DAO testing tutorial

In this tutorial, you'll write integration tests for the Pizza Shop `JdbcSaleDao`.

The tutorial includes a small console application, the `JdbcSaleDao`, and a scaled-down version of the Pizza Shop database, `PizzaShopLite`. You don't need to make any modifications or perform any additional steps such as setting up the database to run the application.

> Note: typically you only use a scaled-down database for your tests, but this tutorial uses the database for the console application too so you can explore the data and help you write tests.

The only changes you'll make are to the `JdbcSaleDaoTests.java` test class file where you'll implement JUnit tests for the CRUD methods in `JdbcSaleDao`.

Tutorial application walkthrough, DAOs in action

As just mentioned, the tutorial includes a small console application. The application allows you to see the DAO in action, not only at run-time, but you can also examine the code in `TutorialController.java` to see how they're called and the results handled.

In addition, the application displays information that's particularly useful for writing tests. For instance, there is a set of breadcrumbs displayed throughout the run of the application which provide values for the currently selected customer and sale:

```
customerId:FullName >> saleId:total:delivery
```

So, if you're interested in testing `getSalesByCustomerId()` for a particular customer, you merely need to select the customer in the application to find their id in the breadcrumbs. There's no need to root around in the database or set breakpoints and examine values.

Start the tutorial by running `Tutorial.main()` and following along with the walkthrough.

> Note: the application refreshes the `PizzaShopLite` database on startup so you might notice a few seconds delay before the `Main Menu` displays:

```
Welcome to the Pizza Shop.

------------------------------
Main Menu
Selected: --- >> ---
------------------------------
1: Customer - select
2: Sale - create, read, update, delete
3: Exit the program
Please select:
```

Although the Pizza Shop database supports customer-less sales, the application requires that every sale has a customer. With that in mind, select `1: Customer - select` from the `Main Menu`. This displays all the customers in the `PizzaShopLite` database:

```
Customer List
-----------------------------
1: Besnardeau, Robin
2: Dobbins, Dud
3: Lampaert, Madge
4: Mallon, Deanne
5: Mamwell, Elenore
6: Woofenden, Row
Please select:
```

Select `3: Lampaert, Madge` from the `Customer List`.

Madge becomes your *selected* customer, and the application returns to the `Main Menu`:

```
------------------------------------------
Main Menu
Selected: 3:Lampaert, Madge >> ---
------------------------------------------
1: Customer - select
2: Sale - create, read, update, delete
3: Exit the program
Please select:
```

> Note: Madge's id and full name now appears in the `Selected:` breadcrumbs under the `Main Menu`.

Next, select `2: Sale - create, read, update, delete` from the `Main Menu` to go to the `Sale Menu` and work with Madge's sales:

```
------------------------------------------
Sale Menu
Selected: 3:Lampaert, Madge >> ---
------------------------------------------
1: Select sale
2: Create new sale
3: Update selected sale
4: Delete selected sale
5: Return to Main Menu
Please select:
```

From the `Sale Menu` select `1: Select sale` to display Madge's existing sales:

```
Sale List
-----------------------------
1: Sale{saleId=5, total=23.98, delivery=true, customerId=3}
2: Sale{saleId=6, total=41.25, delivery=false, customerId=3}
Please select:
```

> Note: the list displays the attribute names and values for each of the `Sale` objects. Once again, this information is particularly useful when you write your tests.

For now, skip selecting a specific sale, press `enter` to return to the `Sale Menu`:

```

------------------------------------------
Sale Menu
Selected: 3:Lampaert, Madge >> ---
------------------------------------------
1: Select sale
2: Create new sale
3: Update selected sale
4: Delete selected sale
5: Return to Main Menu
Please select:
```

Select `2: Create new sale` to add a new sale for Madge:

```
Enter new sale information
-----------------------------
Total: 34.56
Delivery (Y/N)?: Y
-----------------------------

Are you sure you wish to create the new sale (Y/N)?: Y

Sale{saleId=12, total=34.56, delivery=true, customerId=3} CREATED !!!
```

Enter values at the `Total:` and `Delivery:` prompts and `Y` to create when asked.

