lwetl Documentation

Release 0.7.2.master-91863c7948494cc8393462a03724d8042e121d90

R.J. Bakker

Jan 28, 2023

Contents:

CHAPTER 1

Introduction

The module *lwetl* is a light-weight database client to transfer data between various databases, or inbetween tables of the same database.

It is intended as a administrative-, or development-tool to script quick modifications to an existing database. It uses python 3 in combination with the JayDeBeApi module and JDBC jar files.

1.1 Typical usage

- extract data from a database, either through a command-line interface, or through python classes.
- upload table rows into a target database.
- transfer of (modified) data to files or pipes in common formats such as: text, csv, xml, xlsx, or sql.
- extract or upload binary data (not supported by all JDBC drivers).

1.2 Key features

- **A centralized configuration file for database connections:**
  - choice of the JDBC driver.
  - definitions of the JDBC connection URLs.
  - optionally parsing of ORACLE's *tnsnames.ora* for access through JDBC thin client.
  - optionally saving database access credentials as an alias. The password in these credentials may be encrypted with a master password.
- Direct command-line access to a database for upload and download.
- Command-line transfer of tables between independent database instances, possibly of a different server-type.
- Python classes for encapsulated transfer of data.

Due to its nature, the tool is suited for small- or medium-sized transformation-, import-, or extraction-tasks (a throughput rate up to 4000 records per second).

Multi-threading of the database connection is not supported.

1.3 Requirements

- A python 3 environment with permission to install modules (system-wide or as virtual environment).
- A Java 1.7+ runtime environment
- Write-access to the user home-directory (a configuration directory $HOME/.lwetl is auto-created).

1.4 Status

This project is in a pre-pre-alpha stage. It has been tested with drivers for mysql, sqlserver, oracle, postgresql, and sqlite:

- Linux Mint (Ubuntu, Debian) with python 3.5.
- CentOS Linux (Redhat, Fedora) with python 3.4.
- Windows 10 Home with Anaconda python 3.6.

Warning: some database serves (e.g., MySQL) may make a distincion between upper-case and lower-case table-names and/or column-names. This might cause errors, since all current tests have been performed in environments where such a distinction does not exist.

CHAPTER 2

Installation

The module depends on Jpype1 and optionally regex. Both need access to a compiler for installation, if installed with `pip`.

Note: the regex module is used to parse the ORACLE connection configuration (`tnsnames.ora`). If you do not intend to access ORACLE through the settings of this file, the module may be ignored.

2.1 Operating Systems

2.1.1 Linux

The module may in installed in a python virtual environment, for example like:

```
virtualenv --no-site-packages -p /usr/bin/python3 $HOME/my_virtual_envs/jdbc
source $HOME/my_virtual_envs/jdbc/bin/activate
```

The module can be installed with `pip` from github:

```
pip install git+https://github.com/rene-bakker-it/lwetl.git
```

Alternatively the repository may first be cloned:

```
git clone https://github.com/rene-bakker-it/lwetl.git
cd lwetl
pip install .
```

2.1.2 Windows

Note 1: The module depends on Java. Make sure that the JVM and Python are both of the same type, eiter 32 bits, or 64 bits. Only the 64 bits version has been tested:

Note 2: the lwetl package depends on the module cryptography, which depends on openSSL.

```
pip install git+https://github.com/rene-bakker-it/lwetl.git
pip install regex
```

2.2 Dependencies

The module depends on the following packages:

- et-xmlfile,
- JayDeBeApi,
- jdcal,
- Jpype1,
- openpyxl,
- psutil,
- PyYAML,
- cryptography, and
- regex (optionally).

Tests in the `tests` directory are based on pytest, whichalso requires: pytest-html, pytest-metadata, and pytest-progress.

Documentation in the `docs` directory is based on Sphinx and the read the docs theme.

Developers, who want to use the utility function `set-version.py` in the main directectory of the source code, should also install GitPython.

CHAPTER 3

Examples and use

3.1 General

Make sure the java JRE (or JDK) are known to the system. If this is not the case, add `JAVA_HOME` to the system environemnt, or specify it in the `env` section of hte configuration file (see below).

