NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SC2006 - Software Engineering Lab 4 Deliverables

Lab Group	**SCEB**
Team	**Team 3 (HawkerGo)**
Members	**Dosi Veer (U2323005K)**
	Razan Siraj (U2321259G)
	Chin Jiaqi (U2421026J)
	Jain Ishita Chetan (U2321668B)
	Le Nguyen Bao Huy (U2322337G)
	Liong Xun Qi (U2322609H)

Contents

Black Box Testing

1.1 Selected Control Class

For our testing, we chose AuthController as the control class. This controller handles the authentication system within the application, which includes user registration and login functionalities.

During the sign-up process, users are required to provide their email address, password, and full name. Once the registration is successful, their information—along with the securely hashed password—is saved in the database. Users can subsequently log in using the same credentials they registered with.

In the login process, users enter their previously registered email and password. If the provided credentials match an existing user record (both email and hashed password), the user is successfully authenticated and redirected to the home page.

1.2 Equivalence Class Testing

Since the login and sign-up functionalities rely on distinct input values, boundary value testing is not applicable in this context. Instead, equivalence class testing is used.

- **Login Function:**

1. Valid Class**:** Inputs are correctly formatted and correspond to a registered user.

2. Invalid Class: Inputs are either incorrectly formatted, incomplete, or do not match any registered account.

- **Sign-Up Function:**

1. Valid Class: All required fields—email, password, and full name—are correctly filled, and the email is not already registered.
2. Invalid Class: The Inputs are missing, the email format is incorrect, or the email already exists in the system.

1.3. Test Cases and Testing Results

1.3.1 Sign up

Input parameters:

1. Email
2. Password
3. Full Name
4. User

Test Case Name	Test Input	Expected Output	Actual Output	Test Result
Signup-01	(Valid) Email: “testuser@gmail.com” (Valid) Password: “testpassword” (Valid) Full Name: “Test Name” User: Customer	Account created! Please login.	Account created! Please login.	Pass
Signup-02	(Invalid) Email: “testuser@gmail.com” (Valid) Password: “testpassword” (Valid) Full Name: “Test Name” User: Customer	Error: “User already exists”	Error: “User already exists.”	Pass
Signup-03	(Invalid) Email: “” (Valid) Password: “testpassword” (Valid) Full Name: “Test Name” User:Customer	Error: “Please fill in all fields”	Error: “Please fill in all fields”	Pass
Signup-04	(Invalid) Email: “testuseratgmaildotcom” (Valid) Password: “testpassword” (Valid) Full Name: “Test Name” User:Customer	Error: Registration failed	Error: Registration failed	Pass
Signup-05	(Valid) Email: “testuser@gmail.com” (Invalid) Password: “” (Valid) Full Name: “Test Name” User:Customer	Error: “Please fill in all fields”	Error: “Please fill in all fields”	Pass
Signup-06	(Valid) Email: “testuser@gmail.com” (Invalid) Password: “1234” (Valid) Full Name: “Test Name” User: Customer	Error: Registration failed	Error: Registration failed	Pass

Signup-07	(Valid) Email: “testuser@gmail.com” (Valid) Password: “testpassword” (Invalid) Full Name: “” User:Customer	Error: “Please fill in all fields”	Error: “Please fill in all fields”	Pass
Signup-08	(Invalid) Email: “testuser@gmail.com” (Valid) Password: “testpassword” (Valid) Full Name: “Test Name” User: Stall Owner	Error: “User already exists”	Error: “User already exists”	Pass

1.3.2 Login

Input parameters:
Email
Password

Test Case Name	Test Input	Expected output	Actual output	Test Result
Login-01	(Valid) Email: “testuser@gmail.com” (Valid) Password: “testpassword”	Login Success	Login Success	Pass
Login-02	Valid Email: “testuser@gmail.com” (Invalid) Password: “wrongpassword”	Error: “Invalid credentials”	Error: “Invalid credentials”	Pass
Login-03	(Valid) Email: “testuser@gmail.com” (Invalid) Password: “”	Error: “Please fill in all fields”	Error: “Please fill in all fields”	Pass
Login-04	(Invalid) Email: “fakeuser@gmail.com” (Valid) Password: “testpassword”	Error: “Invalid credentials”	Error: “Invalid credentials”	Pass
Login-05	(Invalid) Email: “” (Valid) Password: “testpassword”	Error: “Please fill in all fields”	Error: “Please fill in all fields”	Pass
Login-06	(Invalid) Email: “testuseratgmaildotcom” (Valid) Password: “testpassword”	Error: Login Failed	Error: Login Failed	Pass