> Note: the application doesn't perform any validation. *Please be careful when entering values.*

Provided there are no problems, a confirmation message appears, and the application returns to the `Sale Menu`:

```

-------------------------------------------------
Sale Menu
Selected: 3:Lampaert, Madge >> 12:$34.56:true
-------------------------------------------------
1: Select sale
```

```
2: Create new sale
3: Update selected sale
4: Delete selected sale
5: Return to Main Menu
Please select:
```

> Note: the new sale is *auto-magically* your *selected* sale, and appears in the `Selected:` breadcrumbs under the `Sale Menu`.

If you'd like, you can re-select `1: Select sale` from the `Sale Menu` to display the new sale along with the previously existing sales:

```
Sale List
-----------------------------
1: Sale{saleId=5, total=23.98, delivery=true, customerId=3}
2: Sale{saleId=6, total=41.25, delivery=false, customerId=3}
3: Sale{saleId=12, total=34.56, delivery=true, customerId=3}
Please select:
```

Please make sure to re-select `3: Sale(saleId=12, total=34.56, delivery=true, customerId=3)` to keep the new sale as the *selected* sale and return back to the `Sale Menu`:

```
-------------------------------------------------
Sale Menu
Selected: 3:Lampaert, Madge >> 12:$34.56:true
-------------------------------------------------
1: Select sale
2: Create new sale
3: Update selected sale
4: Delete selected sale
5: Return to Main Menu
Please select:
```

Select `3: Update selected sale` to make corrections to the new sale:

```
Current sale information
-----------------------------
Sale Id: 12
Total: 34.56
Delivery: true
Customer: 3
-----------------------------

Update sale information
-----------------------------
Total: 65.43
Delivery (Y/N)?: N
```

```
-----------------------------

Updated campground information
Current sale information
-----------------------------
Sale Id: 12
Total: 65.43
Delivery: false
Customer: 3
-----------------------------

Are you sure you wish to update the sale (Y/N)?: Y

Sale{saleId=12, total=65.43, delivery=false, customerId=3} UPDATED !!!
```

The application displays the existing values and prompts you for the values you can change. Enter only the values you want to change. Leave the prompt blank you if want to accept the existing value.

Enter `Y` to update when asked. The application displays a confirmation message when the update is successful and returns you to the `Sale Menu`:

```

--------------------------------------------------
Sale Menu
Selected: 3:Lampaert, Madge >> 12:$65.43:false
--------------------------------------------------
1: Select sale
2: Create new sale
3: Update selected sale
4: Delete selected sale
5: Return to Main Menu
Please select:
```

The updated sale remains the *selected* sale. Select `4: Delete selected sale` to remove it:

```

Current sale information
-----------------------------
Sale Id: 12
Total: 65.43
Delivery: false
Customer: 3
-----------------------------

Are you sure you wish to delete the sale (Y/N)?: Y

Sale{saleId=12, total=65.43, delivery=false, customerId=3} DELETED !!!
```

The application displays *selected* sale information and asks you to confirm the deletion. Enter `Y` to confirm.

Now that you have seen all the CRUD methods in action, enter `5: Return to Main Menu` on the `Sale Menu` to return to `Main Menu` where you can select another customer and experiment on your own.

> Reminder, you can see the actual handling of the `SaleDao` in the application code by examining `TutorialController.java`. Search for `// DAO in action !!!` to find the calls to and handling of the various `JdbcSaleDao` methods.

Review mock database connection

DAO tests rely upon a connection to a *mock* database. The mock, or testing, database is an entirely separate database from the application database, although the two usually share the same structure in terms of tables and constraints. The difference, if any, between the two is usually in the amount of data each has. The application database typically contains a rich set of records, while the testing database has a limited number of records defining a handful of test cases.

The classes responsible for setting up the database and connection for the integration tests use a framework called Spring and a concept called Dependency Injection. Both of these are topics beyond the scope of this unit, but you'll learn about them at a later time. Right now, you just need to understand the links between code pointed out to you here.

