IntegrationTesting

Worked by: Ester Shumeli

Working Date: 19/01/2026

Accepted by: Ari Gjerazi, Jurgen Cama

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

## IntegrationTest7: Product + Supplier Integration (FK link + DAL search + DB verification)

**Type:** Database Integration Testing
**Method / Strategy: Bottom-Up Integration Testing** (DAL-first integration, then verified against DB state)
**Test Class:** ProductSupplierIntegrationTest.java
**Subsystem Focus:** Inventory + Supplier linkage (`products` ↔ `suppliers`)

Objective
The purpose of this test is to verify that the POS system correctly integrates **Supplier persistence** and **Product persistence/search** such that:

1. A **Supplier** can be inserted through the real **Data Access Layer (DAL)**.
2. A **Product** can be inserted through the DAL and correctly linked to the Supplier using the **foreign key**.
3. The inserted Product can be retrieved using the system's **searchProductsByName(...)** logic.
4. The Product's Supplier linkage is confirmed using a **database-level verification query** (source of truth).
5. All inserted test data is removed at the end, restoring the DB to the original state.

This test confirms correct integration between:
-Supplier module (save/delete supplier via DAL),
-Product module (add/search/delete product via DAL),
-Database referential integrity (FK field stored correctly in products.suppliers_id),
-DAL mapping assumptions vs actual database column names.

No mocks are used. The test verifies real end-to-end behavior of the subsystem inside the POS environment.

Integration strategy explanation (Bottom-Up)

This test follows a **Bottom-Up Integration Testing strategy** because testing begins from the lower-level subsystem that directly interacts with the database:

- The **DALManager** is treated as the low-level integration point ("leaf" layer) because it performs real SQL operations via internal DB components.
- The test integrates upward by validating higher-level behavior (search logic + relationship integrity).
- A JUnit driver triggers the workflow and asserts correctness across multiple integrated components.

Preconditions

1. MySQL server is running.
2. Database schema **`pos`** exists
3. Required tables exist: `suppliers`, `products`, `category`
4. The `products` table has the real columns required for integration: PK: `products.id`
   - Supplier FK column: `products.suppliers_id`
   - Category FK column: `products.category_id`
5. At least **one category row exists**, because product insertion requires a valid category reference.
6. Application database credentials are valid (same as used by DALManager).

