IntegrationTesting

Project: Point of Sale (POS) System

Worked by: Ester Shumeli

Working Date: 18/01/2026

Accepted by: Ari Gjerazi, Jurgen Cama

Team: Ester Shumeli, Abdulaziz Bezan, Eridjon Krrashi, Evisa Nela

Integration Test 2: User authentication (login verification)

Type: Database Integration Testing.
Method: Decomposition-Based Integration Testing(testing the interaction between the application logic and the database layer during login/authentication).
Objective:
The purpose of this test is to verify that the application can authenticate a user by validating the entered username and password against the records stored in the MySQL database. The test ensures that the integration between the login logic (application layer) and the pos.users table (database layer) works correctly for both valid and invalid credentials

Test Scenario
Preconditions

-The test is run against a **test database** (`test_pos`) to ensure no impact on production data.

-The `pos.users` table exists and has the following structure:

-id (INT, AUTO_INCREMENT, PRIMARY KEY)
-username (VARCHAR, NOT NULL)
-password (VARCHAR, NOT NULL)
-role (VARCHAR, NOT NULL)

-A **test user record is inserted before the test** runs to simulate a real system login scenario.

Test Steps

1. **Create a Test User Record:** Insert a user into the database (e.g, username = `"it_user_xxx"`, password = `"pass12345"`, role = `"admin"`).
2. **Authenticate with Valid Credentials:** Execute a query using both correct username and correct password.
3. **Authenticate with Invalid Password:** Execute a query using correct username but wrong password.
4. **Authenticate with Unknown Username:** Execute a query using a username that does not exist in the table.
5. **Cleanup:** Delete the inserted test user from the database after the test ends.

Test Step Table

Test Step	Action	Expected Outcome
Create Test User Record	Insert a test user into `pos.users`	A user row is inserted successfully
Valid Login Attempt	Query DB with correct username + correct password	User is found → authentication succeeds
Invalid Password Attempt	Query DB with correct username + wrong password	No user found → authentication fails
Unknown Username Attempt	Query DB with unknown username + any password	No user found → authentication fails
Cleanup	Delete test user from database	Database returns to original state

Expected Outcome

- Valid credentials should successfully match a record in the database and return a positive authentication result.
- Invalid credentials (wrong password or unknown username) should not match any record and should return a negative authentication result.
- No unexpected SQL errors should occur during authentication checks, and test data should be removed after execution.

Test Flow

Test Flow	Action	Expected Behavior	Actual Outcome
Positive Flow	Authenticate user using correct username and correct password	User record is found in the database and authentication succeeds	Passed: User authenticated successfully
Negative Flow	Authenticate user using correct username and wrong password	No matching record is found and authentication fails	Passed: Authentication failed as expected
Negative Flow	Authenticate user using unknown username	No matching record is found and authentication fails	Passed: Authentication failed as expected

Testing the positive flow, with valid input:

In the positive flow, the system uses the **correct username and correct password** for the inserted test user. The database query returns a matching row, confirming that the application can correctly authenticate valid users through the database.

Testing the negative flow, with invalid input:

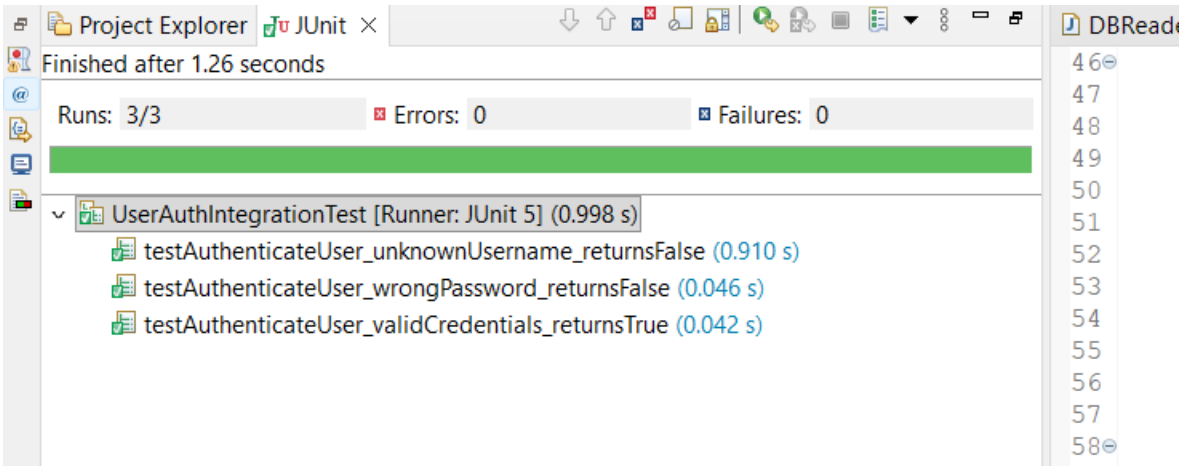
In the negative flow, the system attempts authentication with:

1. a correct username but an incorrect password, and
2. an unknown username.
   In both cases, the database returns **no matching record**, confirming that invalid logins are correctly rejected.


DBReader.java
IDALManager...
ObjectMappe...
ApplicationS...
POS.java
MySQLConnec...
UserIntegra...
UserAuthInte...
1 package IntegrationTesting;
2
3 import org.junit.After;
4 import org.junit.Before;
5 import org.junit.Test;
6
7 import java.sql.*;
8 import java.util.UUID;
9
10 import static org.junit.Assert.*;
11
12 public class UserAuthIntegrationTest {
13
14 private Connection conn;
15
16 // Use unique username so tests don't clash with existing data
17 private String testUsername;
18 private String correctPassword;
19 private String role;
20
21 @Before
22 public void setUp() throws Exception {
23 conn = DriverManager.getConnection(
24 "jdbc:mysql://localhost:3306/test_pos",
25 "root",
26 "centralcee23" // <-- change if needed
27);
28
29 testUsername = "it_user_" + UUID.randomUUID().toString().substring(0, 8);
30 correctPassword = "pass12345";
31 role = "admin";
32
33 // Insert a known user for authentication tests
34 String insertSql = "INSERT INTO pos.users (username, password, role) VALUES (?, ?, ?)";
35 try (PreparedStatement ps = conn.prepareStatement(insertSql)) {
36 ps.setString(1, testUsername);
37 ps.setString(2, correctPassword);
38 ps.setString(3, role);
39
40 int rows = ps.executeUpdate();
41 assertEquals("Setup failed: user insert did not insert exactly 1 row", 1, rows);
42 }
43 }

```java
    }

    // Helper method that simulates "login check"
    private boolean authenticate(String username, String password) throws SQLException {
        String sql = "SELECT 1 FROM pos.users WHERE username = ? AND password = ? LIMIT 1";
        try (PreparedStatement ps = conn.prepareStatement(sql)) {
            ps.setString(1, username);
            ps.setString(2, password);

            try (ResultSet rs = ps.executeQuery()) {
                return rs.next(); // true if a match exists
            }
        }
    }

    @Test
    public void testAuthenticateUser_validCredentials_returnsTrue() throws Exception {
        boolean result = authenticate(testUsername, correctPassword);
        assertTrue("Expected authentication to succeed with correct credentials", result);
    }

    @Test
    public void testAuthenticateUser_wrongPassword_returnsFalse() throws Exception {
        boolean result = authenticate(testUsername, "wrongPassword!");
        assertFalse("Expected authentication to fail with wrong password", result);
    }

    @Test
    public void testAuthenticateUser_unknownUsername_returnsFalse() throws Exception {
        boolean result = authenticate("no_such_user_123", correctPassword);
        assertFalse("Expected authentication to fail for unknown username", result);
    }

    @After
    public void tearDown() throws Exception {
        // Clean inserted test user
        String deleteSql = "DELETE FROM pos.users WHERE username = ?";
        try (PreparedStatement ps = conn.prepareStatement(deleteSql)) {
            ps.setString(1, testUsername);
            ps.executeUpdate();
        }

        if (conn != null && !conn.isClosed()) conn.close();
```

Writable | Smart Insert | 88 : 1 : 3090



Evidence
JUnit Result: Runs = 3, Errors = 0, Failures = 0
The authentication integration tests executed successfully, confirming that the application correctly validates user credentials against the database and handles invalid login attempts as expected."

