Unit-Testing Database Operations with DBUnit

Clark D. Richey, Jr., Principal Technologist, Mark Logic
James Morgan, Senior Software Engineer, SRA

TD-5859

MARK LOGIC

Java

Sun microsystems

Learn how to effectively unit test your data access code using DBUnit

Agenda

- Unit testing and databases
- Bare bones unit testing
- DBUnit Basics
- Advanced Topics
- Limitations

Agenda

- Unit testing and databases
- Bare bones unit testing
- DBUnit Basics
- Advanced Topics
- Limitations

Why Unit Test Data Access Code?

- Data access code is still code
  - Always unit test code, right?
- Need a way to validate against a variety of data
- Persistence frameworks (JPA, JDO, etc) help but…
  - Frameworks can be misconfigured
  - Frameworks can be asked to do the wrong thing
    - Annotate wrong class
    - Incorrect mappings

Testing with a Persistence Framework

- Trust that the framework itself works
- Verify proper usage of the framework
- No different than testing JDBC based code
- As with most good unit tests, the implementation of the method to be tested is independent from the unit test

Good Unit Tests Are...

Automatic

You don't have to manually inspect the results;
your framework tells you if the tests pass or fail

Good Unit Tests Are…

Thorough

Operations are tested against varying data to ensure that no side effects occur (e.g. existing rows are unaffected by an insert)

Operations are tested against a database that is in various states (e.g. tests against empty tables)

Verify that transactions work as expected

Good Unit Tests Are...

Repeatable

Execute the same test against the database multiple times with the same result

The database must be put into a known state prior to each test

Good Unit Tests Are…

Independent

Previous tests don't affect the currently executing test

The database must be put into a known state prior to each test

Good Unit Tests Are...

Professional

Test code is of the same quality as the business code

Unit Testing Best Practices

- Create multiple databases per developer
  - One for development (db name_dev)
  - One for testing (db name_test)
- Ensure the state of the database prior to testing
- Test in small chunks of data
  - Don't try to load everything into the database for a single test

Agenda

- Unit testing and databases
- Bare bones unit testing
- DBUnit Basics
- Advanced Topics
- Limitations

JUnit Test Outline

- Setup the database connection
- Initialize the test database with test data
- Execute the code to be tested
- Thoroughly test the results
  - Make sure to verify that there are no unwanted side effects
- Cleanup the database connections

Database Housekeeping

```
private Connection connection;
private EmployeeJDBCPersistence persistence;

@Before public void setup() {
  persistence = new EmployeeJDBCPersistence();
}

private void initializeConnection() throws Exception
{
  Class.forName("com.mysql.jdbc.Driver");
  connection = DriverManager.getConnection(url);
}
```

More Database Housekeeping

```
private void emptyDatabase() throws SQLException {
  Statement statement =
    connection.createStatement();

  statement.executeUpdate("delete from employees");

  statement.close();
}
```

Setup the Data

```
public void testUpdateEmployee() throws Exception {
  initializeConnection();

  emptyDatabase();

  Statement insertStatement =
    connection.createStatement();

  insertStatement.execute("insert into employees
    values('bob@bigcompany.com', 'Bob', 'Smith')");
  insertStatement.execute("insert into employees
    values('sally@bigcompany.com', 'Sally',
    'Smith')");
  insertStatement.execute("insert into employees
    values('ron@bigcompany.com', 'Ron', 'Fink')");

  insertStatement.close();
```

Perform the Update

```
Employee testEmployee = new
  Employee("sally@bigcompany.com", "Sally", "Fink");

persistence.updateEmployee(testEmployee);
```

Test the Results

```
Statement selectStatement =
  connection.createStatement();

ResultSet employeeResults =
  selectStatement.executeQuery("Select * from
  employees where email = 'sally@bigcompany.com'");

assertThat(employeeResults.next(), is(true));
```

Keep Testing the Results

```
String email = employeeResults.getString("email");
String firstName =
  employeeResults.getString("firstName");
String lastName =
  employeeResults.getString("lastName");

assertThat(testEmployee.getEmailAddress(),
  equalTo(email));

assertThat(testEmployee.getFirstName(),
  equalTo(firstName));

assertThat(testEmployee.getLastName(),
  equalTo(lastName));
```

