Module 2-8

Integration Testing

Integration Testing vs Unit Testing

- Unit Testing ensures that discrete single components of your code (i.e. methods) work correctly.

- Integration testing ensures that when various components (i.e. several methods, possibly from several classes) work well together.

Integration Testing

Integration testing is designed to answer some bigger questions, for example:

- A unit test asserts that given two inputs, the method produces a specific output.

- An integration test could ask for example - “Can I add a new customer to the database?” A task that might include calls to several methods across several classes.

JUnit Method Level Annotations

@Before: the method will be run before each @Test.

@After: the method will be run after each @Test.

@BeforeClass: the method is run once before any and all @Test methods.

@AfterClass: the method is run once after all @Test methods have finished.

Database Considerations

We will be inevitably be creating “test data” while integration testing:

- When testing any functionality that inserts new data.
- When testing any functionality that updates data.
- When testing any functionality that removes rows of data.

Obviously we don’t these to be permanent changes, so our integration tests will need to **roll back** these changes when we’re done.

Database Considerations: The SingleConnectionDataSource class

---

- For production code we will be using **BasicDataSource**.

- For integration tests we want to use **SingleConnectionDataSource**.

- Per the Spring documentation, **SingleConnectionDataSource** is the preferred implementation of DataSource for testing.

Database Considerations: The SingleConnectionDataSource class

---

Note that both **SingleConnectionDataSource** and **BasicDataSource** are implementations of DataSource.

Database Considerations: The SingleConnectionDataSource class

---

We generally setup the data source in the @BeforeClass method:

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

Note how autocommit is set to false.

Database Considerations: Rolling back any changes

We generally want to roll back any changes done during an @Test by implementing a rollback in the @After.

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

Database Considerations: Destroy the Data Source

In the @AfterClass we destroy the data source.

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