Assignment 2: Group 5

Section: 11
Course: CSE470

Name: Abrar Samin
ID: 22301739

Name: Mahir Tajwar Rahman
ID: 22299422

Name: Asiful Islam Mahir
ID: 22299318

Name: Mohd Tashwaruddin Safin
ID: 21201160

Department of Computer Science and Engineering
Brac University

We've implemented the Observer Pattern in the order routes to handle order status changes. This pattern improves the code by:

```
// Observer Pattern implementation
class OrderStatusSubject {
  constructor() {
    this.observers = [];
  }

  subscribe(observer) {
    this.observers.push(observer);
  }

  unsubscribe(observer) {
    this.observers = this.observers.filter(obs => obs !== observer);
  }

  notify(order, previousStatus) {
    this.observers.forEach(observer => observer.update(order,
previousStatus));
  }
}

// Observer implementations
class InventoryObserver {
  update(order, previousStatus) {
    // Handle inventory updates based on status change
    if (order.status === 'cancelled' && previousStatus !== 'cancelled') {
      // Restore inventory when order is cancelled
      this.restoreInventory(order);
    }
  }

  async restoreInventory(order) {
    try {
      for (const item of order.items) {
        await Product.findByIdAndUpdate(
          item.product._id,
          { $inc: { stock: item.quantity } },
          { new: true }
```

```
        );
        console.log(`Restored ${item.quantity} units to product
${item.product.name} (${item.product._id})`);
      }
    } catch (error) {
      console.error('Error restoring inventory:', error);
    }
  }
}

class NotificationObserver {
  update(order, previousStatus) {
    console.log(`Order ${order._id} status changed from ${previousStatus}
to ${order.status}`);

    // Create in-app notification based on status
    this.createUserNotification(order);
  }

  async createUserNotification(order) {
    try {
        const notificationMessage = this.getNotificationMessage(order);

      console.log(`Creating in-app notification for user ${order.user}:
${notificationMessage}`);

      const notification = new Notification({
        user: order.user,
        message: notificationMessage,
        type: 'order_update',
        orderId: order._id,
        read: false
      });
      await notification.save();

    } catch (error) {
      console.error('Error creating notification:', error);
    }
  }
```

```
  getNotificationMessage(order) {
    switch(order.status) {
      case 'confirmed':
        return `Your order #${order._id} has been confirmed and is being
processed.`;
      case 'shipped':
        return `Your order #${order._id} has shipped and is on the way!`;
      case 'delivered':
        return `Your order #${order._id} has been delivered. Enjoy!`;
      case 'cancelled':
        return `Your order #${order._id} has been cancelled.`;
      default:
        return `Your order #${order._id} status has been updated to:
${order.status}`;
    }
  }
}


// Create the subject and observers
const orderStatusSubject = new OrderStatusSubject();
const inventoryObserver = new InventoryObserver();
const notificationObserver = new NotificationObserver();

// Subscribe observers
orderStatusSubject.subscribe(inventoryObserver);
orderStatusSubject.subscribe(notificationObserver);
```

```
    // Store previous status for observers
    const previousStatus = order.status;
```

```

    // Notify observers about the status change
    orderStatusSubject.notify(order, previousStatus)
```

The Observer pattern in this implementation works through a well-defined relationship between subjects and observers:

Subject-Observer Relationship

- `**OrderStatusSubject**` maintains a list of observers that are notified when an order's status changes
- Each observer implements an `**update()**` method that defines how it responds to the status change

Flow of Operation

- When an order status changes (via admin update or user cancellation), the route handler:
  - Stores the previous status
  - Updates the order status
  - Calls `**orderStatusSubject.notify(order, previousStatus)**`
- The subject loops through all registered observers and calls each one's `**update()**` method
- Each observer independently decides what actions to take based on the status change

Decoupling Benefits:

- The route handlers don't need to know what side effects occur when status changes
- New observers can be added without modifying the route handlers
- Each observer handles a specific responsibility (inventory, notifications, etc.)

Real-world Example:

- When an order is cancelled:
  - The route handler changes the status and notifies the subject
  - `InventoryObserver` detects the cancellation and restores product inventory
  - `NotificationObserver` logs the status change and could send notifications