What each test confirms

1. testAuthenticateUser_validCredentials_returnsTrue -> **Positive Flow**
-Uses correct username and correct password
-Database returns a matching record

-Authentication succeeds
The application correctly integrates with the database to authenticate valid users.

2. testAuthenticateUser_wrongPassword_returnsFalse -> **Negative Flow**

-Username exists, password is incorrect
-Database returns no matching record
-Authentication fails as expected
The system properly rejects incorrect credentials and does not allow unauthorized access.

3. testAuthenticateUser_unknownUsername_returnsFalse -> **Negative Flow**

-Username does not exist
-Database returns no result
-Authentication fails correctly
The system handles non-existing users safely and correctly.

Integration Test 3: Checkout process – stock validation and sale saving

Type: Integration Testing
Method: Top-Down Integration Testing

Objective:

The purpose of this test is to verify that the **checkout process** of the Point of Sale (POS) system functions correctly when integrating high-level business logic with lower-level components. This test focuses on validating product stock availability, calculating the total sale amount, updating stock quantities, and saving the sale transaction.
Lower-level database components are replaced with **stubs**, allowing early validation of the checkout logic without relying on the real database.

Top-Down: Checkout → calculates total + updates stock (via stubs)
What is integrated
High-level: CheckoutService (simulated higher module)
Lower-level dependencies (stubbed):
ProductDAL (stock read/update)
SaleDAL (save sale)
So I am testing the "root" logic first

Test Scenario

Preconditions

- The test is executed without using a real database.
- Lower-level components responsible for database access are replaced by **stub classes**.
- Test data (products, prices, and stock quantities) is preloaded into the stubs before test execution.
- The checkout logic is executed through a high-level service component.

Test Steps

1. **Initialize Stub Components:** Create stub implementations for product data access and sale persistence.

2. **Seed Product Data:** Preload product prices and stock quantities into the product stub.
3. **Execute Checkout (Valid Case):** Provide a set of products and quantities where sufficient stock exists.
4. **Validate Checkout Result:** Verify that the total price is calculated correctly, stock quantities are updated, and the sale is saved.
5. **Execute Checkout (Invalid Case):** Attempt checkout with insufficient product stock.
6. **Validate Failure Behavior:** Ensure an exception is thrown, stock remains unchanged, and the sale is not saved.

Test Step Table

Test Step	Action	Expected Outcome
Initialize Stubs	Create ProductDALStub and SaleDALStub	Stub components are ready
Seed Product Data	Insert product prices and stock into stub	Product data is available
Valid Checkout	Checkout with sufficient stock	Total calculated, stock reduced, sale saved
Validate Result	Verify total, stock, and saved sale	Data matches expected values
Invalid Checkout	Checkout with insufficient stock	Exception is thrown
Validate Failure	Check stock and sale status	Stock unchanged, sale not saved