If successful the command *sql-query list* should run without errors. You may see messages like:

```
INFO: ojdbc6-12.1.0.1-atlassian-hosted.jar downloaded to: ./jdbc/lib/python3.4/site-
↪packages/lib
INFO: postgresql-9.4.1208-jdbc42-atlassian-hosted.jar downloaded to: ./jdbc/lib/
↪python3.4/site-packages/lib
INFO: sqlite-jdbc-3.21.0.jar downloaded to: ./jdbc/lib/python3.4/site-packages/lib
INFO: mysql-connector-java-5.1.39.jar downloaded to: ./jdbc/lib/python3.4/site-
↪packages/lib
INFO: mssql-jdbc-6.3.5.jre8-preview.jar downloaded to: ./jdbc/lib/python3.4/site-
↪packages/lib
```

These are downloads, which typically take place once only. The origin of these file may be found in:

```
$HOME/.lwetl/config-example.yml
```

To connect to a database, server definitions must be added to the YAML file `$HOME/.lwetl/config.yml`, see the file `$HOME/.lwetl/config-example.yml` for some examples.

3.2 Invocation from the command-line

A correctly configured connection may then be used like:

```
sql-query <username/password@server> "SQL statement"
```

or with a configured alias:

```
sql-query <alias> "SQL statement"
```

Implemented command line options are (use the -h option for help):

lwetl-security to encrypt/decrypt the passwords in the alias with a master password.

sql-query as a general purpose command-line sql parser, up-loader, or down-loader.

table-cardinality to dump cardinality data of a table into an xlsx spreadsheet.

db-copy to copy entire tables between database instances.

Alternatively they may be invoced as a module, for example:

```
python -m lwetl.programs.sql_query.main
```

3.3 Invocation inside python

The repository directory `examples` is intended for a collection of simple example scripts. The module directories `lwetl/programs` provides more advanced examples.

An example to dump binary images from the database into the current directory:

```
from lwetl import Jdbc

jdbc = Jdbc('login alias')
for fname, img in jdbc.query("SELECT file_name, image_field FROM MY_TABLE"):
    with open(fname,'wb') as f:
        f.write(img)
```

CHAPTER 4

Module components

4.1 Index

4.1.1 The configuration file (config.yml)

Locations

The configuration-file (`config.yml`) may be stored at the following locations:

1. in the root of the module (always present),
2. for Linux systems in the system directory `/etc/lwetl/config.yml`,
3. in the user-home directory `$HOME/.lwetl/config.yml` (auto-created),
4. in the current directory

Upon invocation the program scans the locations in the order given above, and identical definitions are successively overwritten.

Format

The configuration must in yaml markup format and may contain any of the following sections:

env: a section to specify or change system environment variables

drivers: to specify the used jdbc drivers and their configuration

servers: to specify database servers and the database schemes (instances)

alias: containing access credentials and references to database servers, which were specified in the `servers` section.

encrypt: (true|false) to specify if the passwords in the alias should be encrypted with a master password.

Note: access credentials in the alias section may stored in plain text. If security is an issue, you have the following options:

- make sure that the configuration file is properly read-protected.
- use the lwetl-security program to encrypt the passwords with a master password. By default the master password is asked each time you open a database connection. As an alternative it may may be stored in the environment variable LWETL (less secuure).
- If you do not create aliases the username and password must be entered in the appropriate methods.

Encryption with a master password

```
# (re) encrypt the home configuration file with a password

# 1. make sure the master password has to be entered on the command line:
unset LWETL

# 2. (re) encrypt
# the current and new password will be asked on the command line
lwetl-security -c ~/.lwetl/config.yml

# remove encryption
lwetl-security -r ~/.lwetl/config.yml
```

