CS306 Project Phase III:
Web Integrations of SQL and NoSQL Databases

Group 26: Ekin Renas Katırcı (31302) Fikret Dara Aktaş (32511)
Başar Zaim (30759)

May 28, 2025

Contents

1 Introduction

This document details the design, structure, and functionality of our pharmacy management web application. We expose MySQL triggers and stored procedures through PHP pages and implement a MongoDB-backed support ticket system.

2 System Design and Features

2.1 Overall Structure

The project is organized under `htdocs/cs306_hw3/`:

- `user/`: user portal
  - `triggers/`: one page per trigger
  - `procedures/`: one page per stored procedure
  - `tickets/`: support ticket subsystem
- `admin/`: admin ticket dashboard
- `config.php`: shared DB connection settings
- `vendor/`, `composer.json`: MongoDB PHP driver

2.2 User Interface

1. **Homepage** (`user/index.php`): Links to trigger demos, procedure demos, and support tickets.
2. **Trigger Pages**:
   - Each page describes a trigger's purpose.
   - Two "Case" buttons simulate fixed inputs to demonstrate valid vs. invalid behavior.
   - A "Custom" form lets users supply any `order_id`, `medicine_id`, and `quantity`.
   - Result messages appear inline (green for success, red for failure).
3. **Stored Procedure Pages**:
   - Each page shows a description, input fields for parameters, two fixed "Case" buttons, and a custom form.
   - On submit, the page calls the stored procedure and displays the result set or summary.
4. **Support Ticket Subsystem** (`user/tickets/`):
   - `index.php`: Dropdown of active usernames (via MongoDB `distinct`), shows their tickets.
   - `create.php`: Form to open a new ticket (username, description).
   - `detail.php`: View ticket metadata, comments list, add new comment or mark resolved.

2.3 Admin Interface

- `admin/index.php`: Lists all active tickets across users.
- `admin/ticket_details.php`: View any ticket's details, add comments (as "admin"), or mark resolved.
- MongoDB connection is shared via `admin/mongo.php`.

3 Triggers

3.1 1. prevent_zero_quantity_orders

Purpose: Disallow nonsensical orders of zero quantity. **Behavior:** Fires `BEFORE INSERT` on `Order_Medicine`, and if `NEW.quantity = 0`, aborts with an error.

SQL script:

```
USE pharmacydb;
DELIMITER $$
CREATE TRIGGER prevent_zero_quantity_orders
BEFORE INSERT ON Order_Medicine
FOR EACH ROW
BEGIN
  IF NEW.quantity = 0 THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Quantity must be greater than 0.';
  END IF;
END $$
DELIMITER ;
```

Demo page: `user/triggers/prevent_zero_quantity_orders.php`

- **Case 1 (Valid):**
  `order_id=22, medicine_id=14, quantity=5`
  → The trigger condition is false, insert succeeds (green).
- **Case 2 (Invalid):**
  `order_id=23, medicine_id=15, quantity=0`
  → Trigger aborts with SQLSTATE[45000] "Quantity must be greater than 0." (red).
- **Custom:** Any values; users can verify mixed behavior.

3.2 2. prevent_expired_medicine_order

Purpose: Prevent ordering expired stock. **Behavior:** Fires `BEFORE INSERT` on `Order_Medicine`; retrieves `expiration_date` from `Medicine`. If it's before `CURDATE()`, abort.

SQL script:

```
USE pharmacydb;
DELIMITER $$
CREATE TRIGGER prevent_expired_medicine_order
BEFORE INSERT ON Order_Medicine
FOR EACH ROW
BEGIN
  DECLARE expiry DATE;
  SELECT expiration_date INTO expiry
    FROM Medicine
   WHERE medicine_id = NEW.medicine_id;
  IF expiry < CURDATE() THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Cannot order expired medicine.';
  END IF;
END $$
DELIMITER ;
```

Demo page: `user/triggers/prevent_expired_medicine_order.php`

- **Case 1 (Valid):**
  `order_id=22, medicine_id=14, quantity=1`
  (medicine_id 14 expiry 2026-12-31). Trigger condition false → insert succeeds.
- **Case 2 (Invalid):**
  `order_id=23, medicine_id=7, quantity=1`
  (medicine_id 7 expiry 2024-11-25). Trigger aborts "Cannot order expired medicine."
- **Custom:** Try any ID to see acceptance vs. rejection.

3.3 3. update_stock_after_order

Purpose: Keep inventory in sync by deducting sold quantity. **Behavior:** Fires `AFTER INSERT` on `Order_Medicine` and subtracts `NEW.quantity` from `Pharmacy_Medicine.stock_quantity`.

SQL script:

```
USE pharmacydb;
DELIMITER $$
CREATE TRIGGER update_stock_after_order
AFTER INSERT ON Order_Medicine
FOR EACH ROW
BEGIN
  UPDATE Pharmacy_Medicine
    SET stock_quantity = stock_quantity - NEW.quantity
   WHERE medicine_id = NEW.medicine_id;
END $$
DELIMITER ;
```

Demo page: `user/triggers/update_stock_after_order.php`

- **Case 1 (Small order):**
  `order_id=22, medicine_id=14, quantity=2`
  Initial stock 100 → new stock 98. (green shows updated stock).
- **Case 2 (Large order):**
  `order_id=23, medicine_id=15, quantity=50`
  Initial stock 200 → new stock 150.
- **Custom:** Any combination to observe stock decrements.