Test for Side Effects

```
// have other employees been modified?

employeeResults =
  selectStatement.executeQuery("Select * from
  employees where email = 'bob@bigcompany.com'");

assertThat(employeeResults.next(), is(true));

email = employeeResults.getString("email");
firstName = employeeResults.getString("firstName");
lastName = employeeResults.getString("lastName");

assertThat("bob@bigcompany.com", equalTo(email));
assertThat("Bob", equalTo(firstName));
assertThat("Smith", equalTo(lastName));
```

Still Testing for Side Effects…

```
employeeResults =
  selectStatement.executeQuery("Select * from
  employees where email = 'ron@bigcompany.com'");

assertThat(employeeResults.next(), is(true));

email = employeeResults.getString("email");
firstName = employeeResults.getString("firstName");
lastName = employeeResults.getString("lastName");

assertThat("ron@bigcompany.com", equalTo(email));
assertThat("Ron", equalTo(firstName));
assertThat("Fink", equalTo(lastName));
```

Yet More Testing for Side Effects…

```
// Were new employees created in the update process?

employeeResults =
  selectStatement.executeQuery("select count(*) as
  total FROM employees");

assertThat(employeeResults.next(), is(true));

int totalEmployees =
  employeeResults.getInt("total");

assertThat(totalEmployees, is(3));
```

Cleanup

```
employeeResults.close();
selectStatement.close();

connection.close();
```

JUnit Testing

JUnit Summary

- Thorough testing requires a lot of code
  - Not boilerplate – corner cases and side effects must be looked for
- Extensive coding required to initialize the database
- JUnit is not able to directly facilitate any of this
- Takes too long and is too error prone

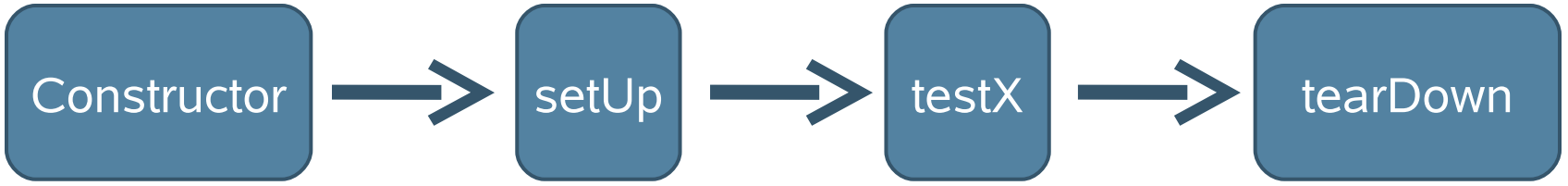
Agenda

- Unit testing and databases
- Bare bones unit testing
- DBUnit Basics
- Advanced Topics
- Limitations