Example

```
# user defined environment variables
env:
    ORACLE_HOME:   /usr/lib/oracle/12.1/client64
    TNS:           /usr/lib/oracle/12.1/client64/network/admin/tnsnames.ora

# jdbc drivers identified by a type (used as type in the next section)
#
# required parameters:
# - jar:    url to download the jdbc jar file if not found on the module library
#           may either be an url or a fixed reference to a file on the file-system
# - class:  name of the class to use in the jar file
# - url:    start of the connection url, will be extended with the url defined in the␣
↪section 'servers'
#
# optional parameters:
# - attr:   additional attributes to add at the end of the generated connection url
# - escape: boolean - if set to true, all column names will be escaped in the␣
↪uploader routines. Permits the
#           use of reserved words as column names.
#           WARNING: not implemented for postgresql
#
# WARNING: the strings used to define the driver types below are also used in the␣
↪python code and should not be changed.
drivers:
    sqlserver:
        # binary upload for jtds driver not supported
        jar:   'http://central.maven.org/maven2/com/microsoft/sqlserver/mssql-jdbc/6.
↪3.5.jre8-preview/mssql-jdbc-6.3.5.jre8-preview.jar'
        class: 'com.microsoft.sqlserver.jdbc.SQLServerDriver'
        url:   'jdbc:sqlserver://'
```

(continues on next page)

(continued from previous page)

```
    mysql:
        jar:   'http://central.maven.org/maven2/mysql/mysql-connector-java/5.1.39/
↪mysql-connector-java-5.1.39.jar'
        class: 'com.mysql.jdbc.Driver'
        url:   'jdbc:mysql://'
        attr:  '?autoReconnect=true&useSSL=false'
        escape: true

    oracle:
        jar:   'https://maven.atlassian.com/3rdparty/com/oracle/ojdbc6/12.1.0.1-
↪atlassian-hosted/ojdbc6-12.1.0.1-atlassian-hosted.jar'
        class: 'oracle.jdbc.OracleDriver'
        url:   'jdbc:oracle:thin:@'

    postgresql:
        # jar:   'http://central.maven.org/maven2/org/postgresql/postgresql/42.1.4.
↪jre7/postgresql-42.1.4.jre7.jar'
        jar:   'https://maven.atlassian.com/3rdparty/postgresql/postgresql/9.4.1208-
↪jdbc42-atlassian-hosted/postgresql-9.4.1208-jdbc42-atlassian-hosted.jar'
        class: 'org.postgresql.Driver'
        url:   'jdbc:postgresql://'

# servers
# defines database servers on the schema (instance) level
#
# required parameters:
# - type - must be one of the types defined in drivers
# - url  - connection url. The complete url is <url_driver><url server><attr driver>
#
# NOTE: for ORACLE additional server names may be obtained from the file tnsnames.ora
servers:
    scott_mysql:
        type: mysql
        url:  "192.168.7.33:3306/scott"
    scott_postgresql:
        type: postgresql
        url:  "172.56.11.41:5432/scott"
    scott_sqlserver:
        type: sqlserver
        url:  '172.56.11.41\scott:1534'

# alias for connections, in ORACLE credentials format
# <username>/<password>@<servername>
encrypted: false
alias:
    scott_mysql:      "scott/tiger@scott_mysql"
    scott_postgresql: "scott/tiger@scott_postgresql"
    scott_sqlserver:  "scott/tiger@scott_sqlserver"
    scott_oracle:     "scott/tiger@scott_oracle"
    scot:             "scot/xxxxxxxx@tns_entry"
```

Sections

env - environment

Function:

- specify the jre/jdk for the jdbc drivers
- specify the location of ORACLE configurations

By default this section is empty.

Example

```
env:
    # Windows
    JAVA_HOME:      'C:\Progra~1\Java\jre1.8.0_65'
    ORACLE_HOME:    'C:\Oracle\product\11.2.0'
    # Linux
    TNS:            /usr/lib/oracle/12.1/client64/network/admin/tnsnames.ora
```

Note 1: if only ORACLE_HOME is specified, the program will search for the file `$ORACLE_HOME/network/admin/tnsnames.ora`. If also TNS is specified, the program will first look at the location specified by `$TNS`. Only if this section is not found, it will look at the previous location.

Note 2: On Windows 64-bit systems:

```
Progra~1 = 'Program Files'
Progra~2 = 'Program Files(x86)'
```

drivers - Jdbc driver definitions

Function - associate a unique tag to a database server type:

- specify a source location of a jdbc jar file (url or file)
- specify the jdbc driver class of the jar file
- specify the base of the connection url

servers - Database server definitions

Function - associate a unique tag to a database connection:

- the driver driver used (see previous section)
- main connection URL specifying: - the IP address of the database server - the scheme/instance of the database

alias - Connection aliases

Function:

- specify the jre/jdk for the jdbc drivers

4.1.2 The main connection

The class `Jdbc` creates a connection to a database, which remains open until the object isdestroyed.

class Jdbc(*login*, *auto_commit=False*, *upper_case=True*)

Creates a connection. `Raises` an exception if the connection fails, see the example below.