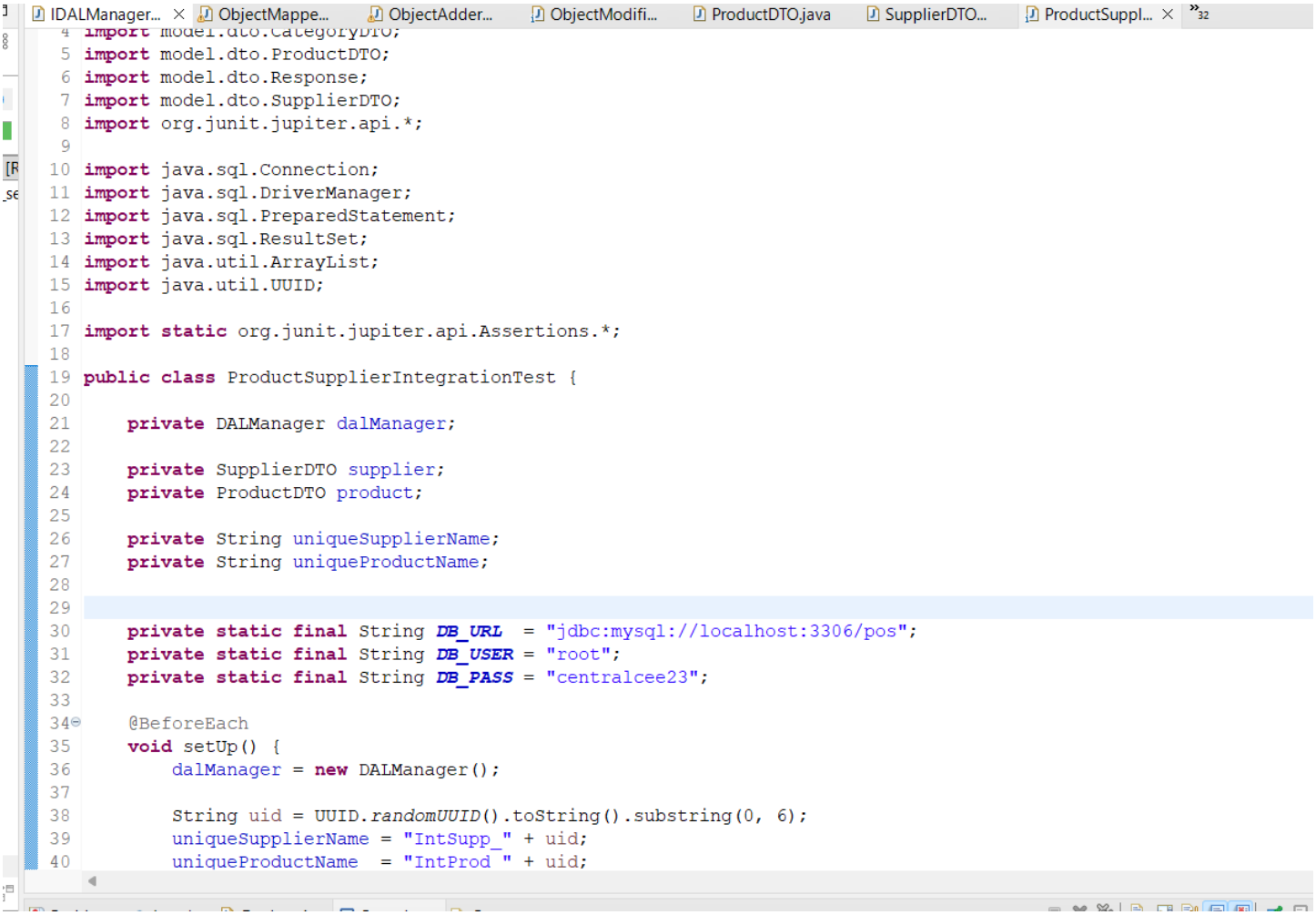
**Flow:**
Insert Supplier → Sync Supplier ID → Insert Product linked to Supplier → Search Product → Sync Product ID → Verify DB FK (`suppliers_id`) → Cleanup Product → Cleanup Supplier

Test Step Table

| Test Step | Action | Expected Outcome |
|---|---|---|
| 1 | Initialize DALManager | DAL is ready |
| 2 | Insert Supplier via DAL | Supplier inserted successfully |
| 3 | Sync Supplier ID | Supplier ID becomes > 0 |
| 4 | Retrieve Category ID | categoryId > 0 available |
| 5 | Insert Product with supplierId + categoryId | Product inserted successfully |
| 6 | Search product by substring | Inserted product found in results |
| 7 | Query DB FK (`suppliers_id`) by product id | FK value matches supplier id |
| 8 | Cleanup (delete product → delete supplier) | Database restored, no leftovers |

Test Flow Table

| Test Flow | Action | Expected Behavior | Actual Outcome |
|---|---|---|---|
| Positive Flow | Insert Supplier → Insert Product → Search Product → Verify FK | Product exists and FK correctly links to supplier | **Passed** |
| Cleanup Flow | Delete Product → Delete Supplier | No test rows remain in DB | **Passed** |


IDALManager...
ObjectMappe...
ObjectAdder...
ObjectModifi...
ProductDTO.java
SupplierDTO...
ProductSuppl...
import model.dto.CategoryDTO;
import model.dto.ProductDTO;
import model.dto.Response;
import model.dto.SupplierDTO;
import org.junit.jupiter.api.*;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.UUID;
import static org.junit.jupiter.api.Assertions.*;
public class ProductSupplierIntegrationTest {
private DALManager dalManager;
private SupplierDTO supplier;
private ProductDTO product;
private String uniqueSupplierName;
private String uniqueProductName;
private static final String DB_URL = "jdbc:mysql://localhost:3306/pos";
private static final String DB_USER = "root";
private static final String DB_PASS = "centralcee23";
@BeforeEach
void setUp() {
dalManager = new DALManager();
String uid = UUID.randomUUID().toString().substring(0, 6);
uniqueSupplierName = "IntSupp_" + uid;
uniqueProductName = "IntProd " + uid;