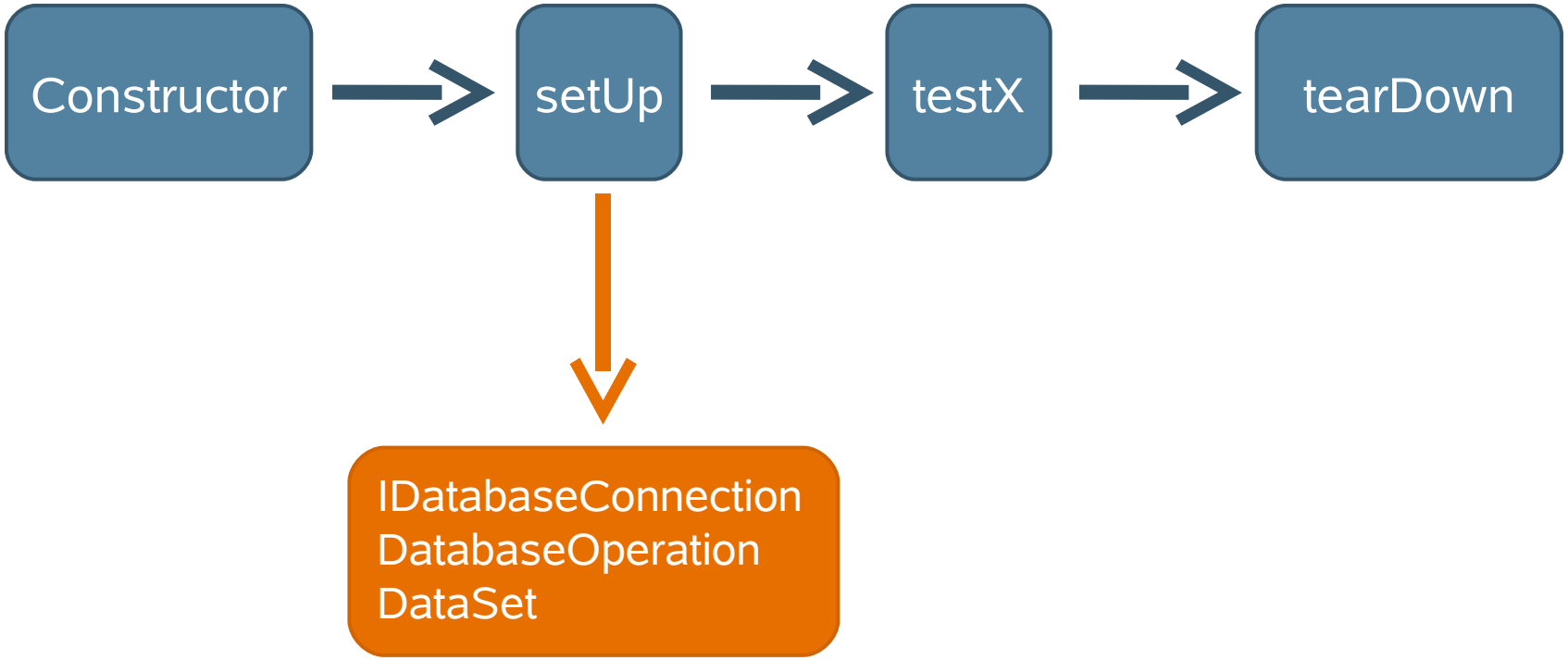
DBUnit Basics

- http://www.dbunit.org/
- JUnit extension for databases
- Provides ability to
  - Ensure database state
  - Compare actual database state to expected
  - Filter and sort data as needed

Simplified DBUnit Lifecycle



Simplified DBUnit Lifecycle



DBUnit Best Practice

- Don't extend DBTestCase
  - Locks you into JUnit
  - DBUnit lifecycle can make it more difficult to perform certain tests
  - Just isn't necessary
- Simply leverage the fundamental DBUnit classes

DBUnit Fundamental Objects

- IDatabaseTester
  - Responsible for adding DBUnit features as composition on existing test cases (instead of extending DBTestCase directly)
- Implementations
  - DefaultDatabaseTester – provides no database connectivity
  - DataSourceDatabaseTester
  - JdbcDatabaseTester
  - JndiDatabaseTester

DBUnit Fundamental Objects

- ITable
  - Represents a collection of tabular data
  - Used in assertions to compare actual database tables to expected tables
- Implementations
  - DefaultTable
  - PrimaryKeyFilteredTableWrapper
  - SortedTable

DBUnit Fundamental Objects

- DatabaseOperation
  - Represents an operation performed on the database before and after each test
  - DatabaseOperation.CLEAN_INSERT
  - DatabaseOperation.REFRESH
  - DatabaseOperation.DELETE_ALL

DBUnit Fundamental Objects

- IDataSet
  - Collection of tables
  - Used to place the database into a known state
  - Used to compare current database state against expected state
- Implementations
  - FlatXmlDataSet
  - StreamingDataSet
  - XmlDataSet

FlatXmlDataSet

- Reads and writes flat XML dataset document
- Each XML element corresponds to a table row
- Each XML element name corresponds to a table name

Example FlatXmlDataSet

```
dataset>

   <employees email=“bob@bigcompany.com”

     firstName="Bob" lastName="Smith"/>

   <employees email=“sally@bigcompany.com”

     firstName="Sally" lastName="Smith"/>

   <employees email="ron@bigcompany.com”

     firstName="Ron" lastName="Fink"/>
```

Example FlatXmlDataSet

```
dataset>

    <employees email="bob@bigcompany.com"

      firstName="Bob" lastName="Smith"/>

    <employees email="sally@bigcompany.com"

      firstName="Sally" lastName="Smith"/>

    <employees email="ron@bigcompany.com"

      firstName="Ron" lastName="Fink"/>
```

Example FlatXmlDataSet

```
dataset>

   <employees email="bob@bigcompany.com"
     firstName="Bob" lastName="Smith"/>

   <employees email="sally@bigcompany.com"
     firstName="Sally" lastName="Smith"/>

   <employees email="ron@bigcompany.com"
     firstName="Ron" lastName="Fink"/>
```