Parameters

- **login** (*str*) – login alias defined in `config.yml`, or authentication credentials like:

  username/password@service

  The parser assumes that the name of the service does not contain the '@' character. The password should not contain a '/'.

- **auto_commit bool** (*bool*) – specifies the auto-commit mode of the connection at startup. Defaults to False (auto-commit disabled).
- **upper_case** (*bool*) – specifies if the column names of SQL queries are converted into upper-case. Convenient if the result of queries is converted into dictionaries.

Example:

```
from lwetl import Jdbc, ServiceNotFoundException, DriverNotFoundException

# create a connection to 'scott' with password 'tiger' to oracle server 'osrv1'
# (as defined in tnsnames.ora)

try:
    jdbc = Jdbc('scott/tiger@osrv01')
except (ServiceNotFoundException, DriverNotFoundException) as nfe:
    print('ERROR - could initialize: ' + str(nfe))
except ConnectionError as ce:
    print('ERROR - failed to connect: ' + str(ce))
```

connection

The connection to the database as returned by `jaydebeapi.connect`. See PEP249 for further details.

execute(*sql: str*, *parameters: (list, tuple) = None*, *cursor: object = None*) → Cursor

Execute a query, optionally with list of parameters, or a list of a list of parameters. `Raises` an *SQLExecutionException* if the query fails, see the example below.

Parameters

- **sql** (*str*) – query to execute
- **parameters** (*tuple, list, None*) – parameters specified in the sql query. May also be None (no parameters), or a list of lists
- **cursor** (*Cursor, None*) – the cursor to use for exection. Create a new one if None (default), or if the cursor is not an open cursor associated to the current connection

Returns a `jaydebeapi.connect.Cursor` for further processing.

Example:

```
from lwetl import Jdbc, SqlExecutionException

jdbc = Jdbc('scott/tiger@osrv01')

try:
```

(continues on next page)

(continued from previous page)

```
    cur = jdbc.execte("INSERT INTO TST_NAMES (ID, USERNAME) VALUES (17,'scott
↪')")
except SQLExectionException as sqle:
    print('ERROR - could not execute: ' + str(sqle))
```

close(cursor=None):
Closes the specified cursor. Use the current if not specified. Cursors which are aleady closed, or are not associated to the jdbc conection are silently ignored.

get_columns(*cursor=None*) → OrderedDict:

Parameters **cursor** (*Cursor*) – the cursor to query. Uses the last used (current) cursor, if not specified.

Returns the column associated to the cursor as an OrderedDict, or an empty dictionary if no columns were found.

commit(cursor=None):
Commits pending modifications of the specified cursor to the database. If not specified, the current corsor is assumed.

Parameters **cursor** (*Cursor*) – the cursor to query. Uses the last used (current) cursor, if not specified.

Warning: This may also commit pending modifications of other cursors associated to the connection.

rollback(cursor=None):
Rolls back pending modifications of the specified cursor to the database. If not specified, the current corsor is assumed.

Parameters **cursor** (*Cursor*) – the cursor to query. Uses the last used (current) cursor, if not specified.

Warning: This may also commit pending modifications of other cursors associated to the connection.

get_data(*cursor: Cursor = None*, *return_type=tuple*, *include_none=False*, *max_rows: int = 0*, *array_size: int = 1000*) → iterator:
Get the data retrieved from a *execute()* command.

Parameters

- **cursor** (*Cursor*) – cursor to query, use current if not specified
- **return_type** (*Any*) – the return type of the method. Defaults to `tuple`. Other valid options are `list`, `dict`, `OrderedDict`, or a (tuple of) stings. In case of the latter, the output is casted to the specified types. Supported types are `Any` (no casting), `str`, `int`, `bool`, `float`, `date`, or a format string compatible with :class:'datetime.strptime()'. The format string for 'date' is '%Y-%m-%d [%H:%M:%S]'. If a single string is specified, the returned row will only be the first value of each row. Otherwise the output is a tuple of values with a maximum length of the specified input tuple. This option is particularly useful for connections to a sqlite, where the auto-casting casting of the types in the jdbc driver may fail.
- **include_none** (*bool*) – if set to `True`, also returns `None` values in dictionaries. Defaults to `False`. For `tuple`, or `list`, all elements are always returned.