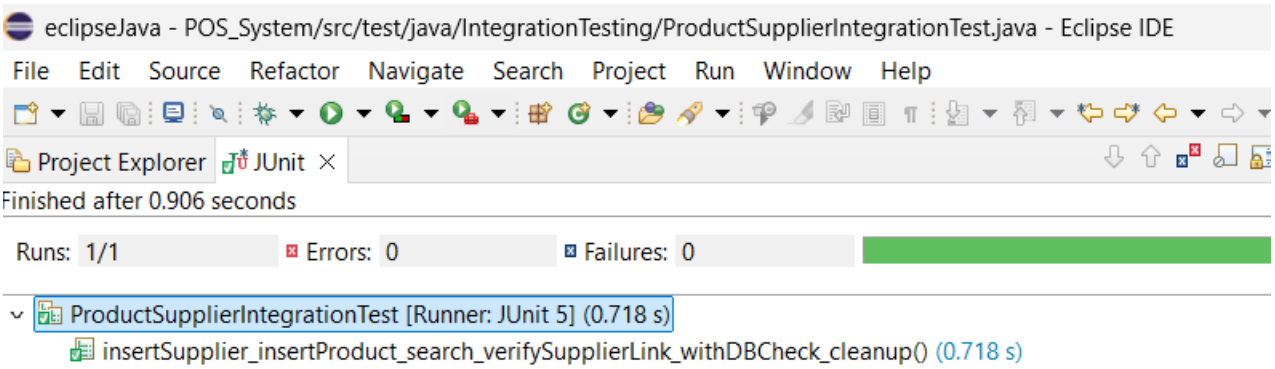
```
        // 1) INSERT SUPPLIER (DAL)
        supplier = new SupplierDTO();
        supplier.setName(uniqueSupplierName);
        supplier.setPhoneNumber("0691111111");

        Response res = new Response();
        dalManager.saveSupplier(supplier, res);
        assertTrue(res.isSuccessfull(), "Supplier insert failed: " + res.getErrorMessages());

        // 2) SYNC SUPPLIER ID (DAL)
        syncSupplierIdFromDb();
        assertTrue(supplier.getId() > 0, "Supplier ID should be > 0 after sync");
    }

    @AfterEach
    void tearDown() {
        // IMPORTANT: delete product first (product references suppliers_id)
        if (product != null && product.getProductId() > 0) {
            Response res = new Response();
            dalManager.deleteProduct(product, res);
            assertTrue(res.isSuccessfull(), "Product delete failed: " + res.getErrorMessages());
        }

        if (supplier != null && supplier.getId() > 0) {
            Response res = new Response();
            dalManager.deleteSupplier(supplier, res);
            assertTrue(res.isSuccessfull(), "Supplier delete failed: " + res.getErrorMessages());
        }
    }

    @Test
    void insertSupplier_insertProduct_search_verifySupplierLink_withDBCheck_cleanup() throws Exception {

        // 3) GET VALID CATEGORY ID (DAL)
        int categoryId = getAnyCategoryId();
        assertTrue(categoryId > 0, "categoryId must be > 0");
```

```
        // 4) INSERT PRODUCT LINKED TO SUPPLIER (DAL)
        product = new ProductDTO();
        product.setProductName(uniqueProductName);
        product.setBarcode("BC" + UUID.randomUUID().toString().substring(0, 8));
        product.setPrice(99.99);
        product.setStockQuantity(10);
        product.setCategoryId(categoryId);

        // DTO uses supplierId, DB column is suppliers_id (your DAL should map it)
        product.setSupplierId(supplier.getId());

        // must match your DB constraints; adjust if your system expects other values
        product.setQuantityType("counted");

        Response resInsert = new Response();
        dalManager.addProduct(product, resInsert);
        assertTrue(resInsert.isSuccessfull(), "Product insert failed: " + resInsert.getErrorMessages());

        // 5) SEARCH PRODUCT (DAL)
        Response resSearch = new Response();
        ArrayList<ProductDTO> results = dalManager.searchProductsByName("IntProd_", resSearch);

        assertTrue(resSearch.isSuccessfull(), "Product search failed: " + resSearch.getErrorMessages());
        assertNotNull(results, "Search results should not be null");
        assertFalse(results.isEmpty(), "Search results should not be empty");

        ProductDTO found = results.stream()
                .filter(p -> uniqueProductName.equals(p.getProductName()))
                .findFirst()
                .orElse(null);

        assertNotNull(found, "Inserted product not found in search results");

        // 6) SYNC PRODUCT ID FOR CLEANUP
        product.setProductId(found.getProductId());
        assertTrue(product.getProductId() > 0, "Product ID should be > 0 after sync");
```

```
        // 7) VERIFY SUPPLIER LINK USING REAL DB COLUMNS (source of truth)
        int supplierIdInDb = fetchSupplierIdFromDbByProductId(product.getProductId());
        assertEquals(supplier.getId(), supplierIdInDb,
                "DB suppliers_id should match inserted supplier id");
    }

    // -------------------- Helpers --------------------

    private void syncSupplierIdFromDb() {
        Response res = new Response();
        ArrayList<SupplierDTO> suppliers = dalManager.getSuppliers(res);

        assertTrue(res.isSuccessfull(), "Get suppliers failed: " + res.getErrorMessages());
        assertNotNull(suppliers, "Suppliers list is null");

        SupplierDTO found = suppliers.stream()
                .filter(s -> uniqueSupplierName.equals(s.getName()))
                .findFirst()
                .orElse(null);

        assertNotNull(found, "Could not sync supplier ID from DB");
        supplier.setId(found.getId());
    }

    private int getAnyCategoryId() {
        Response res = new Response();
        ArrayList<CategoryDTO> categories = dalManager.getCategories(res);

        assertTrue(res.isSuccessfull(), "Get categories failed: " + res.getErrorMessages());
        assertNotNull(categories, "Categories list is null");
        assertFalse(categories.isEmpty(), "No categories in DB. Insert at least one category first.");
```

```
        try {
            return (int) categories.get(0).getClass().getMethod("getId").invoke(categories.get(0));
        } catch (Exception e) {
            throw new RuntimeException("CategoryDTO does not have getId(). Adjust getAnyCategoryId().", e);
        }
    }

    /**
     * Uses your REAL schema from DESCRIBE products:
     * - PK column: id
     * - supplier FK column: suppliers_id
     */
    private int fetchSupplierIdFromDbByProductId(int productId) throws Exception {
        try (Connection conn = DriverManager.getConnection(DB_URL, DB_USER, DB_PASS)) {

            String sql = "SELECT suppliers_id FROM products WHERE id = ?";

            try (PreparedStatement ps = conn.prepareStatement(sql)) {
                ps.setInt(1, productId);

                try (ResultSet rs = ps.executeQuery()) {
                    assertTrue(rs.next(), "No product row found for id=" + productId);
                    return rs.getInt("suppliers_id");
                }
            }
```