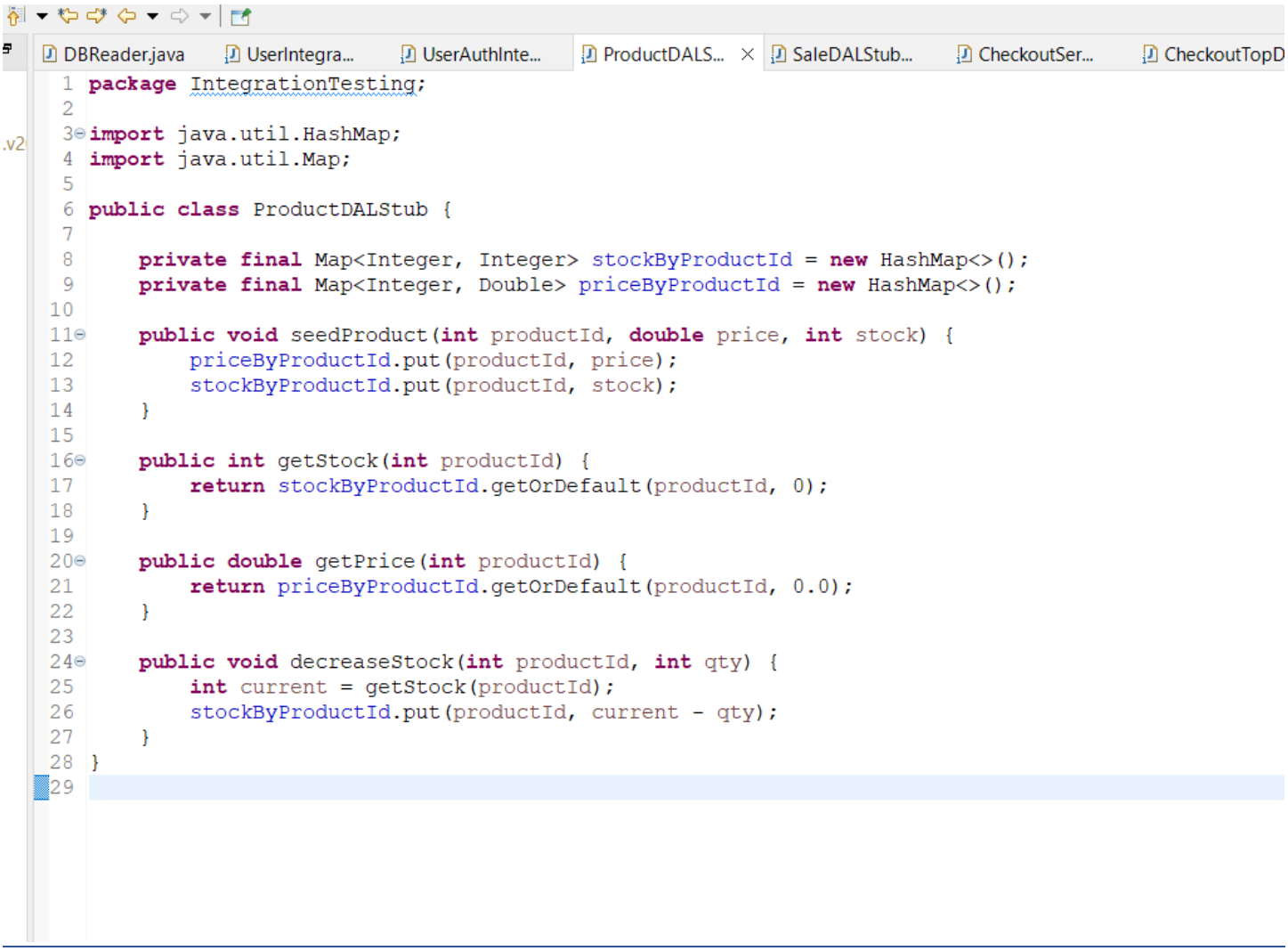
Expected Outcome

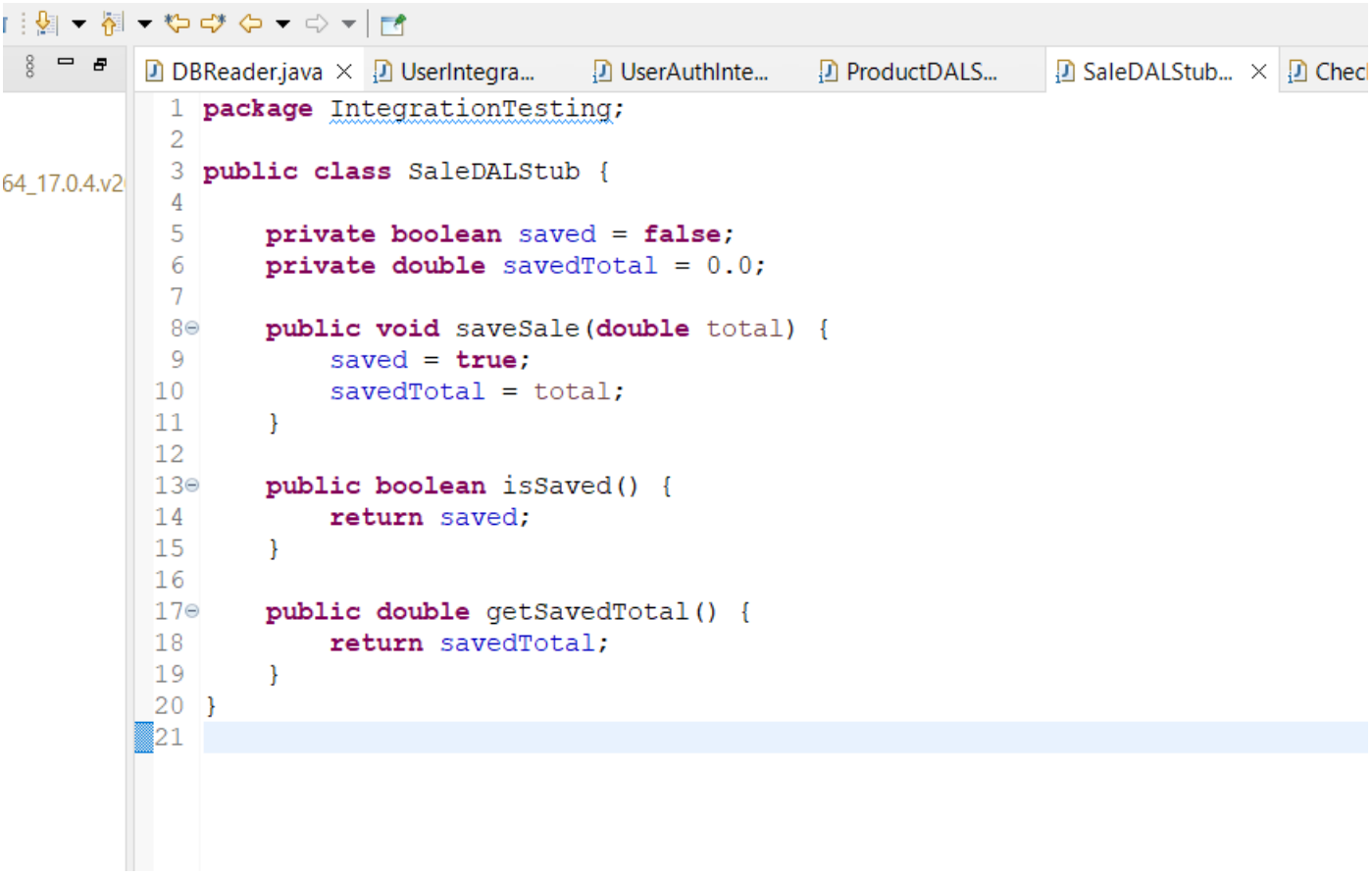
- When valid items are provided, the checkout process should:

Calculate the correct total price, Reduce product stock accordingly, Save the sale successfully

- When invalid items (insufficient stock) are provided:

An exception should be thrown, No stock updates should occur, The sale should not be saved


