pyDAL Documentation

Release 16.8

web2py-developers

August 13, 2016

Contents

Contents:

CHAPTER 1

Indices and tables

- genindex
- modindex
- search

1.1 Subpackages

1.1.1 pydal.adapters package

Submodules

pydal.adapters.base module

class pydal.adapters.base.**BaseAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)

Bases: *pydal.connection.ConnectionPool*

adapt(*value*)

alias(*table*, *alias*)

close_connection(**args*, ***kwargs*)

common_filter(*query*, *tablenames*)

connector()

dbengine = 'None'

drivers = ()

drop_table(*table*, *mode=''*)

expand_all(*fields*, *tablenames*)

find_driver()

get_table(**queries*)

iterparse(*sql*, *fields*, *colnames*, *blob_decode=True*, *cacheable=False*)

Iterator to parse one row at a time. It doen't support the old style virtual fields

parse(*rows*, *fields*, *colnames*, *blob_decode=True*, *cacheable=False*)

parse_value(*value*, *field_itype*, *field_type*, *blob_decode=True*)

represent(*obj*, *field_type*)

rowslice(*rows*, *minimum=0*, *maximum=None*)

support_distributed_transaction = False

tables(**queries*)

test_connection()

types

uploads_in_blob = False

class pydal.adapters.base.**DebugHandler**(*adapter*)
Bases: *pydal.helpers.classes.ExecutionHandler*

before_execute(*command*)

class pydal.adapters.base.**NoSQLAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)
Bases: *pydal.adapters.base.BaseAdapter*

can_select_for_update = False

commit()

commit_prepared(*key*)

create_table(*table*, *migrate=True*, *fake_migrate=False*, *polymodel=None*)

drop(**args*, ***kwargs*)

drop_table(*table*, *mode=''*)

id_query(*table*)

prepare()

rollback()

rollback_prepared(*key*)

sqlsafe_field(*fieldname*)

sqlsafe_table(*tablename*, *original_tablename=None*)

class pydal.adapters.base.**NullAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)
Bases: *pydal.adapters.base.BaseAdapter*

connector()

find_driver()

class pydal.adapters.base.**SQLAdapter**(**args*, ***kwargs*)
Bases: *pydal.adapters.base.BaseAdapter*

adapt(*obj*)

bulk_insert(*table*, *items*)

can_select_for_update = True

commit(**args*, ***kwargs*)

commit_on_alter_table = False

commit_prepared(**args*, ***kwargs*)

count(*query*, *distinct=None*)

create_index(*table*, *index_name*, **fields*, ***kwargs*)

create_sequence_and_triggers(*query*, *table*, ***args*)

create_table(**args*, ***kwargs*)

delete(*tablename*, *query*)

distributed_transaction_begin(*key*)

drop(**args*, ***kwargs*)

drop_index(*table*, *index_name*)

drop_table(*table*, *mode=''*)

execute(**args*, ***kwargs*)

execution_handlers = []

fetchall()

fetchone()

filter_sql_command(*command*)

id_query(*table*)

index_expander(**args*, ***kwds*)

insert(*table*, *fields*)

iterselect(*query*, *fields*, *attributes*)

lastrowid(*table*)

prepare(**args*, ***kwargs*)

represent(*obj*, *field_type*)

rollback(**args*, ***kwargs*)

rollback_prepared(**args*, ***kwargs*)

select(*query*, *fields*, *attributes*)

smart_adapt(*obj*)

sqlsafe_field(*fieldname*)

sqlsafe_table(*tablename*, *original_tablename=None*)

table_alias(*tbl*)

test_connection()

truncate(*table*, *mode=''*)

update(*tablename*, *query*, *fields*)

pydal.adapters.couchdb module

class pydal.adapters.couchdb.**CouchDB**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)

