Court Kart: E-Commerce Platform

PWEB and BDD Project

Submitted by:
HADJ ARAB Adel
Student ID: 222231482117

May 20, 2025

TABLE OF CONTENTS

1 Introduction

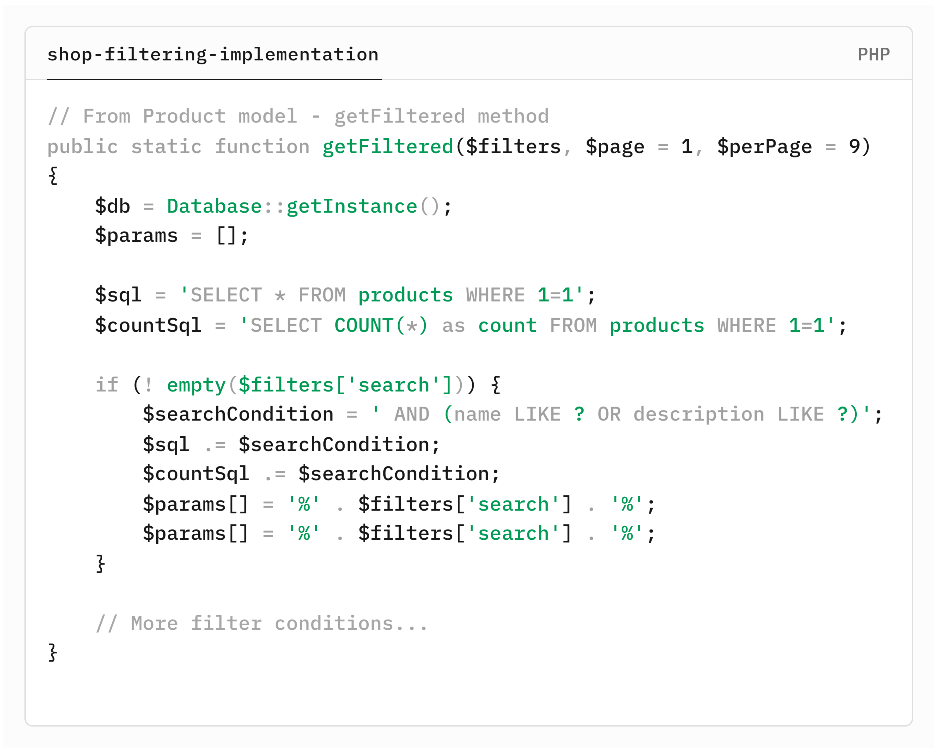
Court Kart is a specialized e-commerce platform designed for basketball enthusiasts, offering footwear, apparel, gear, and merchandise. The application follows the MVC (Model-View-Controller) architecture and implements features required for a complete online shopping experience. The platform provides a comprehensive solution for both customers and administrators, with features for inventory management, order processing, and user account management.

2 Web Application Features

2.1 Main Shop Page Features

The shop page implements all required features:

- **Product display** with images, descriptions, and prices
- **Advanced search and filtering**:
  - Text search (name, description)
  - Category filters
  - Price range filters
  - Sort options (price, popularity, newest)
- **Pagination** for browsing large product catalogs
- **Wishlist integration** for saving products