White Box Testing

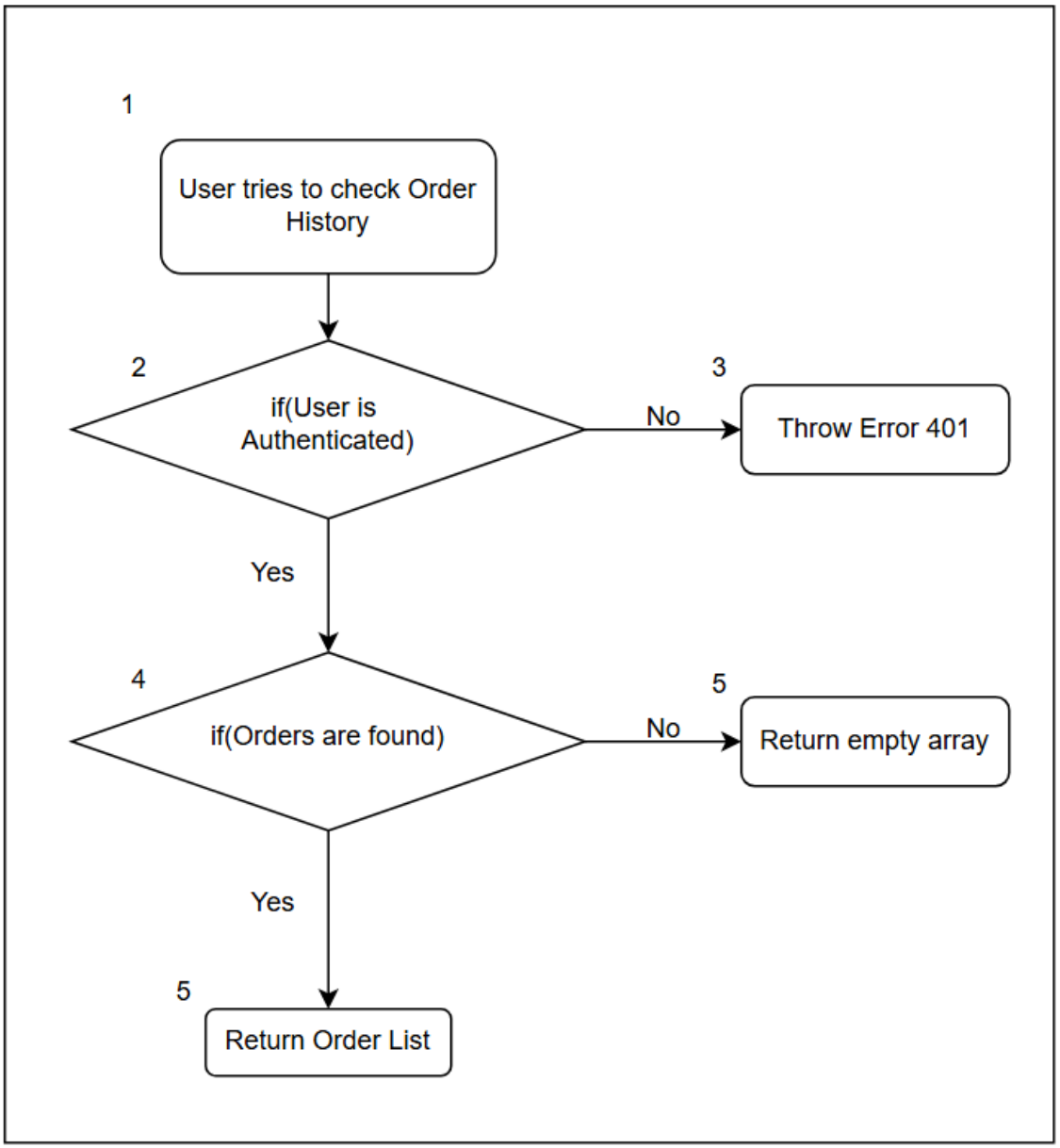
A. getUserOrders

1. Test Code

```
describe('GET /api/orders/user (getUserOrders)', () => {
    it('should return orders placed by the authenticated user', async () => {
        const res = await request(app)
            .get('/api/orders/user')
            .set('x-auth-token', tokenCustomer);

        expect(res.statusCode).toBe(200);
        expect(Array.isArray(res.body)).toBe(true);
        expect(res.body.length).toBeGreaterThanOrEqual(1); // Should include testOrder at least
        // Check if testOrder is present and user matches
        const foundOrder = res.body.find(o => o._id.toString() === testOrder._id.toString());
        expect(foundOrder).toBeDefined();
        expect(foundOrder.user._id.toString()).toBe(customerUser._id.toString());
        // Check population
        expect(foundOrder).toHaveProperty('stall');
        expect(foundOrder.stall).toHaveProperty('name', testStall.name);
        expect(foundOrder.user).toHaveProperty('name', customerUser.name);
    });

    it('should return an empty array if the authenticated user has no orders', async () => {
        const res = await request(app)
            .get('/api/orders/user')
            .set('x-auth-token', tokenOtherCustomer); // Use user who hasn't ordered yet (in this suite)

        expect(res.statusCode).toBe(200);
        expect(Array.isArray(res.body)).toBe(true);
        expect(res.body.length).toBe(1);
    });

     it('should return 401 if not authenticated', async () => {
        const res = await request(app).get('/api/orders/user');
        expect(res.statusCode).toBe(401);
    });
});
```

let testStall: any

2. Control Flow Diagram



3. Cyclomatic Complexity

Cyclomatic Complexity (CC) is calculated as the number of binary decision points + 1.
From the control flow diagram, there are two decision points:
1. Check if the user is authenticated
2. Check if the user has placed any orders

Thus,
Cyclomatic Complexity (CC) = 2 + 1 = 3

4. Basis Paths

The following are the independent basis paths derived from the control flow logic:

- Path #1 (Baseline): Authenticated user with existing orders retrieves them successfully.
- Path #2: Authenticated user with no existing orders receives an empty array.
- Path #3: Request made without authentication — receives 401 Unauthorized.

