

# DAO 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.

Test data is loaded into the database before the tests are run and removed once the tests are complete. Each test can set up its data in relation to the test data to achieve the desired test scenario. The `rollback()` method in the `BaseDaoTests` class removes it and any other test data that was inserted or modified during the tests.

If you look closely, you will see that all our test classes inherit from the `BaseDaoTests` class, which allows them all have their data rolled back after each test automatically.

# USING TRANSACTIONS FOR DAO INTEGRATION TESTING

When integration tests run, they should not modify the actual data in the database.

We accomplish this 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 create a **SingleConnectionDataSource** object rather than a **BasicDataSource**. Using this particular implementation of **DataSource** every database interaction is part of the same database session and hence the same database transaction

# USING TRANSACTIONS FOR DAO INTEGRATION TESTING

Using Spring, we can set up an alternate datasource to be used for tests.

In our tests, we will use a **SingleConnectionDataSource** rather than **BasicDataSource** to allow us to turn off the functionality of committing automatically when a SQL statement is executed.

We will leverage the ability to use transactions that don't automatically commit to use real data without impacting the database by rolling back any database changes once a test is complete.

# USING TRANSACTIONS FOR DAO INTEGRATION TESTING

In order to allow us to rollback transactions, we use `setAutoCommit(false)` when creating the test datasource. 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.

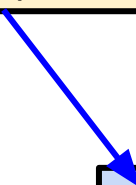
```
@Bean
public DataSource dataSource() throws SQLException {
    SingleConnectionDataSource dataSource = new SingleConnectionDataSource();
    dataSource.setUrl("jdbc:postgresql://localhost:5432/UnitedStates");
    dataSource.setUsername("postgres");
    dataSource.setPassword("postgres1");
    dataSource.setAutoCommit(false);
    return dataSource;
}
```

We call `setAutoCommit(false)` here.

# 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.

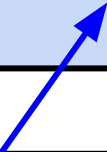
We use the connection of the `SingleConnectionDataSource` to roll back any database changes caused by the test.

```
@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.

```
public void doCleanup() {  
    dataSource.destroy();  
}
```



In the code that gets called when the tests are complete, we call the `destroy()` method of the `SingleConnectionDataSource` to make sure it is cleaned up properly.