- **max_rows** (*int*) – maximum number of rows to return before closing the cursor. Negative or zero implies all rows
- **array_size** (*int*) – the buffer size to retrieve batches of data.

Returns an iterator with rows of data obtained from an SQL with the data-type specified with the *return_type* parameter.

query (*sql: str*, *parameters=None*, *return_type=tuple*, *max_rows=0*, *array_size=1000*) → iterator:
Combines the *execute()* and *get_data()* into a single statement.

query_single (*sql: str*, *parameters=None*, *return_type=tuple*) → (tuple, list, dict, OrderedDict):

Returns only the first row from *query()*

query_single_value(sql: str, parameters=None):

Returns the first column from *query_single()*

get_int(sql: str, parameters=None):
A short-cut for:

```
int(query_single_value(sql, parameters))
```

Exceptions

class SQLExcecuteException

Raised when an `execute()` command cannot be parsed.

class ServiceNotFoundException

Raised when a database connection cannot be reach the database server.

class DriverNotFoundException

Raised when the jdbc driver, associated to the database connection, cannot be retrieved.

class CommitException

Raised when a `commit()` command fails.

Utility functions and classes

class JdbcInfo (*login: str*)
Displays parameter information of the jdbc driver.

Parameters **login** (*str*) – login alias defined in `config.yml`, or authentication credentials.

Example:

```
from lwetl JdbcInfo

jdbc_info = JdbcInfo('scott')
jdbc_info()
```

get_execution_statistics () → str
Retrieves some timing statistics on the established connections.

Return type multi-line string

tag_connection(*tag:str*, *jdbc:Jdbc*)
Marks specific connections, such that the function *get_execution_statistics()* provides more detail.

Parameters

- **tag** (*str*) – a tag for a connection
- **jdbc** (Jdbc) – an established database connection

Example:

```
from import Jdbc, get_execution_statistics, tag_connection

jdbc = {
    'SRC': Jdbc('scott_source'),
    'TRG': Jdbc('scott_target')
}
for tag, con in jdbc.items():
    tag_connection(tag,con)

# do lots of work

print(get_execution_statistics())
```

4.1.3 Import into a database

Import comes either as a sql import reader or as objects to read tablels of data.

SQL Import

InputParser(sql_or_filename_or_stream=None, sql_terminator:str=';'):
Class to parse SQL input, either from file or from a stream (e.g., stdin) Assumes that all SQLs are terminated with an sql_terminator character (defaults to a semi-colon) at the end of a line.

> **Warning:** This class may fail on multi-line string inputs that use the same character and the and of a CRLF.

Example:

```
import sys

from lwetl Jdbc, InputParser

jdbc = Jdbc('scott/tiger@osrv01')
with InputParser(sys.stdin) as parser:
    for sql in parser.parse():
        jdbc.execute(sql)
```

set_sql_terminator(sql_terminator):
Specifies the SQL terminator.

Parameters **sql_terminator** (*str*) – The specified SQL terminator

open(sql_fn_stream=None):
Opens a TextIOWrapper for input

Parameters **sql_fn_stream** (*TextIOWrapper*) – the stream to open

close():
Closes the input stream

parse (*array_size=1*) → iterator:
parses the input stream.

Parameters **array_size** (*int*) – buffer size of the iterator

Returns an iterator of SQL commands

Data Readers

class CsvImport (*filename_or_stream=None*, *delimiter="t"*, *encoding='utf-8'*)
Open a CSV file and extract data by row in the form of a dictionary Expects the first row if the dictionary to contain the column names.