DBReader.java
UserIntegra...
UserAuthInte...
ProductDALS...
SaleDALStub...
CheckoutSer...
CheckoutTopD
.v2
1 package IntegrationTesting;
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 public class ProductDALStub {
7
8 private final Map<Integer, Integer> stockByProductId = new HashMap<>();
9 private final Map<Integer, Double> priceByProductId = new HashMap<>();
10
11 public void seedProduct(int productId, double price, int stock) {
12 priceByProductId.put(productId, price);
13 stockByProductId.put(productId, stock);
14 }
15
16 public int getStock(int productId) {
17 return stockByProductId.getOrDefault(productId, 0);
18 }
19
20 public double getPrice(int productId) {
21 return priceByProductId.getOrDefault(productId, 0.0);
22 }
23
24 public void decreaseStock(int productId, int qty) {
25 int current = getStock(productId);
26 stockByProductId.put(productId, current - qty);
27 }
28 }
29


DBReader.java
UserIntegra...
UserAuthInte...
ProductDALS...
SaleDALStub...
Chec
64_17.0.4.v2
1 package IntegrationTesting;
2
3 public class SaleDALStub {
4
5 private boolean saved = false;
6 private double savedTotal = 0.0;
7
8 public void saveSale(double total) {
9 saved = true;
10 savedTotal = total;
11 }
12
13 public boolean isSaved() {
14 return saved;
15 }
16
17 public double getSavedTotal() {
18 return savedTotal;
19 }
20 }
21

DBReader.java | UserIntegra... | UserAuthInte... | ProductDALS... | SaleDALStub... | CheckoutSer... | CheckoutTopD...

```java
package IntegrationTesting;

import java.util.Map;

public class CheckoutService {

    private final ProductDALStub productDal;
    private final SaleDALStub saleDal;

    public CheckoutService(ProductDALStub productDal, SaleDALStub saleDal) {
        this.productDal = productDal;
        this.saleDal = saleDal;
    }

    // items: productId -> quantity
    public double checkout(Map<Integer, Integer> items) {
        // 1) Validate stock
        for (Map.Entry<Integer, Integer> e : items.entrySet()) {
            int productId = e.getKey();
            int qty = e.getValue();
            if (qty <= 0) throw new IllegalArgumentException("Quantity must be > 0");
            if (productDal.getStock(productId) < qty) {
                throw new IllegalStateException("Insufficient stock for product " + productId);
            }
        }

        // 2) Calculate total + update stock
        double total = 0.0;
        for (Map.Entry<Integer, Integer> e : items.entrySet()) {
            int productId = e.getKey();
            int qty = e.getValue();
            total += productDal.getPrice(productId) * qty;
            productDal.decreaseStock(productId, qty);
        }

        // 3) Save sale (stubbed persistence)
        saleDal.saveSale(total);

        return total;
    }
}
```

Writable | Smart Insert | 42 : 1 : 1339

DBReader.java | UserIntegra... | UserAuthInte... | ProductDALS... | SaleDALStub... | CheckoutSer... | CheckoutTopD...

```java
package IntegrationTesting;

import org.junit.Test;

import java.util.HashMap;
import java.util.Map;

import static org.junit.Assert.*;

public class CheckoutTopDownIntegrationTest {

    @Test
    public void checkout_validItems_updatesStock_andSavesSale() {
        // Arrange (stubs)
        ProductDALStub productDal = new ProductDALStub();
        SaleDALStub saleDal = new SaleDALStub();

        // Seed products: id, price, stock
        productDal.seedProduct(101, 50.0, 10); // product 101: price 50, stock 10
        productDal.seedProduct(202, 20.0, 5);  // product 202: price 20, stock 5

        CheckoutService service = new CheckoutService(productDal, saleDal);

        Map<Integer, Integer> cart = new HashMap<>();
        cart.put(101, 2); // 2 * 50
        cart.put(202, 1); // 1 * 20

        // Act
        double total = service.checkout(cart);

        // Assert total
        assertEquals(120.0, total, 0.0001);

        // Assert stock updated
        assertEquals(8, productDal.getStock(101)); // 10 - 2
        assertEquals(4, productDal.getStock(202)); // 5 - 1

        // Assert sale saved
        assertTrue(saleDal.isSaved());
        assertEquals(120.0, saleDal.getSavedTotal(), 0.0001);
    }

    @Test
    public void checkout_insufficientStock_throwsException_andDoesNotSaveSale() {
        ProductDALStub productDal = new ProductDALStub();
        SaleDALStub saleDal = new SaleDALStub();

        productDal.seedProduct(101, 50.0, 1); // only 1 in stock
        CheckoutService service = new CheckoutService(productDal, saleDal);

        Map<Integer, Integer> cart = new HashMap<>();
        cart.put(101, 2); // request 2 -> should fail

        try {
            service.checkout(cart);
            fail("Expected exception due to insufficient stock");
        } catch (IllegalStateException ex) {
            assertTrue(ex.getMessage().contains("Insufficient stock"));
        }

        assertFalse("Sale should not be saved when checkout fails", saleDal.isSaved());
        assertEquals(1, productDal.getStock(101)); // stock unchanged
    }
}
```

