etlTest Documentation

Release 0.1.2 - beta

Alex Meadows, Coty Sutherland

October 02, 2014

Contents

etlTest is a testing framework for data integration code.

The home for etlTest is on GitHub .

The goal of etlTest is to make creating and executing tests for data integration code as simple as possible. We have achieved this through the use of YAML files to store test data sets and the actual tests. Those files then get translated into Python unittests and can be executed from the command line. The results will show where there is more work to be done.

We have initially focused on unit tests, but we are planning on building out to cover other types of testing as well.

We are always looking for new feature requests, bugs, and other contributions! To learn more on how to contribute, please refer to our *Contributing* page. To see what is on the roadmap, please check out our Issue Board.

CHAPTER 1

Introduction

This section provides high level details on etlTest and it's various components.

1.1 Overview

Below is an overview of what etlTest is and how it works. It is important to understand some fundamental concepts:

- Unit testing
- Test driven development
- Continuous Integration

1.2 Why etlTest?

Data integration tools do not have standard output in terms of code. To make matters even more interesting, many of them do not integrate with external version control systems (like Subversion or Git) let alone have a universal way to test code. etlTest aims to change that last part by providing a universal way to work with data integration tests. This way, regardless of the data source or data integration tool your tests will be able to be used with minimal effort to convert them over when the stack you're working on changes.

1.3 How Does It Work?

Developing tests in etlTest is designed to be as simple as possible. All that is required (other than installing etlTest ;)) is to generate a sample data file...

```
//etltest/samples/data/etlUnitTest/users.yml
 1:
   user_id: 1
   first_name:  Bob
   last_name:  Richards
   birthday:  2000-01-04
   zipcode:  55555
   is_active: 0
 2:
   user_id: 2
   first_name:  Sarah
   last_name: Jenkins
```

```
   birthday:  2000-02-02
   zipcode:  12345
   is_active: 1
 ...
```

and a test file...

```
//etltest/samples/test/dataMart/users_dim.yml
DataMart\UsersDim:
   suites:
     - suite: dataMart
   processes:
     - tool:  PDI
       processes:
         - name:  data_mart/user_dim_jb.kjb
           type:  job
   dataset:
     - source:  etlUnitTest
       table:  users
       records:  [1, 2]
   tests:
     - name: testFirstNameLower
       desc:  Test for process that lower cases the first name field of a users table record.
       type: NotEqual
       query:
         select: first_name
         from: user_dim
         where: user_id = 2
         source:  etlUnitTest
         result: {'first_name': 'sarah'}
```

See *sample data file standards* and *test file standards* for full template details.

Once your tests have been written, you can then have etlTest generate and execute your code.

```
$ etlTest.py -f <path_to_your_test.yml> -o <path_to_your_output_dir> -g -e
```

Which will generate and run something similar to:

```
//etltest/samples/output/DataMart/UsersDim.py
#!/usr/bin/python
#
# This file was created by etlTest.
#

# These tests are also run as part of the following suites:
#
#    dataMart
#
# The following processes are executed for these tests:
#
#    PDI:
#      data_mart/user_dim_jb.kjb

import unittest
import datetime
from os import path

from etltest.data_connector import DataConnector
from etltest.process_executor import ProcessExecutor
```

```
from etltest.utilities.settings_manager import SettingsManager


class DataMartUsersDimTest(unittest.TestCase):

    def setUp(self):
          # Queries for loading test data.
            DataConnector("etlUnitTest").insert_data("users", [1, 2])

            PDI_settings = SettingsManager().get_tool("PDI")
            PDI_code_path = SettingsManager().system_variable_replace(PDI_settings["code_path"])
            ProcessExecutor("PDI").execute_process("job",
            path.join(PDI_code_path, "data_mart/user_dim_jb.kjb"))

    def tearDown(self):
       # Clean up testing environment.

        DataConnector("etlUnitTest").truncate_data("users")

    def testFirstNameLower(self):
        # Test for process that lower cases the first name field of a users table record.

        given_result = DataConnector("etlUnitTest").select_data("first_name",
                        "user_dim", "user_id = 2")

        expected_result = [{'first_name': 'sarah'}]

        self.assertNotEqual(given_result, expected_result)

if __name__ == "__main__":
    unittest.main()
```

Notice that etlTest generates actual Python code so that you can leverage a full blown testing framework without writing a single line of code! We'll go over the various components of the test suites in *Test Components*

1.3.1 Overview

Below is an overview of what etlTest is and how it works. It is important to understand some fundamental concepts:

- Unit testing
- Test driven development
- Continuous Integration

1.3.2 Why etlTest?

Data integration tools do not have standard output in terms of code. To make matters even more interesting, many of them do not integrate with external version control systems (like Subversion or Git) let alone have a universal way to test code. etlTest aims to change that last part by providing a universal way to work with data integration tests. This way, regardless of the data source or data integration tool your tests will be able to be used with minimal effort to convert them over when the stack you're working on changes.