Parameters

- **filename_or_stream** – if the argument is a string, the program will try to open the file with this name. For streams, it will use the stream as-is. Defaults to the stdin.
- **delimiter** (*str*) – specifies the column delimiter of the CSV file. Defaulst to the TAB character.
- **encoding** (*str*) – character encoding to use for the input file. Defaults to utf-8

open (*filename_or_stream=None*, *delimiter=None*, *encoding=None*)

Parameters

- **filename_or_stream** – specifies the input. If not specified, it takes the specifier when the class object was created.
- **delimiter** (*str*) – specifies the column deliter. If not specified, it takes the delimiter specified when the object was created.
- **encoding** (*str*) – specifies the character encoding. If not specified, it takes the encoding specified when the object was created.

close():
closes the input stream. Only has an effect, if the input was specified as a filename.

get_data (*max_rows=1000*) → iterator

Parameters **max_rows** (*int*) – retrieve the data as an generator/iterator. The parameter specifies the buffer size.

class LdifImport (*filename_or_stream=None*, *separator=None*, *encoding='utf-8'*)
Open a LDIF file and extract data as a dictionary with the attribute names as keys.

Parameters

- **filename_or_stream** – if the argument is a string, the program will try to open the file with this name. For streams, it will use the stream as-is. Defaults to the stdin.
- **separator** (*str*) – in an ldif file, an attibure may occur multiple times in the same record. In such cases the value of the dictionary becomes a list. In the case the separator is specified, this list is transformed into a string, separating the elements with the specified separator.
- **encoding** (*str*) – character encoding to use for the input file. Defaults to utf-8

open (*filename_or_stream=None*, *separator=None*, *encoding=None*)
opens the file or stream of input.

Parameters

- **filename_or_stream** – specifies the input. If not specified, it takes the specifier when the class object was created.
- **separator** (*str*) – in an ldif file, an attibure may occur multiple times in the same record. In such cases the value of the dictionary becomes a list. In the case the separator is specified, this list is transformed into a string, separating the elements with the specified separator.
- **encoding** (*str*) – specifies the character encoding. If not specified, it takes the encoding specified when the object was created.

close():
closes the input stream. Only has an effect, if the input was specified as a filename.

get_data() → iterator
retrieve the data as an generator/iterator. The parameter specifies the buffer size.

class XlsxImport(*self*, *file_name: str*, *sheet_name: str = None*)
Open an xls worksheet and extract the data by row in the form of a dictionary Expects the first row of the worksheet to contain the column names

open(file_name: str = None, sheet_name: str = None):

close()

get_data(*max_rows=1000*)

Examples

Import from the `stdin` in CSV format and upload in native query format (see next section).

```
import sys

from lwetl import Jdbc, CsvImport, NativeUploader

jdbc = Jdbc('scott')

with NativeUploader(jdbc,'TARGET_TABLE', commit_mode=lwetl.UPLOAD_MODE_COMMIT) as upl:
    # read CSV from stdin
    with CsvImport(sys.stdin) as csv:
        for r in csv.get_data():
            upl.insert(r)
```

Import from an excel 2007+ spreadsheet and upload using parameterized SQL syntax (see next section).

```
import sys

from lwetl import Jdbc, XlsxImport, ParameterUploader

jdbc = Jdbc('scott')

table = 'TARGET_TABLE'
# alternative to with statement
xls = XlsxImport()
xls.open(table + '.xlsx')
with ParameterUploader(jdbc,table, commit_mode=lwetl.UPLOAD_MODE_COMMIT) as upl:
    for r in xls.get_data():
```

(continues on next page)

(continued from previous page)

```
        upl.insert(r)
        if upl.rowcount > 1000:
            upl.commit()
    if upl.rowcount > 0:
        upl.commit()
xls.close()
```

Upload models

Operational modes

Import into a database has the following modes of operation:

UPLOAD_MODE_DRYRUN SQL statements are generated, but not send to the database.

UPLOAD_MODE_PIPE SQL statements are generated and piped for futher processing. The database itself is not touched.

UPLOAD_MODE_ROLLBACK SQL statements are generated and executed to the database. However, the commit statement performs a rollback instead.

> **Warning:** This mode is not compatible with a database connection in auto-commit mode. It will also fail if the user sends commit commands independently.

UPLOAD_MODE_COMMIT SQL statements are generated and executed to the database. However, the commit statement performs a rollback instead.

Classes

class NativeUploader(*jdbc: Jdbc*, *table: str*, *fstream=None*, *commit_mode=UPLOAD_MODE_DRYRUN*, *exit_on_fail=True*)

Upload data into a table with native SQL (no parameters in the jdbc execute command).