Test Flow Table

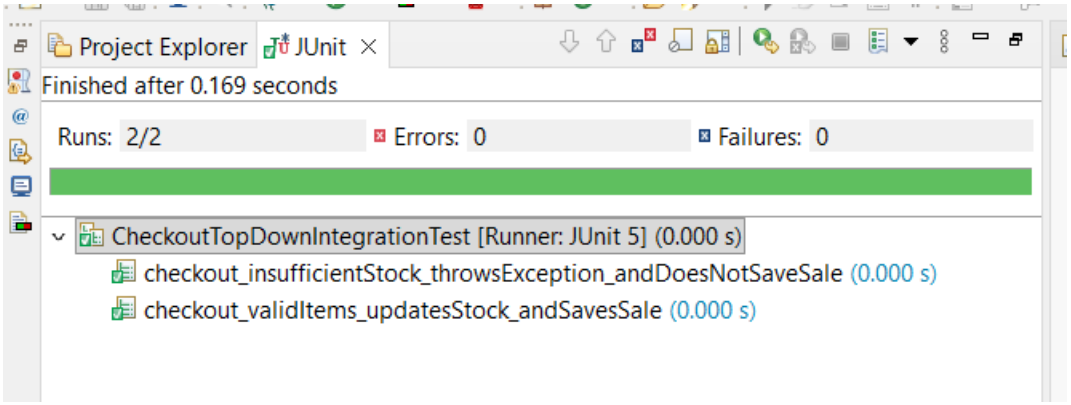
Test Flow	Action	Expected Behavior	Actual Outcome
Positive Flow	Checkout with valid items and sufficient stock	Checkout succeeds, stock updates, sale saved	Passed: Checkout completed successfully
Negative Flow	Checkout with insufficient stock	Exception thrown, no stock update, no sale saved	Passed: Checkout correctly blocked

Testing the positive flow, with valid input:
In the positive flow, products with sufficient stock are processed through the checkout service. The system calculates the total sale amount, updates the product stock, and saves the sale using the stubbed persistence layer. This confirms that the high-level checkout logic integrates correctly with its lower-level components.

Testing the negative flow, with invalid input:
In the negative flow, the checkout process is attempted with quantities exceeding available stock. The system throws an exception, prevents stock modification, and stops the sale from being saved. This confirms that the checkout logic correctly handles invalid transactions.

Integration Strategy
This test follows a **Top-Down Integration Testing** strategy. The integration begins with the high-level checkout logic, while lower-level database-related components are replaced by **stubs**. This approach allows early testing of business rules and control flow without requiring the database layer to be fully implemented.

Conclusion



Evidence: JUnit Result: Runs = 2, Errors = 0, Failures = 0

The checkout integration test executed successfully for both positive and negative scenarios. The results confirm that the POS system's checkout logic behaves correctly and enforces business rules before interacting with lower-level components. This test demonstrates effective use of the **Top-Down Integration Testing** strategy.