JUnit Result:




Runs: 1, Errors: 0, Failures: 0

Conclusion

The Product–Supplier integration test executed successfully and confirmed that:

- Suppliers can be inserted and removed correctly through the DAL.
- Products can be inserted with valid foreign key references (supplier + category).
- Search by substring returns expected products.
- The supplier relationship is correctly persisted in the database (`products.suppliers_id` matches the inserted supplier ID).
- Cleanup was performed safely in the correct order (child then parent), leaving the database unchanged after the test.

Overall, this test provides strong confidence that the POS inventory subsystem correctly integrates supplier management, product persistence, and database referential integrity under a Bottom-Up integration strategy.

## IntegrationTesting8: Employee CRUD Integration Test

Test Level: Integration Testing
Integration Strategy: **Big Bang Integration (Employee Subsystem)**
Purpose of the Test
The purpose of this integration test is to verify the correct interaction between the **Employee Data Transfer Object (EmployeeDTO)**, the **Data Access Layer (DALManager)**, and the **MySQL database** during a complete employee lifecycle.
This test ensures that employee-related operations function correctly when executed sequentially and dependently, validating real data persistence and retrieval without using mocks or stubs.

Scope of the Test: This test covers the following employee-related functionalities:

- Inserting a new employee record
- Updating an existing employee record
- Retrieving employee records from the database
- Verifying data consistency after updates
- Deleting the employee record as cleanup

The test validates the integration of:

- `DALManager`
- `ObjectAdder`
- `ObjectModifier`
- `DBReader`
- `ObjectRemover`
- `ObjectMapper`
- `MySQLConnection`
- Database table: `employees`

Database Schema Used: table employees

| Field | Type | Description |
|---|---|---|
| id | int | Primary key, auto-increment |
| name | varchar(50) | Employee name (NOT NULL) |
| phoneNumber | varchar(10) | Employee phone number (NOT NULL) |

Preconditions
The MySQL database server is running.
The `pos` database exists.
The `employees` table exists with the correct schema.
Database credentials configured in `MySQLConnection` are valid.
No employee-related mocks or stubs are used.