Parameters

- **jdbc** (*Jdbc*) – The target database connection
- **table** (*str*) – Name of the table in the database to insert the data
- **fstream** –
- **commit_mode** (*str*) – The upload mode, see *Operational modes*.
- **exit_on_fail** (*bool*) – Clear the commit buffer and exit if an insert, update, or delete command fails.

insert(data: dict):

Insert into the table

Parameters **data** (*dict*) – a dictionary of key (column name) and values. Keys, which do not correspond to an existing column names are ignored.

update(data: dict, where_clause):

Update an existing row in the table

Parameters

- **data** (*dict*) – a dictionary of key (column name) and values. Keys, which do not correspond to an existing column names are ignored.
- **where_clause** (*None, str, dict*) – filter for column selection. Valid formats for the where clause are:

  None updates all columns.

  str raw SQL WHERE clause (the keyword WHERE may be omitted).

  dict keys are column names. Non exisiting column names are ignored. Multiple columns are combined with the AND statement. The value may be:

  - a value (results in COLUMN_NAME = VALUE)
  - a string with an operator and value, e.g., `LIKE 'ABC%'`
  - a tuple (operator,value), e.g., `('>=', 7)`

delete(where_clause):
Delete rows in the table

Parameters **where_clause** (*None, str, dict*) – filter for the columns to delete. Formats are identical to the `update` statement.

commit()
Processes previous insert/update/delete statements depending on the *Operational modes* of the instance.

UPLOAD_MODE_COMMIT sends a commit statement to the database

UPLOAD_MODE_ROLLBACK sends a rollback statement to the database

UPLOAD_MODE_DRYRUN does nothing

UPLOAD_MODE_PIPE work in progress

> **Warning:** This mode is not compatible with a database connection in auto-commit mode. It will also fail if the user sends commit commands independently.

add_counter(columns: (str, list, set, tuple)):
Mark columns as counters. Assumes the column type is a number. Queries the maximum number of each column and then adds the next value (+1) in the column on each insert.

Parameters **columns** (*str, list, set, tuple*) – names of the columns to add. May be a (comma-separated) string, or a list type.

class ParameterUploader(*self*, *jdbc: Jdbc*, *table: str*, *fstream=None*, *commit_mode=UPLOAD_MODE_DRYRUN*, *exit_on_fail=True*)
Upload data into a table using parameterized SQL commands. See the section *NativeUploader* for details on the command line arguments.

insert(data: dict):
Insert into the table, see the *NativeUploader* for details.

update(data: dict, where_clause):
Update an existing row in the table, see the *NativeUploader* for details.

delete(where_clause):
Delete existing rows from the table, see the *NativeUploader* for details.

commit()
Processes previous insert/update/delete statements depending on the *Operational modes* of the instance. See the *NativeUploader* for details

add_counter(columns: (str, list, set, tuple)):
Mark columns as counters. Assumes the column type is a number. Queries the maximum number of each column and then adds the next value (+1) in the column on each insert. See the *NativeUploader* for details

class MultiParameterUploader(*jdbc: Jdbc*, *table: str*, *fstream=None*, *commit_mode=UPLOAD_MODE_DRYRUN*, *exit_on_fail=True*)
Upload data into a table using the jdbc executemany parameterized command.

insert(data: dict):
Insert into the table, see the *NativeUploader* for details.

commit()
Processes previous insert/update/delete statements depending on the *Operational modes* of the instance. See the *NativeUploader* for details

add_counter(columns: (str, list, set, tuple)):
Mark columns as counters. Assumes the column type is a number. Queries the maximum number of each column and then adds the next value (+1) in the column on each insert. See the *NativeUploader* for details

4.1.4 Export from a database

Formatters are intended to output table data. Supported formatters are:

TextFormatter Outputs tables in plain text format with fixed-width columns.

CvsFormatter Outputs tables in CSV format.