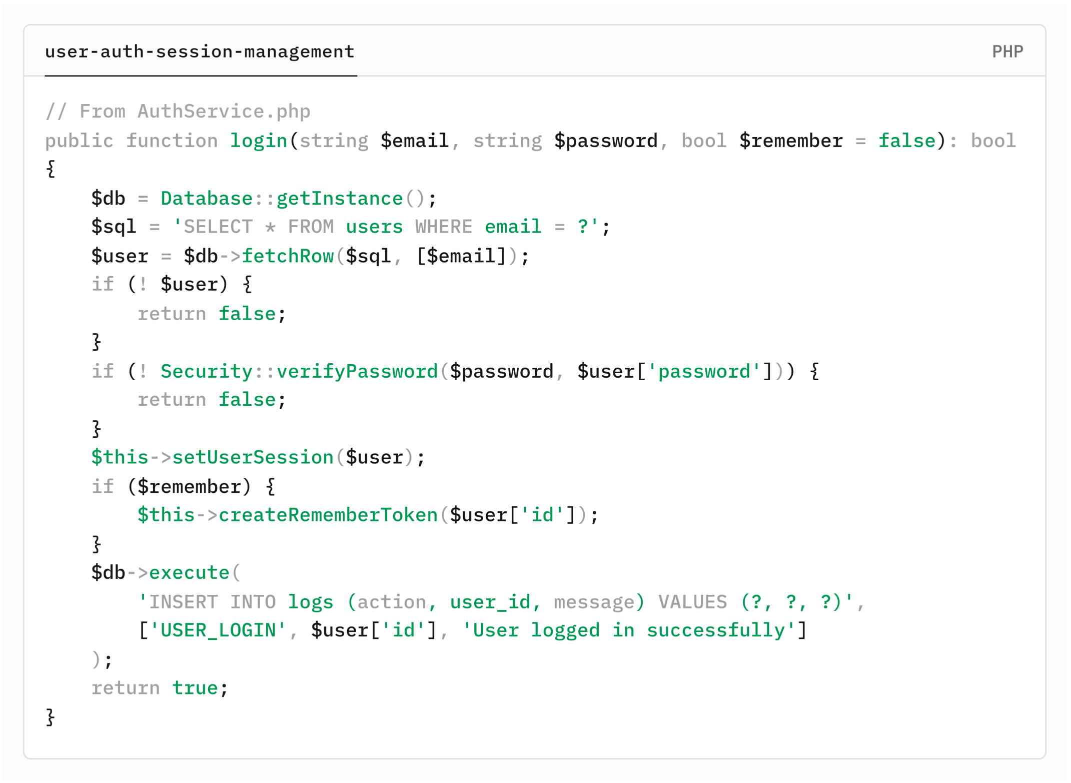
shop-filtering-implementation PHP

```php
// From Product model - getFiltered method
public static function getFiltered($filters, $page = 1, $perPage = 9)
{
    $db = Database::getInstance();
    $params = [];

    $sql = 'SELECT * FROM products WHERE 1=1';
    $countSql = 'SELECT COUNT(*) as count FROM products WHERE 1=1';

    if (! empty($filters['search'])) {
        $searchCondition = ' AND (name LIKE ? OR description LIKE ?)';
        $sql .= $searchCondition;
        $countSql .= $searchCondition;
        $params[] = '%' . $filters['search'] . '%';
        $params[] = '%' . $filters['search'] . '%';
    }

    // More filter conditions...
}
```

Figure 1: Shop Filtering Implementation

2.2 User Features

- **User Authentication**: Secure login/logout with session management
- **Product Detail Views**: Complete product information, specifications, and reviews
- **Shopping Cart System**:
  - Add/remove items
  - Update quantities
  - View cart state and totals
  - Session-based for guest users, database-synced for logged-in users
- **Order Tracking**: View status and history of placed orders



user-auth-session-management PHP

```php
// From AuthService.php
public function login(string $email, string $password, bool $remember = false): bool
{
    $db = Database::getInstance();
    $sql = 'SELECT * FROM users WHERE email = ?';
    $user = $db->fetchRow($sql, [$email]);
    if (! $user) {
        return false;
    }
    if (! Security::verifyPassword($password, $user['password'])) {
        return false;
    }
    $this->setUserSession($user);
    if ($remember) {
        $this->createRememberToken($user['id']);
    }
    $db->execute(
        'INSERT INTO logs (action, user_id, message) VALUES (?, ?, ?)',
        ['USER_LOGIN', $user['id'], 'User logged in successfully']
    );
    return true;
}
```

Figure 2: User Authentication with Session Management

```php
// From OrderController.php
public function show($id)
{
    if (! Session::get('user_id')) {
        Session::flash('error', 'Please login to view your order');
        header('Location: /login');
        exit;
    }

    $userId = Session::get('user_id');
    $orderId = (int) $id;
    $orderDetails = Order::getOrderDetails($orderId);

    if (empty($orderDetails)) {
        Session::flash('error', 'Order not found');
        // Render error view...
        return;
    }

    if ($orderDetails[0]['user_id'] != $userId) {
        Session::flash('error', 'You do not have permission to view this order');
        // Render access denied view...
        return;
    }

    // Process order details and render view...
}
```

Listing 1: Order tracking implementation

2.3 Administrative Features

An admin interface allows store management:

- **Product Management**: Add, edit, delete products, update inventory
- **Order Processing**: View and update order status
- **Inventory Control**: Stock level monitoring with automatic alerts

```
// From AdminController.php
public function updateOrderStatus()
{
    if ($_SERVER['REQUEST_METHOD'] !== 'POST') {
        header('Location: /admin/orders');
        exit;
    }

    $orderId = $_POST['order_id'] ?? 0;
    $status = $_POST['status'] ?? '';

    if (! $orderId || ! $status) {
        Session::set('error', 'Invalid order ID or status');
        header('Location: /admin/orders');
        exit;
    }

    if (Order::updateStatus($orderId, $status)) {
        Session::set('success', 'Order status updated successfully');
    } else {
        Session::set('error', 'Failed to update order status');
    }

    header("Location: /admin/orders/{$orderId}");
    exit;
}
```