1.3.3 How Does It Work?

Developing tests in etlTest is designed to be as simple as possible. All that is required (other than installing etlTest ;)) is to generate a sample data file...

```
//etltest/samples/data/etlUnitTest/users.yml
 1:
   user_id: 1
   first_name:  Bob
   last_name:  Richards
   birthday:  2000-01-04
   zipcode:  55555
   is_active: 0
 2:
   user_id: 2
   first_name:  Sarah
   last_name: Jenkins
   birthday:  2000-02-02
   zipcode:  12345
   is_active: 1
 ...
```

and a test file...

```
//etltest/samples/test/dataMart/users_dim.yml
DataMart\UsersDim:
   suites:
     - suite: dataMart
   processes:
     - tool:  PDI
       processes:
         - name:  data_mart/user_dim_jb.kjb
           type:  job
   dataset:
     - source:  etlUnitTest
       table:  users
       records:  [1, 2]
   tests:
     - name: testFirstNameLower
       desc:  Test for process that lower cases the first name field of a users table record.
       type: NotEqual
       query:
         select: first_name
         from: user_dim
         where: user_id = 2
         source:  etlUnitTest
         result: {'first_name': 'sarah'}
```

See *sample data file standards* and *test file standards* for full template details.

Once your tests have been written, you can then have etlTest generate and execute your code.

```
$ etlTest.py -f <path_to_your_test.yml> -o <path_to_your_output_dir> -g -e
```

Which will generate and run something similar to:

```
//etltest/samples/output/DataMart/UsersDim.py
#!/usr/bin/python
#
# This file was created by etlTest.
```

```
#

# These tests are also run as part of the following suites:
#
#    dataMart
#
# The following processes are executed for these tests:
#
#    PDI:
#      data_mart/user_dim_jb.kjb

import unittest
import datetime
from os import path

from etltest.data_connector import DataConnector
from etltest.process_executor import ProcessExecutor
from etltest.utilities.settings_manager import SettingsManager


class DataMartUsersDimTest(unittest.TestCase):

    def setUp(self):
          # Queries for loading test data.
            DataConnector("etlUnitTest").insert_data("users", [1, 2])

            PDI_settings = SettingsManager().get_tool("PDI")
            PDI_code_path = SettingsManager().system_variable_replace(PDI_settings["code_path"])
            ProcessExecutor("PDI").execute_process("job",
            path.join(PDI_code_path, "data_mart/user_dim_jb.kjb"))

    def tearDown(self):
       # Clean up testing environment.

        DataConnector("etlUnitTest").truncate_data("users")

    def testFirstNameLower(self):
        # Test for process that lower cases the first name field of a users table record.

        given_result = DataConnector("etlUnitTest").select_data("first_name",
                        "user_dim", "user_id = 2")

        expected_result = [{'first_name': 'sarah'}]

        self.assertNotEqual(given_result, expected_result)

if __name__ == "__main__":
    unittest.main()
```

Notice that etlTest generates actual Python code so that you can leverage a full blown testing framework without writing a single line of code! We'll go over the various components of the test suites in *Test Components*

CHAPTER 2

How to contribute

Want to participate/contribute to etlTest? Feel free to add any enhancements, feature requests, etc.