XmlFormatter Outputs tables in XML format

XlsxFormatter Outputs tables in EXCEL xlsx format

All formatters may be used in the following ways:

Example 1: function call

```
from lwetl import Jdbc, TextFormatter

jdbc = Jdbc('scott')

sql = 'SELECT * FROM MY_TABLE ORDER BY ID'
fmt = TextFormatter()
fmt(jdbc=jdbc, sql=sql)
```

Example 2: with statement

```
from lwetl import Jdbc, TextFormatter

jdbc = Jdbc('scott')

cur = jdbc.execute('SELECT * FROM MY_TABLE ORDER BY ID',cursor = None)

formatter = TextFormatter()
with TextFormatter(cursor=cur) as fmt:
    fmt.header()
    for row in jdbc.get_data(cur):
        fmt.write(row)
    fmt.footer()
```

Example 3: open/close

```
from lwetl import Jdbc, TextFormatter

jdbc = Jdbc('scott')

cur = jdbc.execute('SELECT * FROM MY_TABLE ORDER BY ID',cursor = None)

fmt = TextFormatter()
fmt.open(cursor=cur)
fmt.header()
for row in jdbc.get_data(cur):
    fmt.write(row)
fmt.footer()
fmt.close()
```

Below onl the TextFormatter is described in detail. For the otherones only the differences are mentioned.

TextFormatter():

Outputs a table in plain text format with fixed-width columns.

__init__(*args, **kwargs):

Instantiate. All arguments are optional.

Parameters

- **cursor** (*Cursor*) – cursor generated by `jdbc.execute()`
- **filename_or_stream** (*(str, TextIOWrapper, StringIO)*) – specifier of the output stream. May be a filename (string) or a stream object.
- **append** (*bool*) – append the specified file, rather then creating a new one. Defaults to False.
- **column_width** (*int*) – the width of each text column. (Only used in this class)

```
from lwetl import TextFormatter

fmt = TextFormatter(cursor=cursor, filename_or_stream='myoutput.txt',␣
↪append=True)
```

__call__(*args, **kwargs):

Write a table in a single statement, see Example1 above.

Parameters

- **jdbc** (Jdbc) – The Jdbc connection
- **sql** (*str*) – The SQL to parse

Also accepts all arguments of the `__init__()` statement with the exception of the cursor.

open(**args*, ***kwargs*)

Opens the file or stream for writing. Takes the same arguments as the `__init__()` statement.

close()

Closes the output file or stream (if applicable)

format(*row*)

Format the row of data.

Parameters **row** (*(list, tuple)*) – a row of data.

Returns a string.

header()
Write the header (column names) to the specified file or stream.

write(*row: list*)
Writes the output of `format(row)()` to the specified output stream.

CsvFormatter():
Outputs a table in CSV format. The functionality is identical to the `TextFormatter`

all fuctions

Parameters **separator** (*str*) – Specifies the CSV column separator. Defaults to ';'

XmlFormatter():
Outputs a table in XML format. The functionality is identical to the `TextFormatter`

all fuctions

Parameters

- **dialect** (*str*) – Specifies the XML dialect: 'excel', 'value', or 'plain'. Defaults to 'excel'
- **pretty_print** (*bool*) – Output the xml in formatted mode, instead of compact. Defaults to False.
- **sheet_name** (*str*) – Specifies the name of the worksheet. Defaults to 'Sheet1'

next_sheet(*cursor*, *sheet_name=None*)
Initiates a new sheet with a new cursor.

Parameters

- **cursor** (*Cursor*) – cursor generated by `jdbc.execute()`
- **sheet_name** (*str*) – name of the work sheet. Uses a counter like 'SheetN' if not specified.

Example:

```
from lwetl import Jdbc, XmlFormatter

jdbc = Jdbc('scott')
fmt = None
for table in ['MY_TABLE1', 'MY_TABLE_2']:
    cur = jdbc.execute('SELECT * from {0} ORDER BY ID'.format(table), cursor=None)
    if fmt is None:
        fmt = XmlFormatter()
        fmt.open(cursor='cur')
    else:
        fmt.next_sheet(cur)

    fmt.header()
    for row in jdbc.get_data(cur):
        fmt.write(row)
fmt.close()
```

XlsxFormatter(jdbc: Jdbc):
Outputs a table in a Excel 2007+ file. The functionality is identical to the `XmlFormatter` but `dialect` and `pretty_print` are not supported. Instead, the argument `pretty=True` in the open method, will reformat the xlsx file to autoset the column width and print the header line in bold.

Warning: Stream output is not supported. Only valid file-names are accepted for the argument `filename_or_stream`.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index