Testing Policy

Project Name	:	ERP Coin
Version Number	:	Version 1.1
Organisation	:	EPI-USE
Authors	:	Team Turing:

Kyle Pretorius
Darius Scheepers
Sewis van Wyk
Ulrik de Muelenaere
Tristan Rothman
Richard Dixie

Testing Policy

Use cases are to be tested before a branch is pushed up to dev, and after more extensive testing the dev branch may be pushed to master.

Backend Testing

For the testing of backend algorithms a tool call Mocha, together with a tool called Chai will be used.

The use of this tool can be explored using the following link:

http://stackabuse.com/testing-node-js-code-with-mocha-and-chai/

TravisCI will also be used on GitHub.

TravisCI run tests everytime something is added to GitHub.

Thus, tests must be set up for every repository that is pushed to master.

The use of TravisCI can be exlored using the following link:

https://code.tutsplus.com/tutorials/travis-ci-what-why-how--net-34771

Database Testing

For database testing a tool will be used called DBFit.

The appended information is retirieved from http://www.methodsandtools.com/, explaining how to use DBFit and how it is used to test Database connection, Queries, Insertion and Updates:

Installation

To run DbFit you need Java Runtime Environment (JRE) 7 or higher which you may get from Oracle web site. Also you'll need to ensure the accounts and connectivity to the databases which you want to access with DbFit.

To install DbFit you can download it from its web site http://dbfit.github.io/dbfit/ unpack the archive and run startFitnesse.sh or startFitnesse.bat (depending on your operating system). It may take it a few seconds to start the server and then you can access it using your browser at http://localhost:8085/. You'll be able to access some built-in reference documentation and examples from that web page.

Creating a new test page

Open http://localhost:8085/HelloWorldTest in your browser. You should see an editor where you can create and run your test page.

Test pages are described in the FitNesse Wiki syntax that is essentially a simple text format for describing commands in terms of tables. The premise is that this notation is close to relational model, so that you don't need to know Java or other lower-level programming language to create DbFit tests.

Test pages may be grouped in suites that may be organized in hierarchical structure. The names of test pages and suites should be CamelCase.

Although the FitNesse Wiki syntax is really simple, you do not have to use it to write scripts. You can write your tables in Excel (or almost any other spreadsheet program), and then just copy or import them into the FitNesse page editor.

Test pages are stored on the file system in simple text format. You can create and edit them directly using your favorite text editor.

Running tests

To run a test page, you can click the Test button in FitNesse. It is possible to run all tests within a suite and it is also possible to trigger tests execution via REST API, JUnit or a command line allowing you to automate running DbFit tests and integration with a continuous integration (CI) server.

Configuration

For details about configuring different port number, project folder or other settings you may examine the startFitnesse script and its --help option output, plugins.properties file, and also consult FitNesse documentation.

Due to licensing restrictions, the drivers of some of the supported databases are not shipped with DbFit. You will need to provide the required additional JDBC driver jar files by copying them into DbFit lib folder.

In order to load the DbFit extension into FitNesse, your test pages have to load the correct libraries by including the following command:

```
!path lib/*.jar
```

This command can be placed directly into a test page or in a parent page in the test suite hierarchy.

Connecting to the database

DbFit supports two modes of operation: Flow and Standalone. By default, each individual test page in flow mode is executed in a transaction that is automatically rolled back after the test. In the standalone mode, you are responsible for overall transaction control. Most of the commands are common for both modes, but there are some limitations in standalone mode a different syntax for opening a connection.

Here is how to connect to a MySQL database in flow mode:

```
!|dbfit.MySqlTest|
!|Connect|localhost|dbfit_user|password|dbfit|
```

Notice the MySqlTest in the first line above. That tells DbFit which type of database driver to use. The second line defines connection properties. These two lines will typically be the first on every test page.

And here is how to open a connection in standalone mode:

```
|importfixture|
|dbfit.fixture|
!|DatabaseEnvironment|MySql|
|Connect|localhost|dbfit_user|password|dbfit|
```

There is support for configuring the connection details in separate file. Encryption of the database password is also supported so that you can avoid storing it in plain text.

Supported databases

The following database systems are currently supported by DbFit: Oracle, SQL Server, MySQL, Postgres, Derby, HSQLDB, DB2, DB2i, Teradata, Netezza, Informix.

Basic cycle of a DbFit test and its execution

1.Set up the input data and other context. The following DbFit commands are typically used at this stage: Insert, Update, ExecuteDdl, Execute. If common setup steps are used by several tests - it may be useful to keep them in SetUp page that will be automatically included in the beginning of the other tests in the suite.
2.Execute a database operation that you want to test (typically - a stored function or procedure, or an external process such as an ETL tool routine). Useful DbFit commands: Execute Procedure.
3.Verify the result - comparing the actual versus the expected result. Useful DbFit commands: Query, Store Query, Compare Stored Queries
4.Tear Down - clean up the changes in order to avoid affecting other tests. Pages named TearDown are automatically included at the end of all tests in the suite. In Flow mode, the active transaction is automatically rolled back at the end of the test page so often you won't need any explicit tear down steps (assuming you don't commit transactions in your tests).