Test Data
To avoid conflicts with existing records, the test uses dynamically generated unique data.

| Attribute | Example Value |
|---|---|
| name | `IntEmp_5a9c3e` |
| phoneNumber | `0670000000` |
| updatedName | `IntEmp_5a9c3e_UPD` |
| updatedPhone | `0671111111` |

Test scenario description

This test follows a **stateful integration flow**, where each step depends on the success of the previous one.

**Flow:**

Insert → Update → Read → Verify → Delete

Test Steps and Expected Results

Step 1: Insert Employee

A new EmployeeDTO object is created and persisted using DALManager.saveEmployee().

**Expected Result:**

The employee is successfully inserted into the database.

The response object indicates a successful operation.

Step 2: Retrieve Generated Employee ID

**Action:** The test retrieves all employees using DALManager.getEmployees() and identifies the inserted employee by its unique name to obtain the auto-generated ID.

**Expected Result:**

The inserted employee is found.

A valid employee ID greater than zero is retrieved.

Step 3: Update Employee

**Action:** The employee's name and phone number are modified and updated using DALManager.updateEmployee().

**Expected Result:**

The update operation completes successfully.

The database reflects the updated employee information.

Step 4: Retrieve Employees

**Action:** The test retrieves all employees again using DALManager.getEmployees().

**Expected Result:** The updated employee record exists in the returned list.

Step 5: Verify Updated Data

**Action:** The employee is located using its ID and the updated values are compared with the expected ones.

**Expected Result:**

The employee name matches the updated name.

The phone number matches the updated phone number.

Old values are no longer present.

Step 6: Cleanup – Delete Employee

**Action:** The employee record is deleted using DALManager.deleteEmployee() in the cleanup phase.

**Expected Result:**

The employee record is successfully removed & The database state is restored after the test execution.

Postconditions

No test data remains in the database.

The system returns to its original state.

The test does not affect other subsystems.

```java
import dal.DALManager;
import model.dto.EmployeeDTO;
import model.dto.Response;
import org.junit.jupiter.api.*;

import java.util.ArrayList;
import java.util.UUID;

import static org.junit.jupiter.api.Assertions.*;

public class EmployeeCRUDIntegrationTest {

    private DALManager dalManager;

    private EmployeeDTO employee;     // used for update + delete
    private int employeeId;           // synced from DB list

    private String uniqueName;
    private String initialPhone;
    private String updatedPhone;

    @BeforeEach
    void setUp() {
        dalManager = new DALManager();

        String uid = UUID.randomUUID().toString().substring(0, 6);
        uniqueName = "IntEmp_" + uid;
        initialPhone = "0670000000";  // 10 digits
        updatedPhone = "0671111111";  // 10 digits

        // 1) INSERT (DAL)
        employee = new EmployeeDTO();
        employee.setName(uniqueName);
        employee.setPhoneNumber(initialPhone);

        Response resInsert = new Response();
        dalManager.saveEmployee(employee, resInsert);
        assertTrue(resInsert.isSuccessfull(), "Employee insert failed: " + resInsert.getErrorMessages());

        // 2) SYNC ID (DAL read)
        employeeId = syncEmployeeIdByName(uniqueName);
        assertTrue(employeeId > 0, "Synced employee id must be > 0");

        // Set ID so update/delete target the correct row
        employee.setId(employeeId);
    }

    @AfterEach
    void tearDown() {
        if (employee != null && employee.getId() > 0) {
            Response resDelete = new Response();
            dalManager.deleteEmployee(employee, resDelete);
            assertTrue(resDelete.isSuccessfull(), "Employee delete failed: " + resDelete.getErrorMessages());
        }
    }

    @Test
    void insert_update_read_verify_delete_employee_complex_flow() {

        // 3) UPDATE (DAL)
        String updatedName = uniqueName + "_UPD";
        employee.setName(updatedName);
        employee.setPhoneNumber(updatedPhone);

        Response resUpdate = new Response();
        dalManager.updateEmployee(employee, resUpdate);
        assertTrue(resUpdate.isSuccessfull(), "Employee update failed: " + resUpdate.getErrorMessages());

        // 4) READ (DAL)
        Response resRead = new Response();
        ArrayList<EmployeeDTO> employees = dalManager.getEmployees(resRead);

        assertTrue(resRead.isSuccessfull(), "Get employees failed: " + resRead.getErrorMessages());
        assertNotNull(employees);
        assertFalse(employees.isEmpty(), "Employees list should not be empty");

        // 5) VERIFY (find by id)
        EmployeeDTO found = employees.stream()
                .filter(e -> e.getId() == employeeId)
                .findFirst()
                .orElse(null);

        assertNotNull(found, "Updated employee not found in getEmployees()");

        assertEquals(updatedName, found.getName(), "Employee name was not updated correctly");
        assertEquals(updatedPhone, found.getPhoneNumber(), "Employee phoneNumber was not updated correctly");
```

```
89
90          // verify old name is not present for this id
91          assertNotEquals(uniqueName, found.getName(), "Employee name should not still be old value");
92      }
93
94      private int syncEmployeeIdByName(String name) {
95          Response res = new Response();
96          ArrayList<EmployeeDTO> employees = dalManager.getEmployees(res);
97
98          assertTrue(res.isSuccessfull(), "Get employees failed while syncing ID: " + res.getErrorMessages());
99          assertNotNull(employees);
100
101         EmployeeDTO found = employees.stream()
102                 .filter(e -> name.equals(e.getName()))
103                 .findFirst()
104                 .orElse(null);
105
106         assertNotNull(found, "Could not sync inserted employee ID from DB (employee not found by name)");
107         return found.getId();
108     }
109 }
```