2.1 Getting Started

- Create a new, Python 2.7+ virtualenv and install the requirements via pip:

  $ pip install -r requirements.txt

- Make sure you have a GitHub account
- Submit issues/suggestions to the Github issue tracker * For bugs, clearly describe the issue including steps to reproduce. Please include stack traces, logs, screen shots, etc. to help us identify and address the issue. * For text based artifacts, please use: Gist or Pastebin * For enhancement requests, be sure to indicate if you are willing to work on implementing the enhancement * Fork the repository on GitHub if you want to contribute code/docs

2.2 Making Changes

- **etlTest** uses git-flow as the git branching model
  - **All commits should be made to the dev branch**
  - Install git-flow and create a *feature* branch with the following command:

    ```
    $ git flow feature start <name of your feature>
    ```

- Make commits of logical units with complete documentation.
- Check for unnecessary whitespace with *git diff –check* before committing.
- Make sure you have added the necessary tests for your changes.
  - Test coverage is currently tracked via coveralls.io
  - Aim for 100% coverage on your code
    - If this is not possible, explain why in your commit message. This may be an indication that your code should be refactored.
- To make sure your tests pass, run:

```
$ python setup.py test
```

- If you have the *coverage* package installed to generate coverage data, run:

```
$ coverage run --source=etltest setup.py test
```

- Check your coverage by running:

```
$ coverage report
```

2.3 Submitting Changes

- Push your changes to the feature branch in your fork of the repository.
- Submit a pull request to the main repository
- You will be notified if the pull was successful. If there are any concerns or issues, a member of the etlTest maintainer group will reach out.

2.4 Additional Resources

- General GitHub documentation
- GitHub pull request documentation

CHAPTER 3

Setting Up

Installing etlTest is as simple as running:

```
$ pip install etlTest
```

All the requirements will be installed and etlTest is ready to be used.

If you wish to install etlTest manually instead of via `pip`, there are two main options:

- GitHub
- PyPi

If going the manual install route, all requirements can be installed by running:

```
$ cd <where_etlTest_is_downloaded_and_extracted>
$ tox
```

All of etlTest's own unit tests will run once the requirements are installed. If everything passes, you are good to go.

CHAPTER 4

Configuration Settings

Below are the various configuration files used by etlTest.

4.1 etlTest Settings

This section describes the application-level configuration file and it's various options. It is not recommended to make changes to this section.

4.1.1 Default Settings

Here is the default `.etltest-settings.yml` file:

```
app_name: etlTest
logging_level: 20
app_author:  etlTest
settings_file:  properties.cfg
connection_file: connections.cfg
tools_file:  tools.yml
```

These are the settings used by etlTest when running.

- `app_name` is the name of the application. This should remain the default 'etlTest'.
- **`logging_level` is the level of logging desired while etlTest is running. The default is '20', which is the numerical value**
  - 5 - TRACE
  - 21 - TESTING
  - Standard logging level for Python are defined in the official docs .
- `app_author` is the name of the application author group. This should remain the default 'etlTest'.
- `settings_file` is the name of the file used for user-defined settings. The default is 'properties.cfg'.
- `connection_file` is the name of the file used for data tool connections. The default is 'connections.cfg'.
- `tools_file` is the name of the file used for data integration tool configuration. The default is 'tools.yml'.

4.2 Data Connections

This section describes the setting file used for all data source and target connectivity.

4.2.1 Default Settings

Here is the default sample of the `connections.cfg` file. This file can be found in the application settings directory, as described in *Settings*

```
[etlUnitTest]
hostname:  127.0.0.1
username:  root
password:
port:  3306
type:  mysql
dbname:  etlUnitTest
```

While this sample is written for MySQL/MariaDB, you can connect to any data source supported by SQLAlchemy. The full list of SQLAlchemy supported data sources can be found in their official documentation .

- `[etlUnitTest]` - The distinct name of the data source/target. Can be any valid string, as long as it does not break configuration file standards.
- `hostname:` - The host name or the ip address of the system that hosts the data source/target.
- `username:` - The username that will be used to connect to the data source/target.
- `password:` - The password for the user account connecting to the data source/target.
- `port:` - The port on the host that allows for connections to the data source/target.
- `type:` - The type of data source/target being connected to. Must be compliant with the types of SQLAlchemy dialects.
- `dbname:` - The name of the schema/database being connected to. Does not have to match the name used to define

the data source/target.

4.3 etlTest User Properties Settings

This section describes the settings file used for the user's environment settings.

4.3.1 Default Settings

Here is the default sample for `properties.cfg`. This file can be found in the application settings directory, as described in *Settings*

```
[Locations]
tests: ${ETL_TEST_ROOT}/Documents/etlTest/tests
data: ${ETL_TEST_ROOT}/Documents/etlTest/data
output: ${ETL_TEST_ROOT}/Documents/etlTest/output
[Results]
Verbose: True
FailureRate: 10
ReportType: Normal
```

While the sample is written with example paths, any valid directory path can be used. If the directory does not exist, it will be created.