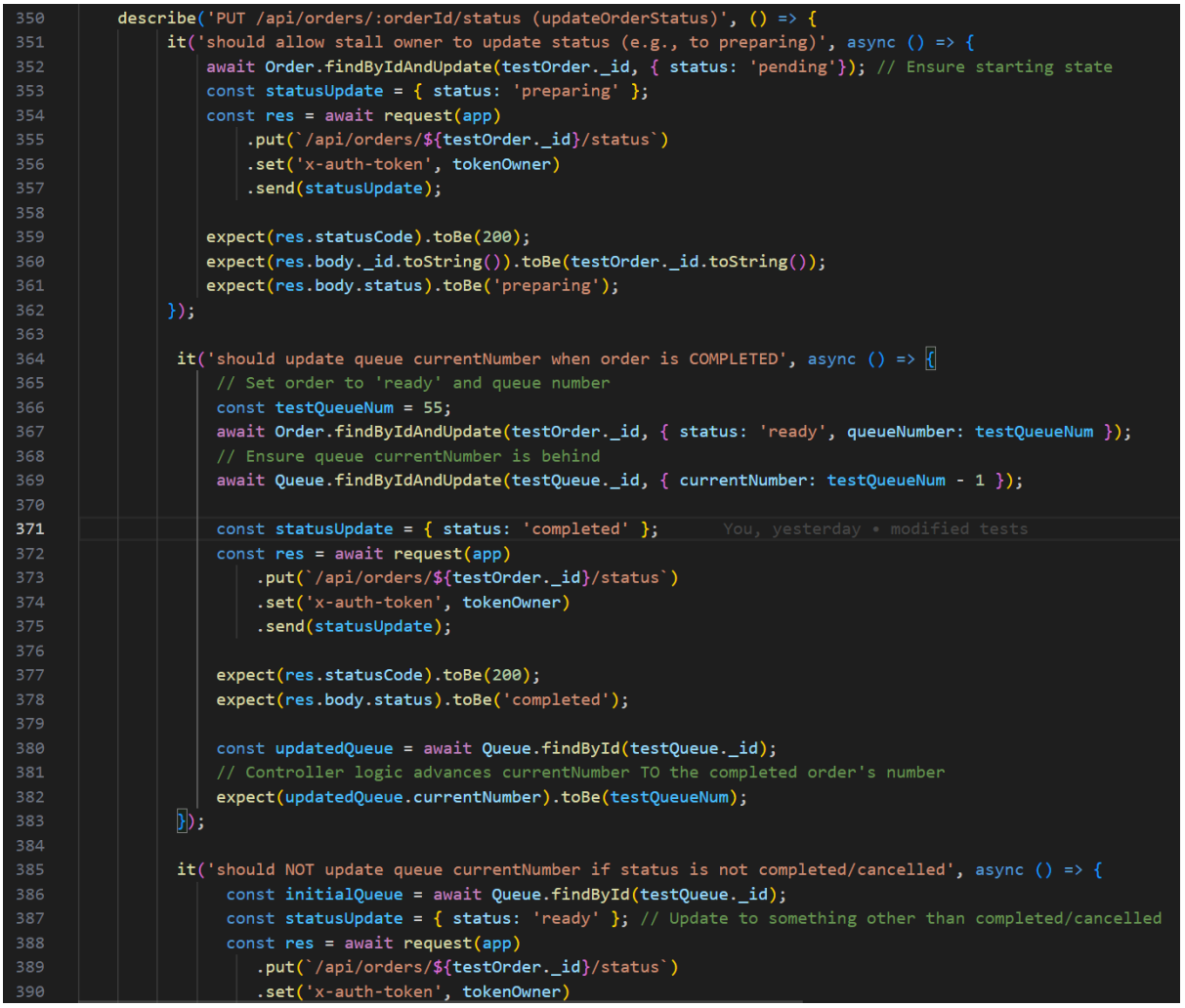
5. Test Cases and Results

Test Case Name	Test Input	Expected Output	Actual Output	Test Result
UserOrders-01	GET /api/orders/user with valid token for user with orders	200 OK, list of user's orders with correct population	200 OK, list of user's orders with correct population	Pass
UserOrders-02	GET /api/orders/user with valid token for user with no orders	200 OK, empty array returned	200 OK, empty array returned	Pass
UserOrders-03	GET /api/orders/user with no token (unauthenticated)	401 Unauthorized	401 Unauthorized	Pass