The `BaseDaoTests.java` and `TestingDatabaseConfig.java` classes share the connection code:

```java
// BaseDaoTests.java
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = TestingDatabaseConfig.class)
public abstract class BaseDaoTests {

    @Autowired
    protected DataSource dataSource;

    // ...

}

// TestingDatabaseConfig.java
@Configuration
public class TestingDatabaseConfig {

    // ...

    @Bean
    public DataSource dataSource() throws SQLException {

        // ...

    }

    // ...

}
```

The `@ContextConfiguration(classes = TestingDatabaseConfig.class)` annotation at the top of `BaseDaoTests` ties the two classes together.

The `@Bean` preceding the `dataSource()` method in `TestingDatabaseConfig` links it to the `@Autowired dataSource` in `BaseDaoTests` by Spring's dependency injection.

> Note: Spring beans are singletons by default. In other words, that means that they're never instantiated more than once in any given run of DAO tests. The code inside the `TestingDatabaseConfig.dataSource()` method is never called more that once.

In the `dataSource()` method, you can see the code that creates the test database. The `adminDataSource` connects to the `postgres` database, allowing it to safely drop and create the `PizzaShopLiteTesting` database:

```
@Bean
public DataSource dataSource() throws SQLException {

    // Drop and then recreate the testing database under separate "admin"
connection
    SingleConnectionDataSource adminDataSource = new SingleConnectionDataSource();
    adminDataSource.setUrl("jdbc:postgresql://localhost:5432/postgres");
    adminDataSource.setUsername("postgres");
    adminDataSource.setPassword("postgres1");
    JdbcTemplate adminJdbcTemplate = new JdbcTemplate(adminDataSource);
    adminJdbcTemplate.update("DROP DATABASE IF EXISTS \"PizzaShopLiteTesting\";");
    adminJdbcTemplate.update("CREATE DATABASE \"PizzaShopLiteTesting\";");

    // ...
}
```

After dropping and creating the new instance of the testing database, the `dataSource()` method sets up the connection and runs the `PizzaShopLite.sql` SQL script to load the test data for the database:

```
@Bean
public DataSource dataSource() throws SQLException {

    // ...
    adminJdbcTemplate.update("CREATE DATABASE \"PizzaShopLiteTesting\";");

    // Setup up the testing connection
    SingleConnectionDataSource dataSource = new SingleConnectionDataSource();
    dataSource.setUrl("jdbc:postgresql://localhost:5432/PizzaShopLiteTesting");
    dataSource.setUsername("postgres");
    dataSource.setPassword("postgres1");
    dataSource.setAutoCommit(false); // So we can rollback after each test.

    // Refresh the testing database by running the setup script
    ScriptUtils.executeSqlScript(dataSource.getConnection(), new
ClassPathResource("PizzaShopLite.sql"));
```

```
    return dataSource;
}
```

Review DAO tests class

You'll add your DAO tests to `JdbcSaleDaoTests`, but first take a look at the provided code:

```
public class JdbcSaleDaoTests extends BaseDaoTests {

    private JdbcSaleDao jdbcSaleDao;

    @Before
    public void setup() {

        // Arrange - new instance of JdbcSaleDao before each and every test
        jdbcSaleDao = new JdbcSaleDao(dataSource);
    }

    // Five empty tests ...

    // Convenience method in lieu of a Sale constructor with all the fields as
parameters.
    // Similar to mapRowToSale() in JdbcSaleDao.
    private static Sale mapValuesToSale(int saleId, BigDecimal total, boolean
delivery, Integer customerId) {

        Sale sale = new Sale();
        sale.setSaleId(saleId);
        sale.setTotal(total);
        sale.setDelivery(delivery);
        sale.setCustomerId(customerId);
        return sale;
    }

    private void assertSalesMatch(String message, Sale expected, Sale actual) {

        Assert.assertEquals(message, expected.getSaleId(), actual.getSaleId());
        Assert.assertEquals(message, expected.getTotal(), actual.getTotal());
        Assert.assertEquals(message, expected.isDelivery(), actual.isDelivery());
        Assert.assertEquals(message, expected.getCustomerId(),
actual.getCustomerId());
    }
}
```

The `JdbcSaleDaoTests` class extends `BaseDaoTests` so it can use its `dataSource`. This `dataSource` initializes the `jdbcSaleDao` in the `setup()` method every time a test runs.