DBUnit Outline

- Setup the database connection
- Initialize the test database with test data
- Execute the code to be tested
- Thoroughly test the results
  - Make sure to verify that there are no unwanted side effects
- Cleanup the database connections
- Same process as with JUnit but with less code

Housekeeping

```
rivate EmployeeJDBCPersistence persistence;

rivate IDatabaseTester databaseTester;


 @BeforeClass

 public void setUp() {

   persistence = new EmployeeJDBCPersistence();

   databaseTester = new

     JdbcDatabaseTester("com.mysql.jdbc.Driver",uri);
```

Setup the Initial Database State

```
ublic void testUpdateEmployee() throws Exception {

 IDatabaseConnection connection =

   databaseTester.getConnection();

 InputStream dataStream =

etClass().getResourceAsStream("InitialEmployeeUpdateDa
ta.xml");


 IDataSet initialDataSet = new

   FlatXmlDataSet(dataStream);
```

Perform the Update

```
mployee testEmployee = new

 Employee("sally@bigcompany.com", "Sally", "Fink");

ersistence.updateEmployee(testEmployee);
```

Test the Update

```
ataStream =
getClass().getResourceAsStream("ExpectedEmployeeUpdate
Data.xml");

Table expectedEmployeeTable = new

 FlatXmlDataSet(dataStream).getTable("employees");

Table actualTable =

 connection.createDataSet().getTable("employees");


ssertion.assertEquals(new

SortedTable(expectedEmployeeTable), new
```

DBUnit test

Agenda

- Unit testing and databases
- Bare bones unit testing
- DBUnit Basics
- Advanced Topics
- Limitations

Sorting

- Default sorting is by primary key for tables retrieved from the database
- Expected dataset may be in a different order

Initial Dataset

```
<dataset>
    <employees email="bob@bigcompany.com"
  firstName="Bob" lastName="Smith"/>
    <employees email="sally@bigcompany.com"
  firstName="Sally" lastName="Smith"/>
    <employees email="ron@bigcompany.com"
  firstName="Ron" lastName="Fink"/>
</dataset>
```

Expected Dataset

```
<dataset>
    <employees email="bob@bigcompany.com"
  firstName="Bob" lastName="Smith"/>
    <employees email="sally@bigcompany.com"
  firstName="Sally" lastName="Fink"/>
    <employees email="ron@bigcompany.com"
  firstName="Ron" lastName="Fink"/>
</dataset>
```

Default – Sorted by Primary Key (email)

```
<dataset>
    <employees email="bob@bigcompany.com"
  firstName="Bob" lastName="Smith"/>
    <employees email="ron@bigcompany.com"
  firstName="Ron" lastName="Fink"/>
    <employees email="sally@bigcompany.com"
  firstName="Sally" lastName="Fink"/>
</dataset>
```

SortedTable

- Decorator for an existing table
- SortedTable(ITable table)
  - Sort the decorated table by its own columns' order
- SortedTable(ITable table, ITableMetaData metaData)
  - Sort the decorated table by specified metadata columns' order

Sorting

Filtering

- Useful for removing generated primary key values from comparisons
  - keys generated by a sequence
- Removing columns that don't matter to the test
  - time stamps

DefaultColumnFilter

- IColumnFilter implementation that exposes columns matching include patterns and not matching exclude patterns
- excludeColumn(String columnPattern)
  - Wildcard characters supported
    - * - 0 or more
    - ? - 0 or 1
- includeColumn(String columnPattern)
  - Wildcard characters supported
    - * - 0 or more
    - ? - 0 or 1

Filtering

Agenda

- Unit testing and databases
- Bare bones unit testing
- DBUnit Basics
- Advanced Topics
- Limitations

DBUnit Limitations

- Not especially useful for validating queries
  - Query results aren't reflected in the database
  - Query results can be validated through standard methods
  - DBUnit can still be used to populate the database with test data
- Dealing with foreign keys on auto generated ids is tricky
  - Relationships can't always be validated

For More Information

- http://www.dbunit.org/
- Felipe Leme's blog
  - http://weblogs.java.net/blog/felipeal/
- Effective Unit Testing with DBUnit
  - http://www.onjava.com/pub/a/onjava/2004/01/21/dbunit.html
- BOF-5101 Boosting Your Testing Productivity with Groovy

THANK YOU

Clark D. Richey, Jr., Principal Technologist, Mark Logic
James Morgan, Senior Software Engineer, SRA

TS-5859