Listing 2: Admin order status update

```
// From Middleware.php
public static function admin(): bool
{
    $authService = new AuthService;

    if (! $authService->isLoggedIn()) {
        Session::set('redirect_after_login', $_SERVER['REQUEST_URI']);
        header('Location: /login');
        exit;
    }

    if (! $authService->isAdmin()) {
        header('Location: /unauthorized');
        exit;
    }

    return true;
}
```

Listing 3: Admin middleware protection

3 Database Design

3.1 Relational Schema and Relationships

Court Kart's database consists of the following key tables and relationships:

- **users**: Stores authentication details and profile data
  - One-to-many relationship with `orders` and `cart_items`
- **products**: Contains product details including inventory levels and pricing
  - Many-to-many with `orders` (via `order_items`)
  - Many-to-many with users' wishlists (via `wishlists`)
- **cart_items**: Links users to products in their cart
  - Many-to-one relationship with `users` and `products`
- **orders**: Records transactions with status tracking
  - Many-to-one relationship with `users`

  - One-to-many relationship with `order_items`
  - One-to-one relationship with `canceled_orders`

- **order_items**: Contains line items within each order
  - Many-to-one relationship with `orders` and `products`
- **canceled_orders**: Records history and reasons for cancellations
  - One-to-one relationship with `orders`
- **product_reviews**: Stores customer ratings and reviews
  - Many-to-one relationship with `products` and `users`
- **logs**: Maintains a comprehensive audit trail of operations