There are also two helper methods provided, `mapValuesToSale()` and `assertSalesMatch()` to create new instances of `Sale` objects and to assert two `Sale` objects are equal.

In between the `setup()` and two helper methods are five failing test methods. You'll replace each `Assert.fail("Test not implemented.")` in the next five steps.

Step One: Add the `getSale_returns_correct_sale_for_id` test

Test `getSale()` by asking for Madge Lampaert's first sale and asserting the returned sale isn't null and matches the expected values.

First, create constants for Madge's customer id and the first sale id to use where needed:

```java
// Step One: Add constants for Madge
private static final int MADGE_CUSTOMER_ID = 3;
private static final int MADGE_FIRST_SALE_ID = 5;
```

Then, fill out the test method:

```java
@Test
public void getSale_returns_correct_sale_for_id() {

    // Step One: Replace Assert.fail("Test not implemented.")

    // Arrange - Create an instance of Madge's first Sale object.
    Sale madgeFirstSale = mapValuesToSale(MADGE_FIRST_SALE_ID, new
BigDecimal("23.98"), true, MADGE_CUSTOMER_ID);

    // Act - retrieve Madge's first sale
    Sale sale = jdbcSaleDao.getSale(MADGE_FIRST_SALE_ID);

    // Assert - retrieved sale is not null and matches expected sale
    Assert.assertNotNull("getSale(" + MADGE_FIRST_SALE_ID + ") returned null",
sale);
    assertSalesMatch("getSale(" + MADGE_FIRST_SALE_ID + ") returned wrong or
partial data", madgeFirstSale, sale);
}
```

As the comments in the preceding code describe, first you create an instance of Madge's first sale using the constants and other known values of that sale. Next, you call the `getSale()` method using the constant for the first sale. Lastly, you check that the retrieved sale isn't null, and that it's equal to the expected value of Madge's first sale.

Step Two: Add the `getSalesByCustomerId_returns_sales_for_customer_id()` test

Test `getSalesByCustomerId()` by asking for sales for Madge, a customer who has no sales, and a non-existent customer. With each customer's sales, asserting the size of the list returned. Madge has two known sales, while the customer without sales, and the unknown customer both return empty lists.

Add constants for the ids of the other two customers:

```
// Step Two: Add constants for customer without sale and non-existent customer
private static final int CUSTOMER_WITHOUT_SALES_ID = 5;
private static final int NON_EXISTENT_CUSTOMER_ID = 7;
```

Then, fill out the test method:

```
@Test
public void getSalesByCustomerId_returns_sales_for_customer_id() {

    // Step Two: Replace Assert.fail("Test not implemented.")

    // Act - retrieve sales for Madge
    List<Sale> sales = jdbcSaleDao.getSalesByCustomerId(MADGE_CUSTOMER_ID);
    // Assert - Madge has two existing sales
    Assert.assertEquals("getSalesByCustomerId(" + MADGE_CUSTOMER_ID + ") returned
wrong number of sales",
            2, sales.size());

    // Act - retrieve customer with no sales
    sales = jdbcSaleDao.getSalesByCustomerId(CUSTOMER_WITHOUT_SALES_ID);
    // Assert - list of sales is empty for customer with no sales
    Assert.assertEquals("getSalesByCustomerId(" + CUSTOMER_WITHOUT_SALES_ID +
            ") without sales returned wrong number of sales", 0, sales.size());

    // Act - retrieve customer that doesn't exist
    sales = jdbcSaleDao.getSalesByCustomerId(NON_EXISTENT_CUSTOMER_ID);
    // Assert - list of sales is empty for customer that doesn't exist
    Assert.assertEquals("Customer doesn't exist, getSalesByCustomerId(" +
NON_EXISTENT_CUSTOMER_ID +
            ") returned the wrong number of sales", 0, sales.size());
}
```

Like in the first problem, the comments describe what each statement in the test is doing. Each pair of statements retrieves the sales for one of the three customers and asserts that the list is the correct size.