- `[Locations]` - This section of the properties configuration file contains locations for the various inputs and outputs of etlTest.
- `tests:` - The location where the YAML test files are stored.
- `data:` - The location where the YAML data files are stored.
- `output:` - The location where the generated test scripts are created.
- `[Results]` - This section is currently not in use. The intent is to create user/environment based parameters

for how tests are run and the results shown. The parameters underneath are just examples and are ignored by etlTest.

4.4 Data Integration Tool Settings

This section describes the settings file used for Data Integration tool connectivity.

4.4.1 Default Settings

Here is the default sample for `tools.yml`. This file can be found in the application settings directory, as described in *Settings*

```
PDI:
  host_name:  localhost
  port:
  user_name:
  password:
  private_key: '~/.ssh/id_rsa'
  tool_path:  ${TOOL_PATH}
  code_path:  ${ETL_TEST_ROOT}/etltest/samples/etl/
  process_param:  "/file:"
  params:  "/level: Detailed"
  logging_filename_format:  ${name}_%Y-%m-%d
  script_types:
    - type: job
      script:  kitchen.sh
    - type: trans
      script:  pan.sh
```

While the sample is written for Pentaho Data Integration, it can be configured for any data integration tool that can be run from the command line.

- `PDI:` - The unique name of the tool. Can be any string as long as it does not break YAML standards.
- `host_name:` - The unique name or ip address of the host the tool lives on.
- `port:` - The port used to ssh onto the host box.
- `user_name:` - The name of the user account that is used to run data integration code.
- `password:` - The password of the user account that is used to run data integration code.
- `private_key:` - The private key to tunnel onto the box, if needed.
- `tool_path:` - The install location of the data integration tool.

- `code_path:` - The location of the data integration tool's code base. This is where etlTest will look for executable code.
- `process_param:` - Any custom parameters that have to be used to call the code. In PDI's case, files are called with the `/file:` property.
- `params:` - Any parameters that need to be tacked onto the back of the command. In PDI's case, logging is handled by the `/level:` parameter.
- `logging_filename_format:` - If storing of the process logs is desired, this is the format of the logging file name.
- `script_types:` - Multiple script types are allowed in the event that there are different components to the data integration tool.
- `type:` - The callable type of script.
- `script:` - The script that handles the type of process. This is located where the data integration tool is installed.

4.4.2 Sample Configurations

Here are some sample configurations for various tools that have been used with etlTest:

- *Contributing New Tool Configurations*
- *Pentaho Data Integration (file-based)*

Contributing New Tool Configurations

Are you using etlTest with a tool not listed here? Please consider contributing a sample tool setup! Find out how on our How to Contribute page.

Pentaho Data Integration (file-based)

```
PDI:
  tool_path:  ${TOOL_PATH}
  code_path:  ${ETL_TEST_ROOT}/etltest/samples/etl/
  process_param:  "/file:"
  params:  "/level: Detailed"
  logging_filename_format:  ${name}_%Y-%m-%d
  script_types:
    - type: job
      script:  kitchen.sh
    - type: trans
      script:  pan.sh
```

Special note: This sample takes advantage of two system variables:

- TOOL_PATH - Where the tool is installed (~/data-integration).
- ETL_TEST_ROOT - Where etlTest is installed, since we used the test samples for this sample tool configuration.

4.5 Settings File Location

The application settings file (`.etltest-settings.yml`) stays bundled with the application.

All other configuration files go into the data directory created by etlTest and is custom to the operating system that etlTest is installed on. etlTest takes advantage of a Python package named `appdirs` to handle configuration of the directories. At runtime, two directories are created:

- log - which handles logging for etlTest.
- application - which handles all other configuration files.

The location where these directories are set up is based on the `app_name` and `app_author` parameters. On Linux, the directories would be:

- log - ~/.cache/etlTest/log/
- application - ~/.local/share/etlTest/

Please review the appdirs documentation for more details.