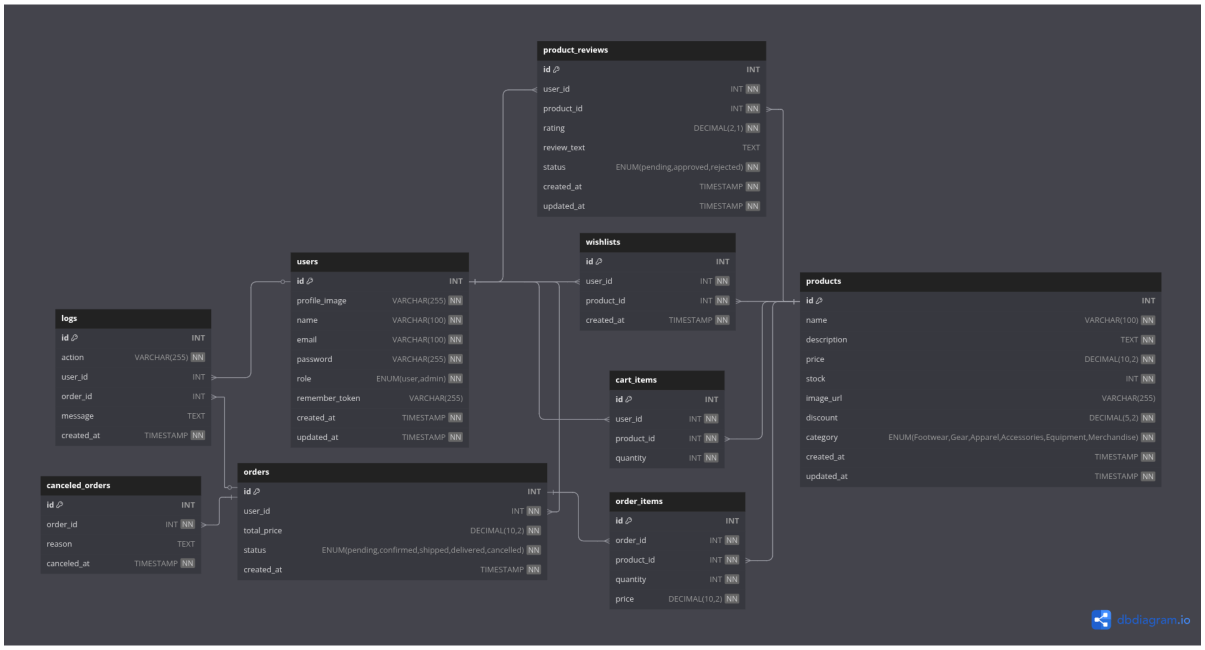


Figure 3: Court Kart Database Schema

4 Stored Procedures Implementation

4.1 GetOrderDetails Procedure

This procedure fulfills the requirement to display order details and total amount:

```
CREATE PROCEDURE GetOrderDetails (IN p_order_id INT)
BEGIN
    SELECT
        o.id AS order_id,
        o.created_at AS order_date,
        o.status,
        u.name AS customer_name,
        u.email AS customer_email,
        p.id AS product_id,
        p.name AS product_name,
        p.image_url,
        oi.quantity,
        oi.price AS unit_price,
        (oi.quantity * oi.price) AS subtotal,
        o.total_price AS total_amount
    FROM
        orders o
        JOIN users u ON o.user_id = u.id
        JOIN order_items oi ON o.id = oi.order_id
        JOIN products p ON oi.product_id = p.id
    WHERE
        o.id = p_order_id;
END
```

4.2 FinalizeOrder Procedure

This procedure finalizes an order and empties the cart once confirmed:

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

    START TRANSACTION;

    SELECT COUNT(*) INTO v_order_exists
    FROM orders
    WHERE id = p_order_id AND user_id = p_user_id AND status = 'pending';

    IF v_order_exists = 1 THEN
        UPDATE orders
        SET status = 'confirmed'
        WHERE id = p_order_id;

        DELETE FROM cart_items
        WHERE user_id = p_user_id;

        INSERT INTO logs (action, user_id, order_id, message)
        VALUES ('CHECKOUT', p_user_id, p_order_id, 'Order finalized and cart emptied');

        COMMIT;
    ELSE
        ROLLBACK;
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Invalid or non-pending order for this user';
    END IF;
```

```
END
```

4.3 GetCustomerOrderHistory Procedure

This procedure displays a customer's order history:

```
CREATE PROCEDURE GetCustomerOrderHistory (
    IN p_user_id INT
)
BEGIN
    SELECT
        o.id AS order_id,
        o.created_at AS order_date,
        o.total_price,
        o.status,
        COUNT(oi.id) AS item_count,
        GROUP_CONCAT(p.name SEPARATOR ', ') AS products
    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, o.created_at, o.total_price, o.status
    ORDER BY
        o.created_at DESC;
END
```

5 Triggers Implementation

5.1 AfterOrderConfirmed Trigger

This trigger automatically updates product stock quantities when an order is confirmed:

```
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 using cursor
        OPEN cur;
        read_loop: LOOP
            FETCH cur INTO v_product_id, v_quantity;
            IF v_done THEN
                LEAVE read_loop;
            END IF;
            UPDATE products
            SET stock = stock - v_quantity
            WHERE id = v_product_id;
        END LOOP;
        CLOSE cur;
    END IF;
END
```

5.2 BeforeOrderItemInsert Trigger

This trigger prevents adding items to orders if the requested quantity exceeds available stock:

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

    SELECT stock INTO available_stock
    FROM products
    WHERE id = NEW.product_id;

    SELECT user_id INTO v_user_id
    FROM orders
    WHERE id = NEW.order_id;

    IF NEW.quantity > available_stock THEN
        -- Log the stock limitation event
        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));

        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Cannot insert order item: requested quantity exceeds
    available stock';
    END IF;
```

```
END
```

5.3 AfterOrderCancelled Trigger

This trigger restores product stock when an order is canceled:

```
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 product stock using cursor
        OPEN cur;
        read_loop: LOOP
            FETCH cur INTO v_product_id, v_quantity;
            IF v_done THEN
                LEAVE read_loop;
            END IF;
            UPDATE products
            SET stock = stock + v_quantity
            WHERE id = v_product_id;
        END LOOP;
        CLOSE cur;
    END IF;
END
```

5.4 LogCanceledOrder Trigger

This trigger logs canceled orders into a history table:

```
CREATE TRIGGER LogCanceledOrder
AFTER UPDATE ON orders
FOR EACH ROW
BEGIN
    IF OLD.status != 'cancelled' AND NEW.status = 'cancelled' THEN
        -- Insert into cancellation history 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 creation
        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
```

6 Conclusion

The Court Kart e-commerce platform successfully implements all required features specified in the project instructions:

- A complete shop page with product listings and filters
- Detailed product views with descriptions and prices
- User authentication with session management
- Shopping cart functionality for adding/removing items
- Admin interface for managing products
- Database integration for all aspects of the application
- Stored procedures for order management and history
- Triggers for inventory control and order handling

The platform balances user experience with robust back-end functionality, creating a complete e-commerce solution for basketball enthusiasts while meeting all technical requirements.