Step Three: Add the `createSale_returns_sale_with_id_and_expected_values()` test

Test `createSale()` by creating a `Sale` object for Madge and asserting the `saleId` for the returned `Sale` has a value set, and the `total`, `delivery`, and `customerId` are the expected values:

```
@Test
public void createSale_returns_sale_with_id_and_expected_values() {

    // Step Three: Replace Assert.fail("Test not implemented.")

    // Arrange - instantiate a new Sale object for Madge
    Sale newSale = new Sale();
    newSale.setTotal(new BigDecimal("34.56"));
    newSale.setDelivery(true);
    newSale.setCustomerId(MADGE_CUSTOMER_ID);

    // Act - create sale from instantiated Sale object
    Sale createdSale = jdbcSaleDao.createSale(newSale);

    // Assert - created sale is correct
    Assert.assertNotEquals("saleId not set when created, remained 0", 0,
createdSale.getSaleId());
    Assert.assertEquals(newSale.getTotal(), createdSale.getTotal());
    Assert.assertEquals(newSale.isDelivery(), createdSale.isDelivery());
    Assert.assertEquals(newSale.getCustomerId(), createdSale.getCustomerId());
}
```

Step Four: Add the `updateSale_has_expected_values_when_retrieved()` test

Test `updateSale()` by retrieving Madge's first sale, modifying the `total` and `delivery` properties, update the modified sale object, and then retrieving the sale and asserting the updated sale has the expected modifications:

```
@Test
public void updateSale_has_expected_values_when_retrieved() {

    // Step Four: Replace Assert.fail("Test not implemented.")

    // Arrange - retrieve Madge's first sale and change values
    Sale saleToUpdate = jdbcSaleDao.getSale(MADGE_FIRST_SALE_ID);
    saleToUpdate.setTotal(new BigDecimal("89.32"));
    saleToUpdate.setDelivery(false);

    // Act - update the existing sale with changed values and re-retrieve
    jdbcSaleDao.updateSale(saleToUpdate);
    Sale updatedSale = jdbcSaleDao.getSale(MADGE_FIRST_SALE_ID);

    // Assert - sale has been updated correctly
    assertSalesMatch("Updated Madge's first sale returned wrong or partial data",
saleToUpdate, updatedSale);
}
```

Step Five: Add the `deleted_sale_cant_be_retrieved()` test

Test `deleteSale()` by deleting Madge's first sale, and then attempting to retrieve it, asserting retrieved sale is null:

```java
@Test
public void deleted_sale_cant_be_retrieved() {

    // Step Five: Replace Assert.fail("Test not implemented.")

    // Act - delete existing first sale for Madge
    jdbcSaleDao.deleteSale(MADGE_FIRST_SALE_ID);

    // Assert - Madge's deleted sale can't be retrieved
    Sale sale = jdbcSaleDao.getSale(MADGE_FIRST_SALE_ID);
    Assert.assertNull(sale);
}
```

Next steps

Now that you've written an initial set of tests for `JdbcSaleDao`, you could enrich them. For instance, you could add a test of `getSale()` with the id of a non-existent sale. Or, testing each sale item returned from `getSalesByCustomerId()` rather than just checking the size of the list.

Another possibility would be to add tests for the two methods in `JdbcCustomerDao`. They would model the existing `JdbcSaleDaoTests` tests and added in a new `JdbcCustomerDaoTest` class. Don't forget to extend `BaseDaoTests`.

Finally, the create, update, and delete tests of the corresponding methods in `JdbcSaleDao` follow the happy path. Try testing deleting a non-existent sale, or updating a sale with an id for an unknown customer.