4.6 Variable Replacement

Many of the values in these configuration files can be platform dependant. It makes sense to create system variables so that the tests are more portable. To use a system variable, enclose the name in `${your_value_here}` . For instance, to use a system variable named `$TOOL_HOME` call it as part of a configuration value like so: `${TOOL_HOME}/some/other/subdirectory` . The variable will be replaced with it's proper value.

CHAPTER 5

Development Standards

Here are the standards used for developing tests in etlTest.

5.1 Sample Data File Standards

Here are the standards for building sample/test data files.

5.2 Test File Standards

Here are the standards for building test files.

5.2.1 Available Test Types

The types of tests available are a subset of the assertion types that are made available with Python's unittest framework. To see more about the tests available in unittest, check out their documentation .

The list of available tests in etlTest is as follows:

etlTest Type	unittest Type	Test Description
Equal	assertEqual	Are given and expected equal?
NotEqual	assertNotEqual	Are given and expected no equal?
BooleanTrue	assertTrue	Is given true?
BooleanFalse	assertFalse	Is given false?
Is	assertIs	Are given and expected the same object?
IsNot	assertIsNot	Are given and expected not the same object?
IsNone	assertIsNone	Is given None?
IsNotNone	assertIsNotNone	Is given not None?
In	assertIn	Is given in expected?
NotIn	assertNotIn	Is given not in expected?
IsInstance	assertIsInstance	Is given an instance of expected?
IsNotInstance	assertIsNotInstance	Is given not an instance of expected?

5.3 Test Components

5.4 Test Templates

All of our tests are generated using the Jinja2 templating framework. This section will cover the various templates used and what they do.

5.4.1 Test Suite Template Overview

5.4.2 Test Template Overview

This template is used to build unit test sets. This overview will break down the template file and describe the various sections.:

```
{{ header }}
# These tests are also run as part of the following suites:
#
{% for suite in tests.suites %}
#    {{ suite.suite }}
{% endfor %}
#
# The following processes are executed for these tests:
#
{% for proc in tests.processes %}
#    {{ proc.tool }}:
  {% for p in proc.processes %}
#      {{ p.name }}
  {% endfor %}
{% endfor %}
```

This is the header section for our test files. It describes which test suites it is a part of (either unit test suites or others) as well as any ETL processes and tools that are used.

```
import unittest
import datetime
from os import path

from etltest.data_connector import DataConnector
from etltest.process_executor import ProcessExecutor
from etltest.utilities.settings_manager import SettingsManager
```

These are all of the requirements for the tests - both external (from other packages) and internal (from etlTest).

```
class {{ testGroup }}Test(unittest.TestCase):
```

Using the name of the testGroup (from the yaml test file) as part of the name of the test class.

```
@classmethod
    def setUpClass(cls):
          # Queries for loading test data.
      {% for set in tests.dataset %}
            DataConnector("{{ set.source }}").insert_data("{{ set.table }}", {{ set.records }})
      {% endfor %}

      {% for tool in tests.processes %}
            {{ tool.tool }}_settings = SettingsManager().get_tool("{{ tool.tool }}")
```

```
            {{ tool.tool }}_code_path = SettingsManager().system_variable_replace({{ tool.tool }}_set
      {% for job in tool.processes %}
            ProcessExecutor("{{ tool.tool }}").execute_process("{{ job.type }}",
            path.join({{ tool.tool }}_code_path, "{{ job.name }}"))
      {% endfor %}
      {% endfor %}
```

During the setup phase of the test the records that are used are inserted into the source database. The ETL processes that are listed in the header are executed here. This is only run once at the start of the run.

```
@classmethod
def tearDownClass(cls):
   # Clean up testing environment.

  {% for set in tests.dataset %}
    DataConnector("{{ set.source }}").truncate_data("{{ set.table }}")
  {% endfor %}
```

