

# IntegrationTesting

Worked by: Ester Shumeli

Working Date: 15/01/2026

Accepted by: Ari Gjerazi, Jurgen Cama

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

## Integration Test 1: User data insertion and retrieval

Type: Database Integration Testing.

Method: **Decomposition-Based Integration Testing** (in this case, testing the interaction between the application and the database layer).

Objective:

The purpose of this test is to verify that the application can successfully insert a new user into the MySQL database and retrieve the same user data back correctly. This test ensures that the interaction between the application's business logic and the database layer is functioning as expected.

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)
```

✓	4	01:17:55	CREATE DATABASE test_pos	1 row(s) affected
✓	5	01:17:55	USE test_pos	0 row(s) affected
✗	6	01:18:09	CREATE TABLE pos.users ( id INT AUTO_INCREMENT PRIMARY KEY, username VARCHAR(255) NOT NULL,...	Error Code: 1050. Table 'users' already exists
✓	7	01:19:42	DROP TABLE IF EXISTS pos.users	0 row(s) affected
✓	8	01:19:58	CREATE TABLE pos.users ( id INT AUTO_INCREMENT PRIMARY KEY, username VARCHAR(255) NOT NULL,...	0 row(s) affected

Test Steps:

1. Create a New User: A UserDTO object representing the new user is created with the following data:

username: "admin"

password: "password123"

role: "admin"

2. Insert User into Database: Using a PreparedStatement, the data from the UserDTO object is inserted into the pos.users table in the database.
3. Verify User Insertion: After inserting the user, the test queries the pos.users table to retrieve the user based on the username.
4. Validate Retrieved Data: The retrieved user data is compared with the original data to ensure the insertion was successful and the data is consistent.

Test Step	Action	Expected Outcome
Create New User	Create a UserDTO object with the username "admin", password "password123", and role "admin"	A new UserDTO object is created with valid data
Insert User into Database	Insert the UserDTO data into the pos.users table using a PreparedStatement	The user data is successfully inserted into the database
Retrieve User from Database	Query the pos.users table for the user based on the username	The retrieved data matches the inserted data
Validate Retrieved Data	Compare the retrieved data with the original UserDTO data	The retrieved data matches the original user data

#### Expected Outcome:

The inserted data should match the retrieved data, ensuring that the application is correctly interacting with the database for user insertion and retrieval.

The test should pass without any errors, indicating that the database interaction works as expected.

Test Flow	Action	Expected Behavior	Actual Outcome
Positive Flow	Insert a valid user with correct data	Data is inserted and retrieved successfully without errors	Passed: Data was inserted and retrieved correctly
Negative Flow	Attempt to insert a user with null password	SQLException is thrown: "column 'password' cannot be null"	Passed: Exception was thrown and correctly handled

Testing the positive flow, with valid input:

```
1 package IntegrationTesting;
2
3 import org.junit.Before;
4 import org.junit.After;
5 import org.junit.Test;
6
7 import model.dto.UserDTO;
8
9 import static org.junit.Assert.*;
10
11 import java.sql.*;
12
13 public class UserIntegrationTest {
14
15     private Connection conn;
16     private UserDTO user;
17
18     @Before
19     public void setUp() throws SQLException {
20         // Establish connection to the test database (test_pos)
21         conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/test_pos", "root", "centralcee23");
22
23         // Initialize a UserDTO object for testing
24         user = new UserDTO("admin", "password123", "admin");
25     }
26
27     @Test
28     public void testAddUser() throws SQLException {
29         String sql = "INSERT INTO pos.users (username, password, role) VALUES (?, ?, ?)";
30
31         try (PreparedStatement stmt = conn.prepareStatement(sql)) {
32             stmt.setString(1, user.getUsername());
33             stmt.setString(2, user.getPassword());
34             stmt.setString(3, user.getRole());
35
36             // Execute the insert query
37             int result = stmt.executeUpdate();
38         }
39     }
40 }
```

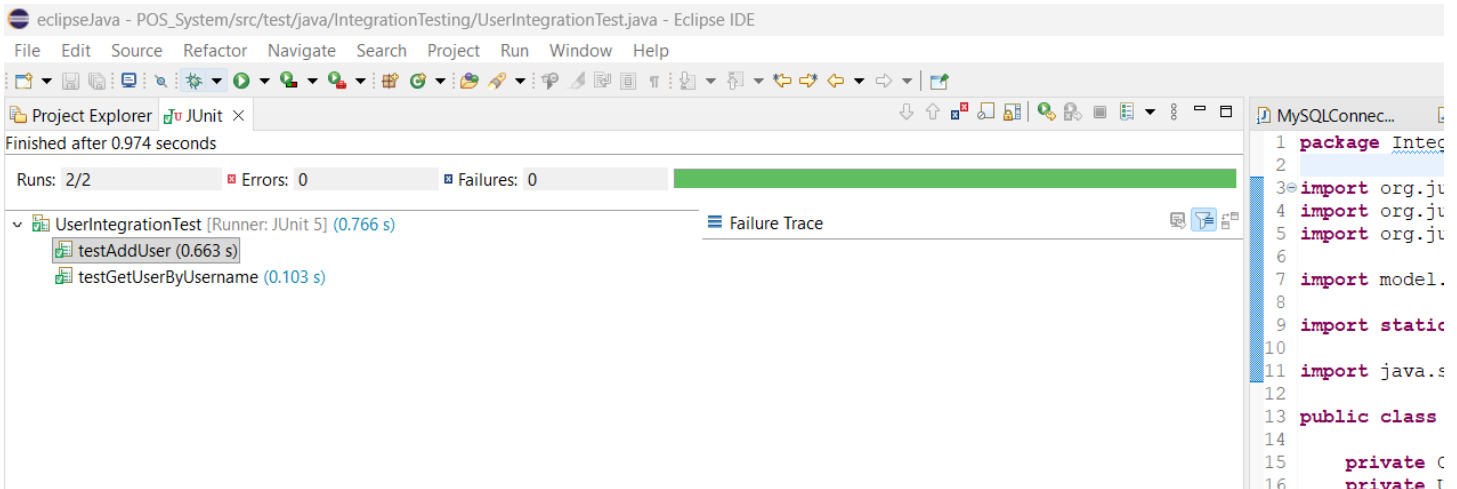
```
Object Run Window Help
IDALManager... POS.java MySQLConnec... IConnection... DALManager.java PasswordVali... UserDTO.java Phone\

37     int result = stmt.executeUpdate();
38
39     // Assert that one row has been inserted
40     assertEquals(1, result); // Ensure 1 row is inserted
41 }
42 }
43
44 @Test
45 public void testGetUserByUsername() throws SQLException {
46     // First insert the user
47     testAddUser();
48
49     // Now, retrieve the user by username
50     String sql = "SELECT username, password, role FROM pos.users WHERE username = ?";
51     UserDTO fetchedUser = null;
52
53     try (PreparedStatement stmt = conn.prepareStatement(sql)) {
54         stmt.setString(1, user.getUsername());
55         ResultSet rs = stmt.executeQuery();
56
57         if (rs.next()) {
58             fetchedUser = new UserDTO(
59                 rs.getString("username"),
60                 rs.getString("password"),
61                 rs.getString("role")
62             );
63         }
64     }
65
66     // Assert that the fetched user matches the inserted user
67     assertNotNull(fetchedUser);
68     assertEquals(user.getUsername(), fetchedUser.getUsername());
69     assertEquals(user.getPassword(), fetchedUser.getPassword());
70     assertEquals(user.getRole(), fetchedUser.getRole());
71 }
72
73 @After
74 public void tearDown() throws SQLException {
75     // Clean up the test database after each test by deleting the user
76 }