B. updateOrderStatus

1. Test Code

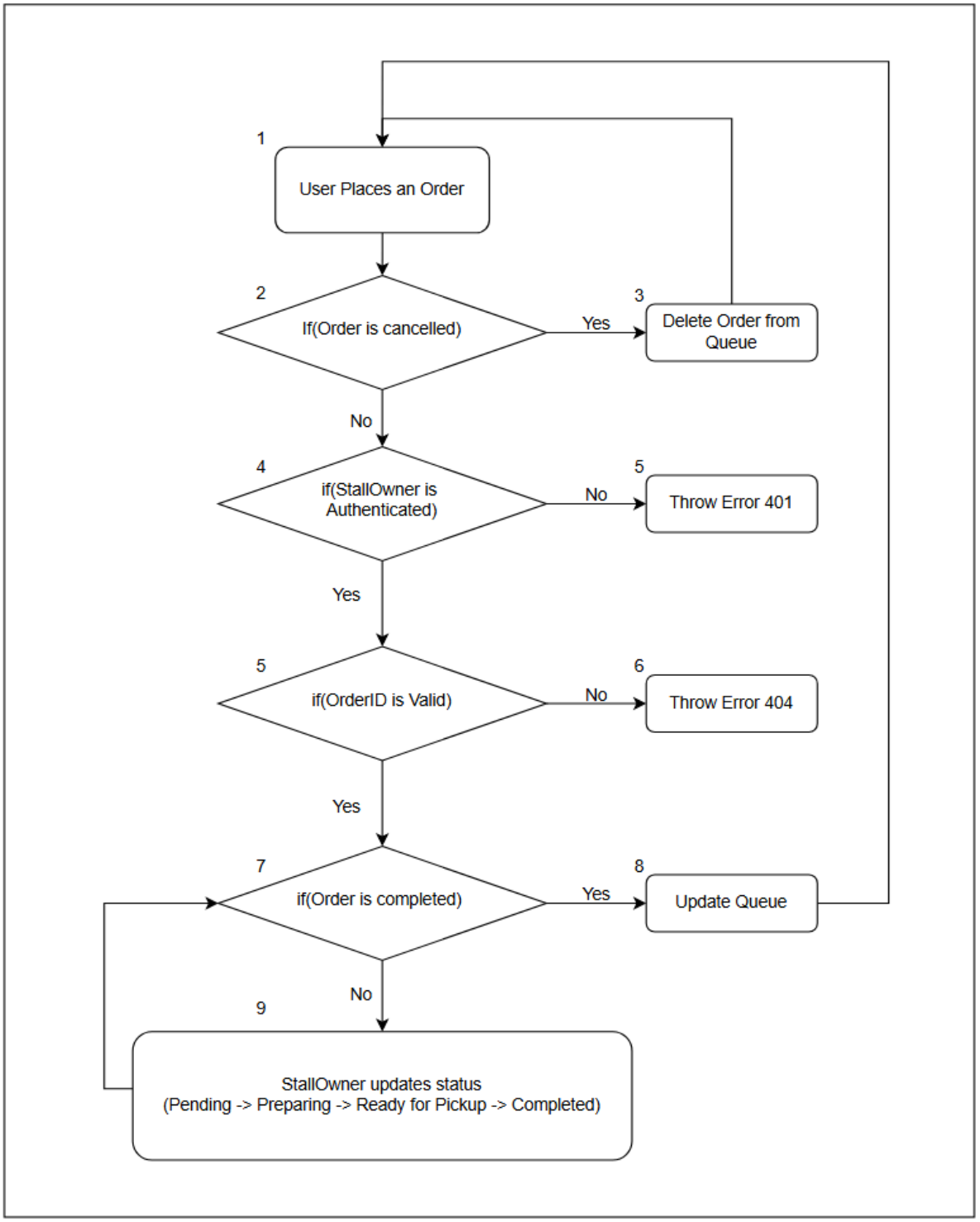
The `updateOrderStatus` method updates the status of an order that is already placed by a customer, potentially affecting the current number in the queue associated with the order. This includes multiple scenarios such as order status updates to various states and validation of user authentication and authorization.



```
describe('PUT /api/orders/:orderId/status (updateOrderStatus)', () => {
    it('should allow stall owner to update status (e.g., to preparing)', async () => {
        await Order.findByIdAndUpdate(testOrder._id, { status: 'pending'}); // Ensure starting state
        const statusUpdate = { status: 'preparing' };
        const res = await request(app)
            .put(`/api/orders/${testOrder._id}/status`)
            .set('x-auth-token', tokenOwner)
            .send(statusUpdate);

        expect(res.statusCode).toBe(200);
        expect(res.body._id.toString()).toBe(testOrder._id.toString());
        expect(res.body.status).toBe('preparing');
    });

     it('should update queue currentNumber when order is COMPLETED', async () => {
        // Set order to 'ready' and queue number
        const testQueueNum = 55;
        await Order.findByIdAndUpdate(testOrder._id, { status: 'ready', queueNumber: testQueueNum });
        // Ensure queue currentNumber is behind
        await Queue.findByIdAndUpdate(testQueue._id, { currentNumber: testQueueNum - 1 });

        const statusUpdate = { status: 'completed' };      You, yesterday • modified tests
        const res = await request(app)
            .put(`/api/orders/${testOrder._id}/status`)
            .set('x-auth-token', tokenOwner)
            .send(statusUpdate);

        expect(res.statusCode).toBe(200);
        expect(res.body.status).toBe('completed');

        const updatedQueue = await Queue.findById(testQueue._id);
        // Controller logic advances currentNumber TO the completed order's number