During the teardown phase of the test, the tables that had records inserted are truncated (this is a current limitation that we are trying to find a work around for). The teardown phase is only run once at the end of the run.

```
{% for test in tests.tests %}
    def {{ test.name }}(self):
        # {{ test.desc }}

        given_result = DataConnector("{{ test.query.source }}").select_data("{{ test.query.select }}"
                       "{{ test.query.from }}", "{{ test.query.where }}")
      {% if test.query.result is defined and test.query.result not in ('BooleanTrue', 'BooleanFalse',

        expected_result = [{{ test.query.result }}]
      {% endif %}

      {% if test.type == 'Equal' or test.type is not defined %}
        self.assertEqual(given_result, expected_result)
      {% elif test.type == 'NotEqual' %}
        self.assertNotEqual(given_result, expected_result)
      {% elif test.type == 'BooleanTrue' %}
        self.assertTrue(given_result)
      {% elif test.type == 'BooleanFalse' %}
        self.assertFalse(given_result)
      {% elif test.type == 'Is' %}
        self.assertIs(given_result, expected_result)
      {% elif test.type == 'IsNot' %}
        self.assertIsNot(given_result, expected_result)
      {% elif test.type == 'IsNone' %}
        self.assertIsNone(given_result)
      {% elif test.type == 'IsNotNone' %}
        self.assertIsNotNone(given_result)
      {% elif test.type == 'In' %}
        self.assertIn(given_result, expected_result)
      {% elif test.type == 'NotIn' %}
        self.assertNotIn(given_result, expected_result)
      {% elif test.type == 'IsInstance' %}
        self.assertIsInstance(given_result, expected_result)
      {% elif test.type == 'IsNotInstance' %}
        self.assertIsNotInstance(given_result, expected_result)
      {% else %}
        self.assertEqual(given_result, expected_result)
      {% endif %}
```

```
{% endfor %}
```

This is the actual test being generated. The test name is used for it's code equivalent. The query used for the test is put in as the given result while the expected result gets stored accordingly (if needed). Depending on the type of test used will determine the type of assertion used (which is the if statement that checks the test type).

```
if __name__ == "__main__":
    unittest.main()
```

This piece allows for the unit tests to be called based on the file name.

CHAPTER 6

The etlTest Tutorial

This is a walk through is intended to assist new users in writing, generating, and executing tests for data integration code. We have tried to keep topics specific to each function of the tool in their own section. This guide assumes that etlTest has been installed on the environment that the tutorial is being performed on. Please refer to the installation instructions found on the *Setup* page if etlTest has not been installed.

For the purposes of this tutorial, we will assume you have access to the following:

- Local MySQL instance (software is available from MySQL)
- Local Pentaho Data Integration instance (software is available from SourceForge)
- Database built using the `etlUnitTest_build.sql` script, found in the scripts directory of where etlTest is installed, or from the *Sample Database Script* page
- Sample data integration code, found in the samples/etl directory of where etlTest is installed (user_dim_jb.kjb and user_dim_load_tr.ktr)

In addition, while you are free to make any modifications to directory locations, we will be using the defaults found in the sample settings files.

6.1 Preparing Your Environment

When you run etlTest for the first time, it checks if it has everything it needs in place to generate and execute tests. Let's go ahead and run etlTest from the command line:

```
etlTest
```

This will now create a data directory (for configuration files) and a log directory (for log files). etlTest uses the appdirs package creating the directories specific to your operating system. Please refer to appdirs' docs (found here)for details about your environment. To keep these docs universal, here are the standard values we will be using:

- Data Directory - <your_data_path>/share/etlTest/
- Log Directory - <your_log_path>/etlTest/log/

Let's check out the data directory and see what got added there:

```
cd <your_data_path>/share/etlTest/
```

There should be three configuration files there:

- connections.cfg - Used for data sources and targets
- properties.cfg - Used for tool configuration settings

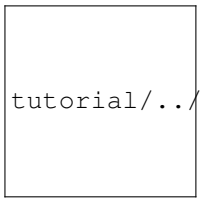
- tools.yml - Used for data integration tool settings

Those files are covered in more detail in the *Configuration Settings* section. Please feel free to make modifications as necessary. Any references to Locations or directory paths will be based on the defaults found in the settings files.

If we also check out the log directory, we shouldn't actually see any files there.

6.2 Writing Your First Test

Now that your environment is set up, it's time to write some tests! Before we do, let's take a look at our sample data integration code - user_dim_load_tr.ktr:



In this transformation, we have a Table Input that is pulling data from our users table (created as part of the *Sample Database Script*), uses a String operations step to lower case the first_name field, and then insert the data using an Insert/Update step into the user_dim table.

6.2.1 What to test for?

So there are many different things that can be tested here:

- How many records do we have in the source? How many are in the target?
- Does the data integration code actually lower case the first_name field?
- Does the data integration code inadvertently modify other data than the first_name field?