Quick reference of the mostly used commands

Query

The Query command allows you to compare the result of a SQL Query to a given expected output. You should specify query as the first fixture parameter, after the Query command. The second table row contains column names, and all subsequent rows contain data for the expected results. You don't have to list all columns in the result set, just the ones that you are interested in testing. Multiple columns and rows are supported.

```
!|Query|select 'test1' as column_one, 'test2' as column_two from dual|
|column_one |column_two|
|test1 |test2 |
```

As output of the Query command execution, matches are marked in green and mismatches are highlighted in red. In case of any mismatch - the test is counted as failing. Missing or unexpected surplus records are also being reported as mismatch.

Ordered Query is a variation of Query command that takes into account the exact ordering of rows.

Store Query, Compare Stored Queries

Store Query reads out query results and stores them into a Fixture symbol for later use. Specify the full query as the first argument and the symbol name as the second argument (without >>). You can then use this stored result set as a parameter of the Query command later:

```
!|Store Query|select n from (select 1 as n from dual union select 2 from dual
union select 3 from dual)|firsttable|
!|Query|<<firsttable|
|n|
|1|
|2|
|3|
```

Compare Stored Queries compares two previously stored query results. Column structure is specified so that some columns can be ignored during comparison (just don’t list them), and for the partial row-key mapping to work. Put a ? after the column names that do not belong to the key to make the comparisons better.

```
|Store Query|select n, name from testtbl|fromtable|
|Store Query|!- select 1 n, 'name1' name from dual|fromdual|
|Compare Stored Queries|fromtable|fromdual|
|name |n? |
```

Typically you should strive for fast and focused tests, comparing just a small amount of records. Nevertheless, in some special cases you could want to run DbFit on top of a relatively large number of rows. Please refer to the DbFit reference guide for some guidelines how to handle such scenarios more efficiently.

Insert

Insert command allows inserting a set of records into a database table (or view). The view or table name is given as the first fixture parameter. The second row contains column names, and all subsequent rows contain data to be inserted. For example:

```
!|Insert|testtbl|
|id|name|
|1|NAME1|
|3|NAME3|
|2 |NAME2|
```

Update

Update allows you to update specified records in a database table or view. Columns ending with = are used to update records. Columns without = are used to select rows. The following example updates the name column where the id matches 1 and 2.

```
!|Update |testtbl|
|name= |id |
|New Name1|1 |
|New Name2|2 |
```

You can use multiple columns for both updating and selecting, and even use the same column for both operations.

Execute and Execute Ddl

Execute executes SQL statement provided using the syntax relevant for the database. You can use query parameters in the DB-specific syntax (eg. @paramname for SQLServer and MySQL, and :paramname for Oracle). Currently, all parameters are used as inputs, and there is no option to persist any statement outputs.

```
!|Execute|alter session set ddl_lock_timeout=600|
!|Set Parameter|name|Darth Maul|
!|Execute|insert into test_table(name, age) values (:name, 10)|
```

Execute Ddl is intended for executing DDL SQL statements (like create, alter, drop). It's similar to Execute with the main difference that it does not support bind variables, and also automatically handles required post-execute activities, if any (e.g. Teradata requires transaction to be closed after each DDL statement).

```
!|Execute Ddl|create table tab_with_trigger(x int)|
!|Execute Ddl|create or replace trigger trg_double_x before insert on
tab_with_trigger for each row begin :new.x := :new.x * 2; end;|
!|Insert|tab_with_trigger|
|x |
|13 |
!|Query|select x from tab_with_trigger|
|x |
|26 |
```

Execute Procedure

Execute Procedure executes a stored procedure or function for each row of data table, binding input/output parameters to columns of the data table, e.g.:

```
!|Execute Procedure|ConcatenateStrings |
|first_string|second_string|concatenated?|
|Hello |World |Hello World |
|Ford |Prefect |Ford Prefect |
```

If a function is getting called, then a column containing just the question mark is used for function results.

```
!|Execute Procedure|ConcatenateF |
|first_string|second_string|? |
|Hello |World |Hello World |
|Ford |Prefect |Ford Prefect|
```

To use IN/OUT parameters, you will need to specify the parameter twice. Once without the question mark, when it is used as the input; and one with the question mark when it is used as output:

```
|Execute Procedure|Multiply|
|factor|val|val? |
|5 |10 |50 |
```

To determine the test result, the actual output and return values are compared with the specified expected ones.

There is a variant Execute Procedure Expect Exception in case you expect your stored procedure to raise an exception.