        expect(updatedQueue.currentNumber).toBe(testQueueNum);
     });

     it('should NOT update queue currentNumber if status is not completed/cancelled', async () => {
         const initialQueue = await Queue.findById(testQueue._id);
         const statusUpdate = { status: 'ready' }; // Update to something other than completed/cancelled
         const res = await request(app)
            .put(`/api/orders/${testOrder._id}/status`)
            .set('x-auth-token', tokenOwner)
```

```
                .send(statusUpdate);

            expect(res.statusCode).toBe(200);
            expect(res.body.status).toBe('ready');

            const finalQueue = await Queue.findById(testQueue._id);
            expect(finalQueue.currentNumber).toBe(initialQueue.currentNumber); // Should not change
        });


        it('should return 401 if user is not the stall owner', async () => {
            const statusUpdate = { status: 'ready' };
            const res = await request(app)
                .put(`/api/orders/${testOrder._id}/status`)
                .set('x-auth-token', tokenCustomer) // Use customer token
                .send(statusUpdate);
            expect(res.statusCode).toBe(401);
        });

        it('should return 404 if order ID is invalid', async () => {
            const invalidMongoId = new mongoose.Types.ObjectId().toString();
            const statusUpdate = { status: 'ready' };
            const res = await request(app)
                .put(`/api/orders/${invalidMongoId}/status`)
                .set('x-auth-token', tokenOwner)
                .send(statusUpdate);
            expect(res.statusCode).toBe(404);
        });
         it('should return 401 if not authenticated', async () => {
             const statusUpdate = { status: 'ready' };
             const res = await request(app)
                .put(`/api/orders/${testOrder._id}/status`)
                .send(statusUpdate); // No token
             expect(res.statusCode).toBe(401);
        });
    });
```