4 Stored Procedures

4.1 1. GetLowStockMedicines

Purpose: Identify items needing restock. **Behavior:** Returns all medicines whose `stock_quantity` is below a user-supplied threshold.

SQL script:

```
USE pharmacydb;
DELIMITER $$
CREATE PROCEDURE GetLowStockMedicines(IN p_threshold INT)
BEGIN
  SELECT medicine_id, name, stock_quantity
    FROM Medicine
   WHERE stock_quantity < p_threshold;
END $$
DELIMITER ;
```

Demo page: `user/procedures/get_low_stock_medicines.php`

- Input field: `threshold`.
- **Case 1:** `threshold=50` → returns medicines with stock<50.
- **Case 2:** `threshold=10` → returns medicines with stock<10.
- **Custom:** Any threshold to filter list dynamically.

4.2 2. GenerateCustomerOrderReport

Purpose: Summarize a customer's order history. **Behavior:** For a given $\texttt{customer}_id, returnstotalordercount, t$

SQL script:

```
USE pharmacydb;
DELIMITER $$
CREATE PROCEDURE GenerateCustomerOrderReport(IN p_customer_id INT)
BEGIN
  SELECT
    COUNT(*)          AS total_orders,
    SUM(total_amount) AS total_amount,
    MAX(order_date)   AS last_order_date
  FROM Orders
 WHERE customer_id = p_customer_id;
END $$
DELIMITER ;
```

Demo page: user/procedures/generate_customer_order_report.php

- Input box: customer_id.
- Case 1: ID 1 → 3 orders, $150 total, last 2025-05-01.
- Case 2: ID 5 → 2 orders, $90 total, last 2025-02-05.
- Custom: Any ID to generate live report.

4.3 3. CreateOrderWithStockUpdate

Purpose: Combine order creation and stock adjustment in one call. Behavior: Inserts a new row into Orders, captures its order_id, then inserts into Order_Medicine (trigger handles stock reduction).

SQL script:

```
USE pharmacydb;
DELIMITER $$
CREATE PROCEDURE CreateOrderWithStockUpdate(
  IN p_customer_id INT,
  IN p_medicine_id INT,
  IN p_quantity INT
)
BEGIN
  -- create order record
  INSERT INTO Orders(order_date,total_amount,customer_id)
    VALUES(
      CURDATE(),
      p_quantity * (SELECT price FROM Medicine WHERE medicine_id=
         p_medicine_id),
```

```
      p_customer_id
    );
  -- retrieve new order_id
  SET @new_oid = LAST_INSERT_ID();
  -- add to join table (trigger will update stock)
  INSERT INTO Order_Medicine(order_id,medicine_id,quantity)
    VALUES(@new_oid,p_medicine_id,p_quantity);
END $$
DELIMITER ;
```

Demo page: user/procedures/create_order_with_stock_update.php

- Inputs: customer_id, medicine_id, quantity.
- Case 1: (1,14,2) → new order 22, stock 100→98.
- Case 2: (2,17,5) → stock 250→245.
- Custom: Full control of parameters.

5 MongoDB Queries (Support System)

All ticket operations use the PHP MongoDB driver via admin/mongo.php:

```
// bootstrap in admin/mongo.php:
$manager    = new MongoDB\Driver\Manager("mongodb://localhost:27017");
$ticketsCol = (new MongoDB\Client)->cs306_hw3->tickets;
```

1. List active usernames

```
// user/tickets/index.php
$usernames = $ticketsCol->distinct(
  'username',
  ['status' => true]
);
```

2. Fetch tickets for a user

```
// after user selects and posts 'username':
$cursor  = $ticketsCol->find([
  'username' => $_POST['username'],
  'status'   => true
]);
$tickets = iterator_to_array($cursor);
```

3. Create a new ticket

```
// user/tickets/create.php
$ticketsCol->insertOne([
  'username'    => $_POST['username'],
  'description' => $_POST['description'],
  'created_at'  => date('c'),
  'status'      => true,
  'comments'    => []
]);
```

4. Fetch ticket details

```
// user/tickets/detail.php
$ticket = $ticketsCol->findOne([
  '_id' => new MongoDB\BSON\ObjectId($id)
]);
```

5. Add comment / mark resolved

```
// Add comment:
$ticketsCol->updateOne(
  ['_id'=> $oid],
  ['$push'=> ['comments'=>[
    'username'=> $_POST['name'],
    'text'    => $_POST['comment'],
    'at'      => date('c')
  ]]]
);
// Mark resolved:
$ticketsCol->updateOne(
  ['_id'=> $oid],
  ['$set'=> ['status'=> false]]
);
```

6 References

- MongoDB PHP Library: https://www.mongodb.com/docs/php-library/current/
- PHP + XAMPP Tutorial: https://www.youtube.com/watch?v=jLqBiSDNXO0
- MySQL + PHP Tutorial: https://www.youtube.com/watch?v=-1DTYAQ25bY