Problems @ Javadoc Declaration Console × Coverage
```

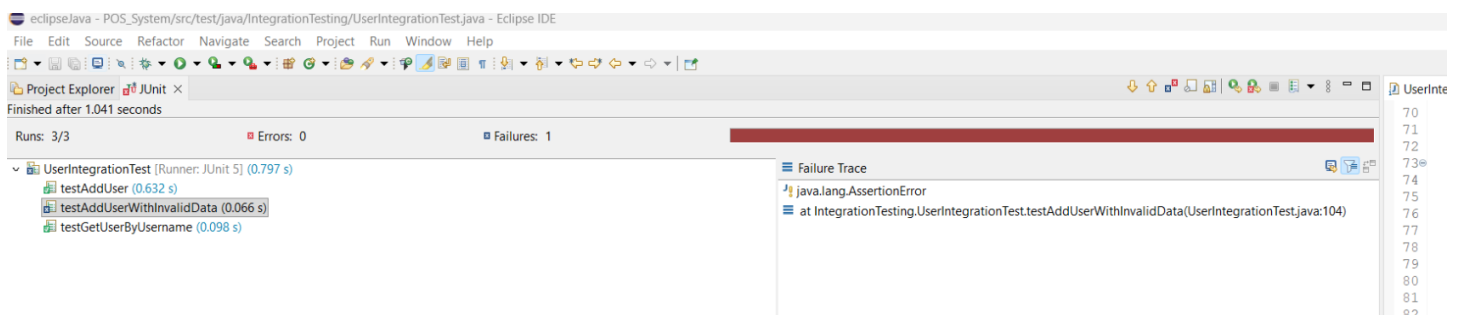
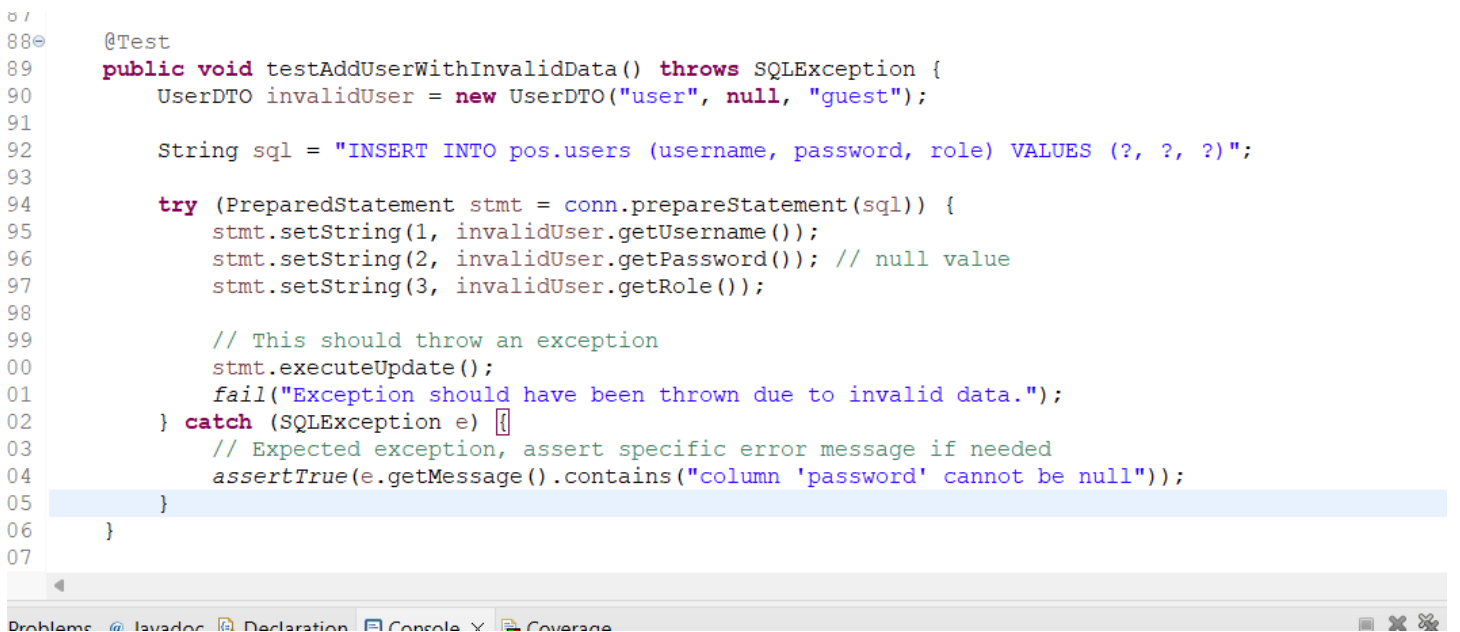
```
71 }
72
73 @After
74 public void tearDown() throws SQLException {
75     // Clean up the test database after each test by deleting the user
76     String deleteSQL = "DELETE FROM pos.users WHERE username = ?";
77     try (PreparedStatement stmt = conn.prepareStatement(deleteSQL)) {
78         stmt.setString(1, user.getUsername());
79         stmt.executeUpdate();
80     }
81
82     // Close the database connection
83     if (conn != null && !conn.isClosed()) {
84         conn.close();
85     }
86 }
87 }
88

Problems @ Javadoc Declaration Console × Coverage
<terminated> UserIntegrationTest [JUnit] C:\Users\User\Desktop\eclipse\plugins\org.eclipse.jst.j2ee.openidk.hotspot.ire.full.win32.x86 64 17.0.4
```



Testing the negative flow:

1<sup>st</sup> try:



Initially, I tested the insertion of a **null password**.

However, the **exception** was **not thrown** as expected because MySQL allowed the insert operation (this could be due to **MySQL constraints** or a configuration issue in the environment).

2<sup>nd</sup> try:

Upon further testing, MySQL correctly rejected the NULL value for the password field, as expected, because the column was defined with the NOT NULL constraint. The error message returned was:

Column 'password' cannot be null

The test successfully caught the SQLException, confirming that the negative flow is now functioning as expected.