2. Control Flow Graph



3. Cyclomatic Complexity

Cyclomatic Complexity (CC) is calculated as the number of binary decision points + 1.
From the control flow diagram, there are four decision points:
1. Order cancellation check

2. User authentication check
3. Order ID validity check
4. Order completion check

Thus,
Cyclomatic Complexity (CC) = 4 + 1 = 5

4. Basis Paths

The following are the independent basis paths derived from the control flow graph:

· Path #1 (Baseline): Authenticated stall owner successfully updates status to 'preparing'.

· Path #2: Status updated to 'completed', triggering queue currentNumber update.

· Path #3: Status updated to a non-terminal state like 'ready', queue remains unchanged.

· Path #4: User is authenticated but not a stall owner — receives 401 Unauthorized.

· Path #5: Authenticated user provides an invalid order ID — receives 404.

· Path #6: Request made without authentication — receives 401 Unauthorized.

5. Test Cases and Results

Test Case Name	Test Input	Expected Output	Actual Output	Test Result
OrderStatus-01	status = "preparing", valid orderId, authenticated stall owner	Status update successful, status = "preparing"	Status update successful, status = "preparing"	Pass
OrderStatus-02	status = "completed", valid orderId, queueNumber = 55, queue currentNumber = 54	Queue currentNumber updated to 55, status = "completed"	Queue currentNumber updated to 55, status = "completed"	Pass
OrderStatus-03	status = "ready", valid orderId, queue currentNumber	Queue currentNumber unchanged, status = "ready"	Queue currentNumber unchanged, status = "ready"	Pass

	should not change			
OrderStatus-04	status = "ready", valid orderId, authenticated non-stall owner	Error 401: Unauthorized user	Error 401: Unauthorized user	Pass
OrderStatus-05	status = "ready", invalid orderId	Error 404: Order ID invalid	Error 404: Order ID invalid	Pass
OrderStatus-06	status = "ready", valid orderId, no authentication token	Error 401: Not authenticated	Error 401: Not authenticated	Pass

```
PASS tests/controllers/queueController.test.js
  Order Controller Tests (Using Test DB & Selective Cleanup)
    POST /api/orders (createOrder)
      √ should create a new order successfully (79 ms)
      √ should create a loyalty record if none exists (51 ms)
      √ should update an existing loyalty record (56 ms)
      √ should return 404 if stallId does not exist (11 ms)
      √ should return 400 if queue is not active (32 ms)
      √ should return 401 if user is not authenticated (8 ms)
    GET /api/orders/user (getUserOrders)
      √ should return orders placed by the authenticated user (25 ms)
      √ should return an empty array if the authenticated user has no orders (14 ms)
      √ should return 401 if not authenticated (4 ms)
    GET /api/orders/stall/:stallId (getStallOrders)
      √ should return non-completed orders for the stall owner (28 ms)
      √ should return 401 if user is not the stall owner (7 ms)
      √ should return 401 if stall does not exist or owner mismatch (6 ms)
      √ should return 401 if not authenticated (2 ms)
    PUT /api/orders/:orderId/status (updateOrderStatus)
      √ should allow stall owner to update status (e.g., to preparing) (31 ms)
      √ should update queue currentNumber when order is COMPLETED (49 ms)
      √ should NOT update queue currentNumber if status is not completed/cancelled (38 ms)
      √ should return 401 if user is not the stall owner (12 ms)
      √ should return 404 if order ID is invalid (8 ms)
      √ should return 401 if not authenticated (2 ms)
    PUT /api/orders/:orderId/cancel (cancelOrder)
      √ should allow customer to cancel a PENDING order (37 ms)
      √ should NOT allow customer to cancel if status is NOT pending (38 ms)
      √ should return 401 if user did not place the order (22 ms)
      √ should return 404 if order ID is invalid (21 ms)
      √ should return 401 if not authenticated (19 ms)

Test Suites: 1 passed, 1 total
Tests:       24 passed, 24 total
Snapshots:   0 total
Time:        2.347 s, estimated 30 s
```