Integration Testing

What Is Integration Testing?

A good unit test isolates the code that it tests from dependencies on outside resources. This is desirable because it usually makes the test more reliable and also easier to debug when it fails.

However, interactions with outside resources are a potential source of bugs and bad assumptions and should definitely be tested. In fact, there are classes whose primary function is to interact with outside resources. Data Access Objects are an example of this.

In order to validate that a DAO is functioning correctly, we really need to test it against a database. This is an example of an "**integration test**".

What Is An Integration Test?

Integration Testing is a broad category of tests that validate the integration between units of code, or between code and outside dependencies such as databases or network resources.

Integration tests:

- Use the same tools as unit tests (i.e. JUnit)
- Usually slower than unit tests (but often still measured in ms)
- More complex to write and debug
- Can have dependencies on outside resources like files or a database.

DAO Integration Testing

Since DAOs exist solely for the purpose of interacting with a database, they are often best tested using an integration test.

Integration tests with a database ensure DAO code functions correctly:

- **SELECT** statements are **tested by inserting dummy data** before the test
- **INSERT** statements are **tested by searching for the data**
- **UPDATE** statements are **tested by verifying dummy data changed** or that rowsAffected == expectedResult
- **DELETE** statements are tested by seeing if **dummy data is missing** or that rowsAffected == expectedResult

DAO Integration Testing

Tests (including integration tests) should be:

- **Repeatable:**
  - If the test passes/fails on first execution, it should pass/fail on second execution if no code has changed.
- **Independent:**
  - A test should be able to be run on it's own, independently of other tests, OR together with other tests and have the same result either way.
- **Obvious:**
  - When a test fails, it should be as obvious as possible why it failed.

Approaches for Managing Test Data

Remotely Hosted Shared Test Database

An RDBMS is installed on a remote server and shared by all developers on the team for testing.

- **Advantages**:
  - Easy setup, often already exists
  - Production-like software and (possibly) hardware
- **Disadvantages**:
  - Unreliable and brittle
  - Lack of test isolation
  - Temptation to rely on existing data (which can change)

Approaches for Managing Test Data

Locally Hosted Test Database

An RDBMS is installed and hosted locally on each developer's machine. (This is the approach we will use.)

- **Advantages**:
  - Production-like software
  - Reliable (local control)
  - Isolation
- **Disadvantages**:
  - Requires local hardware resources
  - RDBMS needs to be installed and managed

Approaches for Managing Test Data

Embedded, In-memory Database

An in-memory, embedded database server is started and managed by test code while running integration tests.

- **Advantages**:
  - Very reliable
  - Consistent across development machines (managed by source control)
  - Lightweight
- **Disadvantages**:
  - Not the same software used in production
  - Cannot use proprietary features of production RDBMS

Let's Look At Some DAO Integration Tests

Writing DAO Integration Tests

You want to have the data in the database in a specific known state before you run your tests. This allows your tests to be **repeatable --** they will return the same results every time you run them.

A test country is loaded into the database before and after **every** test. This happens in the **`setup()`** method of **`JDBCCityDAOIntegrationTest`**. The **`cleanup()`** methods removes it and any other test data that was inserted or modified during the tests.

Using Transactions For DAO Integration Testing

After the integration **tests run**, we want to have the **database restored to its original state** so that **no data is permanently modified**.

This is accomplished by creating a **Transaction Scope** object. By creating a transaction before the test runs, and **rolling back the transaction after it's completed**, this ensures that there are no permanent changes to the database.

For our tests we will use **`SingleConnectionDataSource`** object rather a **`BasicDataSource`**. Using this particular implementation of **`DataSource`** so that every database interaction is part of the same database session and hence the same database transaction

Using Transactions For DAO Integration Testing

In order to allow us to rollback transaction, we **`setAutoCommit(false)`**. This prevents the database from automatically committing after each SQL statement executes. Since queries are not being auto-committed, we can roll them back when our tests complete to avoid permanently affecting data in the database.

```
@BeforeClass
public static void setupDataSource() {
     dataSource = new SingleConnectionDataSource();
     dataSource.setUrl("jdbc:postgresql://localhost:5432/world");
     dataSource.setUsername("postgres");
     dataSource.setPassword("postgres1");
     dataSource.setAutoCommit(false);
}
```

Using Transactions For DAO Integration Testing

In order to allow us to rollback transaction, we **`setAutoCommit(false)`**. This prevents the database from automatically committing after each SQL statement executes. Since queries are not being auto-committed, we can roll them back when our tests complete to avoid permanently affecting data in the database.

The **@BeforeClass** annotation denotes code to be called before the test class is created.

```
@BeforeClass
public static void setupDataSource() {
     dataSource = new SingleConnectionDataSource();
     dataSource.setUrl("jdbc:postgresql://localhost:5432/world");
     dataSource.setUsername("postgres");
     dataSource.setPassword("postgres1");
     dataSource.setAutoCommit(false);
}
```

Using Transactions For DAO Integration Testing

In order to allow us to rollback transaction, we `setAutoCommit(false)`. This prevents the database from automatically committing after each SQL statement executes. Since queries are not being auto-committed, we can roll them back when our tests complete to avoid permanently affecting data in the database.

The **@BeforeClass** annotation denotes code to be called before the test class is created.

We call setAutoCommit(false) here.

```java
@BeforeClass
public static void setupDataSource() {
     dataSource = new SingleConnectionDataSource();
     dataSource.setUrl("jdbc:postgresql://localhost:5432/world");
     dataSource.setUsername("postgres");
     dataSource.setPassword("postgres1");
     dataSource.setAutoCommit(false);
}
```

Using Transactions For DAO Integration Testing

After each test we use the **`SingleConnectionDataSource`** connection to roll back all changes.

```
@After
public void rollback() throws SQLException {
    dataSource.getConnection().rollback();
}
```

Using Transactions For DAO Integration Testing

After each test we use the **`SingleConnectionDataSource`** connection to roll back all changes.

The **`@After`** annotation denotes code to be called after each test completes.

```
@After
public void rollback() throws SQLException {
    dataSource.getConnection().rollback();
}
```

Using Transactions For DAO Integration Testing

After each test we use the **`SingleConnectionDataSource`** connection to roll back all changes.

The **`@After`** annotation denotes code to be called after each test completes.

We use the connection of the **`SingleConnectionDataSource`** to roll back any database changes caused by the test.

```
@After
public void rollback() throws SQLException {
    dataSource.getConnection().rollback();
}
```

Using Transactions For DAO Integration Testing

`SingleConnectionDataSource` is useful because it does not close connections automatically after it used. This allows us to roll back changes after each test but it also requires us to destroy the connection manually when we are done using the test class.

```
@AfterClass
public static void closeDataSource() throws
SQLException {
    dataSource.destroy();
}
```

Using Transactions For DAO Integration Testing

`SingleConnectionDataSource` is useful because it does not close connections automatically after it used. This allows us to roll back changes after each test but it also requires us to destroy the connection manually when we are done using the test class.

The **@AfterClass** annotation denotes code to be called before the test class is destroyed

```
@AfterClass
public static void closeDataSource() throws
SQLException {
    dataSource.destroy();
}
```

Using Transactions For DAO Integration Testing

`SingleConnectionDataSource` is useful because it does not close connections automatically after it used. This allows us to roll back changes after each test but it also requires us to destroy the connection manually when we are done using the test class.

The **@AfterClass** annotation denotes code to be called before the test class is destroyed

In the **@AfterClass** code we destroy the **`SingleConnectionDataSource`**

```
@AfterClass
public static void closeDataSource() throws
SQLException {
    dataSource.destroy();
}
```