Pass / Fail Criteria

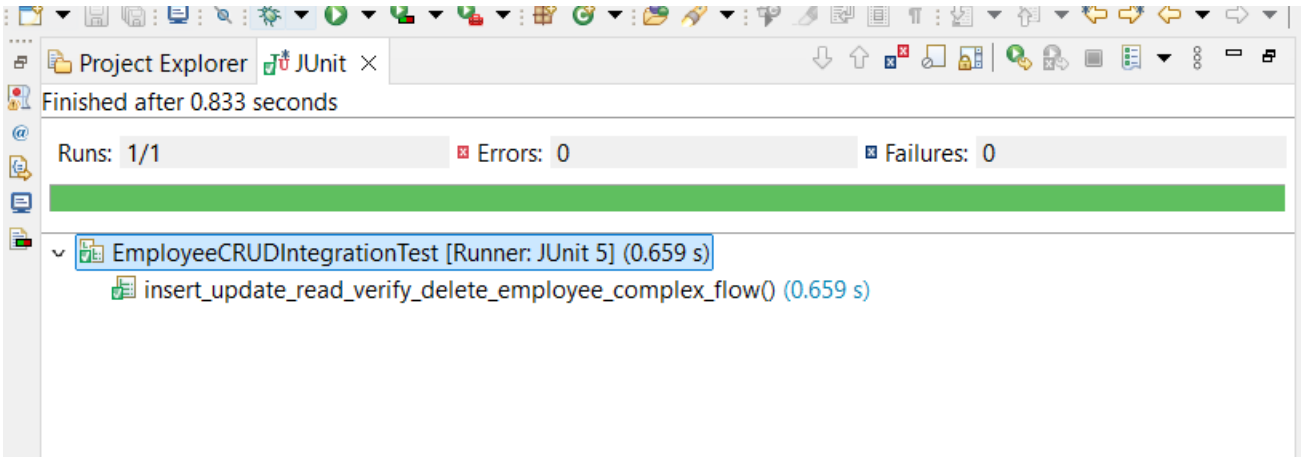
**Pass:**

All database operations complete successfully.
Updated data is retrieved correctly.
Cleanup deletion is successful.

**Fail:**

Any DAL operation fails.
Data inconsistency is detected.
Cleanup operation fails.

**JUnit Result:** PASSED ; **Runs:** 1, **Errors:** 0, **Failures:** 0




Conclusion

The Employee CRUD Integration Test confirms that the employee subsystem operates correctly when performing multiple dependent operations through the real Data Access Layer and MySQL database. The successful execution demonstrates reliable integration, correct data persistence, and safe cleanup behavior.