Bases: *pydal.adapters.base.NoSQLAdapter*

connector()

count(*query*, *distinct=None*)

create_table(*table*, *migrate=True*, *fake_migrate=False*, *polymodel=None*)

dbengine = 'couchdb'

delete(*tablename*, *query*)

drivers = ('couchdb',)

insert(*table*, *fields*)

select(*query*, *fields*, *attributes*)

update(*tablename*, *query*, *fields*)

uploads_in_blob = True

pydal.adapters.cubrid module

pydal.adapters.db2 module

class pydal.adapters.db2.**DB2**(**args*, ***kwargs*)

Bases: *pydal.adapters.base.SQLAdapter*

dbengine = 'db2'

execute(**args*, ***kwargs*)

lastrowid(*table*)

rowslice(*rows*, *minimum=0*, *maximum=None*)

class pydal.adapters.db2.**DB2IBM**(**args*, ***kwargs*)

Bases: *pydal.adapters.db2.DB2*

connector()

drivers = ('ibm_db_dbi',)

class pydal.adapters.db2.**DB2Pyodbc**(**args*, ***kwargs*)

Bases: *pydal.adapters.db2.DB2*

connector()

drivers = ('pyodbc',)

pydal.adapters.firebird module

class pydal.adapters.firebird.**FireBird**(**args*, ***kwargs*)

Bases: *pydal.adapters.base.SQLAdapter*

REGEX_URI = <_sre.SRE_Pattern object at 0x1861770>

commit_on_alter_table = True

connector()

create_sequence_and_triggers(*query*, *table*, ***args*)

dbengine = 'firebird'

drivers = ('kinterbasdb', 'firebirdsql', 'fdb', 'pyodbc')

lastrowid(*table*)

support_distributed_transaction = True

class pydal.adapters.firebird.**FireBirdEmbedded**(**args*, ***kwargs*)
Bases: *pydal.adapters.firebird.FireBird*

REGEX_URI = <_sre.SRE_Pattern object at 0x18501f0>

pydal.adapters.google_adapters module

Adapter for GAE

pydal.adapters.imap module

pydal.adapters.informix module

class pydal.adapters.informix.**Informix**(**args*, ***kwargs*)
Bases: *pydal.adapters.base.SQLAdapter*

after_connection()

connector()

dbengine = 'informix'

drivers = ('informixdb',)

execute(**args*, ***kwargs*)

lastrowid(*table*)

test_connection()

class pydal.adapters.informix.**InformixSE**(**args*, ***kwargs*)
Bases: *pydal.adapters.informix.Informix*

rowslice(*rows*, *minimum=0*, *maximum=None*)

pydal.adapters.ingres module

class pydal.adapters.ingres.**Ingres**(**args*, ***kwargs*)
Bases: *pydal.adapters.base.SQLAdapter*

connector()

create_sequence_and_triggers(*query*, *table*, ***args*)

dbengine = 'ingres'

drivers = ('pyodbc',)

class pydal.adapters.ingres.**IngresUnicode**(**args*, ***kwargs*)
Bases: *pydal.adapters.ingres.Ingres*

pydal.adapters.mongo module

class pydal.adapters.mongo.**Binary**
Bases: object

class pydal.adapters.mongo.**Expansion**(*adapter*, *crud*, *query*, *fields=()*, *tablename=None*, *groupby=None*, *distinct=False*, *having=None*)
Bases: object

Class to encapsulate a pydal expression and track the parse expansion and its results.

Two different MongoDB mechanisms are targeted here. If the query is sufficiently simple, then simple queries are generated. The bulk of the complexity here is however to support more complex queries that are targeted to the MongoDB Aggregation Pipeline.

This class supports four operations: 'count', 'select', 'update' and 'delete'.

Behavior varies somewhat for each operation type. However building each pipeline stage is shared where the behavior is the same (or similar) for the different operations.

In general an attempt is made to build the query without using the pipeline, and if that fails then the query is rebuilt with the pipeline.