Take a few minutes as see if there are other things that could be tested to add to the list. Okay, got a good list? Awesome! Tests will usually fall within one of three categories:

- Positive testing - first_name is lower case
- Negative testing - first_name is not upper case
- Edge cases - first_name with this specific last name is lower case

Each type of test can cause different kinds of issues while writing data integration jobs. Positive testing is the most common type of testing in that it is the 'expected results'. Testing for negatives - things that may have been seen but have since been resolved - is the next common classification of tests. Edge cases are usually the hardest types of tests to cover, especially without the ability to control your test data set.

For this tutorial, we will be creating three tests for our small data integration job.

6.3 Creating A Sample Data Set

Now that we have written our three tests, it's time to create a data set so that we can accurately test them. Remember, we have three tests that will require data:

- Does first name get lower cased?
- Does an upper case first name not return as upper case in the target table?

- Does the birthday field get impacted by the data integration code?

First, let's create a new folder in our data directory (default is `${ETL_TEST_ROOT}/Documents/etlTest/data`).:

```
cd ${ETL_TEST_ROOT}/Documents/etlTest/data
mkdir etlUnitTest
```

We created the `etlUnitTest` directory because that is the source where the data set we're about to create lives. Since the `users` table is the source for our data integration, we should create a new YAML file called users.yml .:

```
touch etlUnitTest/users.yml
vi etlUnitTest/users.yml
```

YAML (which stands for YAML Ain't a Markup Language) was designed to provide some of the same capabilities of XML without the verboseness. To find out more about YAML, head over to The Official YAML Website .

Now let's actually build our data set. Remember, we need a data set that will meet the requirements for our tests. For our first record, let's include a standard, run of the mill users table record.:

```
1:
# Generic record from the users table.
  user_id: 1
  first_name:  Bob
  last_name:  Richards
  birthday:  2000-01-04
  zipcode:  55555
  is_active: 0
```

Notice, the record is identified uniquely with `1` and that all the fields for record one are indented two spaces to indicate they are all together. To give a value to a field, we just put a colon followed by a space and then the value we need for it. i.e. `column_name:  column_value`.

The record we just created will work fine for our first test case, but what do we do for the next one? We could copy the record and change the first_name field to `BOB`, but that could run the risk of test collision when our test suites and data sets get larger. Let's build a new record specific to this test:

```
1:
# Generic record from the users table.
  user_id: 1
  first_name:  Bob
  last_name:  Richards
  birthday:  2000-01-04
  zipcode:  55555
  is_active: 0
2:
# Record for first_name all upper case.
  user_id: 2
  first_name:  Sarah
  last_name: Jenkins
  birthday:  2000-02-02
  zipcode:  12345
  is_active: 1
```

We indicate a new record in the YAML file by removing any indentation in the next line after the zipcode column for record one and give our record another unique identifier (this time `2`). We use the same column names as before, but we now have a record that has an entirely upper-cased first_name field.

For the third test case, we could create a new record or we can utilize one of the existing records to test if the birthday field is manipulated. For the birthday test, we will use record one. Now we can work on building our tests.

6.4 Configuring Your Data Integration Tool

6.5 Generating Test Code

6.6 Executing Your Tests

6.7 Sample MySQL Database

Here is the sample MySQL database script used for the tutorial and for etlTest's own unit tests:

```
CREATE SCHEMA IF NOT EXISTS etlUnitTest;

USE etlUnitTest;

DROP TABLE IF EXISTS users;
DROP TABLE IF EXISTS user_dim;

CREATE TABLE users (
                user_id INT NOT NULL,
                first_name VARCHAR(75) NOT NULL,
                last_name VARCHAR(75) NOT NULL,
                birthday DATE NOT NULL,
                zipcode CHAR(5) NOT NULL,
                is_active TINYINT(1) NOT NULL DEFAULT 1,
                PRIMARY KEY (user_id)
);


CREATE UNIQUE INDEX users_idx
 ON users
 ( user_id );

CREATE TABLE user_dim (
                user_id INT NOT NULL,
                first_name VARCHAR(75) NOT NULL,
                last_name VARCHAR(75) NOT NULL,
                birthday DATE NOT NULL,
                zipcode CHAR(5) NOT NULL,
                PRIMARY KEY (user_id)
);

CREATE UNIQUE INDEX users_idx
 ON user_dim
 ( user_id );
```