

Figure 7: Authentication Flow Diagram

6.3 Shopping Cart Implementation

The shopping cart system provides a seamless experience for users to manage their selected products:

- Add products to cart with quantity selection
- Update product quantities directly from the cart
- Remove individual items or clear the entire cart
- Real-time subtotal and total calculations
- Persistent cart storage using database (for logged-in users)
- Stock validation to prevent ordering unavailable quantities

6.4 Checkout Process

The checkout process is streamlined to minimize friction while collecting necessary information:

- Multi-step checkout with progress indicators
- Shipping information collection with validation
- Payment method selection
- Order summary with final review

- Order confirmation with details and receipt
- Stock verification before order finalization

Figure 8: Checkout Process Flow Diagram

6.5 Admin Interface

The administrative interface provides powerful tools for site management:

- Dashboard with key performance metrics and recent orders
- Comprehensive product management (add, edit, delete)
- Order processing and status updates
- User management with role assignments
- Inventory tracking and stock management

Figure 9: Admin Interface Dashboard

7 Security Measures

7.1 Data Protection

The application implements several security measures to protect user data and prevent unauthorized access:

- Password hashing using PHP's native password_hash() with bcrypt
- Input validation and sanitization to prevent SQL injection
- HTML escaping to prevent Cross-Site Scripting (XSS) attacks
- CSRF tokens for all form submissions
- Prepared statements for all database queries
- Secure session handling with regeneration

7.2 Authentication Security

The authentication system includes:

- Rate limiting for login attempts to prevent brute force attacks
- Secure "Remember Me" implementation with token hashing
- Session timeout for inactive users
- Secure password reset mechanism

7.3 Authorization

Access control is implemented using:

- Route-level middleware for access restrictions
- Role-based permissions (user vs. admin)
- Resource ownership verification

8 Testing and Quality Assurance

8.1 Testing Methodology

The application was tested using multiple approaches:

- Manual testing of all user flows
- Database integrity testing for transactions
- Security testing for common vulnerabilities
- Cross-browser testing for compatibility
- Mobile responsiveness testing on various devices

8.2 Test Cases

Key test cases included:

- User registration and login process
- Shopping cart operations (add, update, remove)
- Order placement with stock verification
- Order cancellation and stock restoration
- Admin functions (product and order management)

9 Future Enhancements

While the current implementation of Court Kart provides a complete e-commerce solution, several potential enhancements could extend its functionality:

- **Payment Gateway Integration:** Integration with popular payment processors such as PayPal, Stripe, or Square
- **Advanced Product Recommendations:** Implementation of a recommendation engine based on user browsing and purchase history
- **Email Notifications:** Automated emails for order confirmations, shipping updates, and promotional content
- **Customer Reviews and Ratings:** Allow customers to rate products and leave reviews
- **Wishlist Functionality:** Enable users to save products for future consideration
- **Advanced Analytics Dashboard:** Enhanced reporting tools for business intelligence
- **Inventory Forecasting:** Predictive analytics for inventory management
- **Mobile Application:** Native mobile applications for iOS and Android

10 Conclusion

The Court Kart e-commerce platform successfully implements all required features for online shopping, with a particular focus on database integrity through stored procedures and triggers. The application follows best practices in web development, including separation of concerns through the MVC pattern, secure authentication and authorization, and responsive design.

The database design ensures data integrity while supporting complex business operations through the use of stored procedures and triggers. These database features automate critical processes such as inventory management, order processing, and canceled order handling.

The platform provides intuitive interfaces for both customers and administrators, making it easy to browse products, manage shopping carts, place orders, and handle administrative tasks. The system architecture ensures maintainability and scalability for future enhancements.

Note

Court Kart demonstrates how modern web technologies and advanced database features can be combined to create a robust e-commerce solution that meets both business requirements and user expectations.