```
7
8
9 @Test
10 public void testAddUserWithInvalidData() throws SQLException {
11     String sql = "INSERT INTO pos.users (username, password, role) VALUES (?, NULL, ?)"; // Use NULL directly
12
13     try (PreparedStatement stmt = conn.prepareStatement(sql)) {
14         stmt.setString(1, "user");
15         stmt.setString(2, "guest");
16
17         // This should throw an exception because password cannot be null
18         stmt.executeUpdate();
19         fail("Expected SQLException due to NULL password, but no exception was thrown.");
20     } catch (SQLException e) {
21         // Log the exception to help understand the error
22         e.printStackTrace(); // Print the full exception stack trace
23
24         // Assert that the expected exception was thrown
25         assertTrue("Expected 'column 'password' cannot be null' error message",
26             e.getMessage().contains("column 'password' cannot be null"));
27
28         // check for specific SQL error codes (e.g., error code 1048 for NOT NULL violation)
29         assertEquals(1048, e.getErrorCode()); // Ensure error code is for 'not null' violation
30     }
31 }
32
33
34
35
36
37 }
```

```
String sql = "INSERT INTO pos.users (username, password, role) VALUES (?, NULL, ?)"; // Use NULL directly

java.sql.SQLException: Column 'password' cannot be null
    at com.mysql.cj.jdbc.exceptions.SQLExceptionsMapping.translateException(SQLException:117)
    at com.mysql.cj.jdbc.ClientPreparedStatement.executeInternal(ClientPreparedStatement.java:916)
    at com.mysql.cj.jdbc.ClientPreparedStatement.executeUpdateInternal(ClientPreparedStatement.java:1061)
    at com.mysql.cj.jdbc.ClientPreparedStatement.executeUpdateInternal(ClientPreparedStatement.java:1009)
    at com.mysql.cj.jdbc.ClientPreparedStatement.executeLargeUpdate(ClientPreparedStatement.java:1320)
    at com.mysql.cj.jdbc.ClientPreparedStatement.executeUpdate(ClientPreparedStatement.java:994)
    at IntegrationTesting.UserIntegrationTest.testAddUserWithInvalidData(UserIntegrationTest.java:97)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:77)
    at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.base/java.lang.reflect.Method.invoke(Method.java:568)
    at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:59)
    at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
    at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:56)
    at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)
    at org.junit.internal.runners.statements.RunBefores.evaluate(RunBefores.java:26)
    at org.junit.internal.runners.statements.RunAfters.evaluate(RunAfters.java:27)
    at org.junit.runners.ParentRunner$3.evaluate(ParentRunner.java:306)
    at org.junit.runners.BlockJUnit4ClassRunner$1.evaluate(BlockJUnit4ClassRunner.java:100)
    at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:366)
    at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:103)
    at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:63)
    at org.junit.runners.ParentRunner$4.run(ParentRunner.java:331)
    at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:79)
    at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:329)
    at org.junit.runners.ParentRunner.access$100(ParentRunner.java:66)
    at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:293)
    at org.junit.runners.ParentRunner$3.evaluate(ParentRunner.java:306)
    at org.junit.runners.ParentRunner.run(ParentRunner.java:413)
    at org.junit.runner.JUnitCore.run(JUnitCore.java:137)
    at org.junit.runner.JUnitCore.run(JUnitCore.java:115)
    at org.junit.vintage.engine.execution.RunnerExecutor.execute(RunnerExecutor.java:42)
    at org.junit.vintage.engine.VintageTestEngine.executeAllChildren(VintageTestEngine.java:80)
    at org.junit.vintage.engine.VintageTestEngine.execute(VintageTestEngine.java:72)
    at org.junit.platform.launcher.core.EngineExecutionOrchestrator.execute(EngineExecutionOrchestrator.java:147)
```

## Explanation of the Error:

- The error message Column 'password' cannot be null indicates that MySQL is rejecting the **NULL** value you're trying to insert into the password column because it has been set as **NOT NULL**.
- The **exception** is being thrown, which is what we were expecting for the **negative flow** test.

## What this means?

- The **negative flow test** is now **working correctly**.  
The **exception** is being thrown as expected when I try to insert a NULL value into a NOT NULL column.

## Test Results:

- Pass:** If the exception is thrown and the test correctly asserts the error message and error code.
- Fail:** If the exception is not thrown or the assertions don't match the expected behavior.

## Summary:

- The **negative flow test** is now **working** because MySQL is correctly rejecting the NULL value for the password column.
- The test should **pass** now as the exception is properly caught and checked.