QUERY constructed in _build_pipeline_query(): $project : used to calculate expressions if needed $match: filters out records

FIELDS constructed in _expand_fields():

FIELDS:COUNT $group : filter for distinct if needed $group: count the records remaining

FIELDS:SELECT $group : implement aggregations if needed $project: implement expressions (etc) for select

FIELDS:UPDATE $project: implement expressions (etc) for update

HAVING constructed in _add_having(): $project : used to calculate expressions $match: filters out records $project : used to filter out previous expression fields

annotate_expression(*expression*)

dialect

get_collection(*safe=None*)

class pydal.adapters.mongo.**Mongo**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)
Bases: *pydal.adapters.base.NoSQLAdapter*

after_connection()

bulk_insert(*table*, *items*)

check_notnull(*table*, *values*)

check_unique(*table*, *values*)

connector()

count(*query*, *distinct=None*, *snapshot=True*)

dbengine = 'mongodb'

delete(*tablename*, *query*, *safe=None*)

drivers = ('pymongo',)

find_driver()

insert(*table*, *fields*, *safe=None*)
Safe determines whether a asynchronous request is done or a synchronous action is done For safety, we use by default synchronous requests

object_id(*arg=None*)
Convert input to a valid Mongodb ObjectId instance

self.object_id("<random>") -> ObjectId (not unique) instance

represent(*obj*, *field_type*)

select(*query*, *fields*, *attributes*, *snapshot=False*)

truncate(*table*, *mode*, *safe=None*)

update(*tablename*, *query*, *fields*, *safe=None*)

class pydal.adapters.mongo.**MongoBlob**
Bases: *pydal.adapters.mongo.Binary*

MONGO_BLOB_BYTES = 0

MONGO_BLOB_NON_UTF8_STR = 1

static decode(*value*)

pydal.adapters.mssql module

class pydal.adapters.mssql.**MSSQL**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)
Bases: *pydal.adapters.base.SQLAdapter*

REGEX_ARGPATTERN = <_sre.SRE_Pattern object>

REGEX_DSN = <_sre.SRE_Pattern object>

REGEX_URI = <_sre.SRE_Pattern object at 0x1847ac0>

connector()

dbengine = 'mssql'

drivers = ('pyodbc',)

lastrowid(*table*)

class pydal.adapters.mssql.**MSSQL1**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)
Bases: *pydal.adapters.mssql.MSSQL*, *pydal.adapters.mssql.Slicer*

class pydal.adapters.mssql.**MSSQL1N**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)
Bases: *pydal.adapters.mssql.MSSQLN*, *pydal.adapters.mssql.Slicer*

class pydal.adapters.mssql.**MSSQL3**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)
Bases: *pydal.adapters.mssql.MSSQL*

class pydal.adapters.mssql.**MSSQL3N**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)
Bases: *pydal.adapters.mssql.MSSQLN*

class pydal.adapters.mssql.**MSSQL4**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)
Bases: *pydal.adapters.mssql.MSSQL*

class pydal.adapters.mssql.**MSSQL4N**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)
Bases: *pydal.adapters.mssql.MSSQLN*

class pydal.adapters.mssql.**MSSQLN**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)
Bases: *pydal.adapters.mssql.MSSQL*

execute(**args*, ***kwargs*)

represent(*obj*, *field_type*)

class pydal.adapters.mssql.**Slicer**
Bases: object

rowslice(*rows*, *minimum=0*, *maximum=None*)

class pydal.adapters.mssql.**Sybase**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)
Bases: *pydal.adapters.mssql.MSSQL1*

connector()

dbengine = 'sybase'

class pydal.adapters.mssql.**Vertica**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)
Bases: *pydal.adapters.mssql.MSSQL1*

lastrowid(*table*)

pydal.adapters.mysql module

class pydal.adapters.mysql.**Cubrid**(**args*, ***kwargs*)
Bases: *pydal.adapters.mysql.MySQL*

dbengine = 'cubrid'

drivers = ('cubriddb',)

class pydal.adapters.mysql.**MySQL**(**args*, ***kwargs*)
Bases: *pydal.adapters.base.SQLAdapter*

REGEX_URI = <_sre.SRE_Pattern object at 0x1847d80>

after_connection()

commit_on_alter_table = True

commit_prepared(**args*, ***kwargs*)

connector()

dbengine = 'mysql'

distributed_transaction_begin(*key*)

drivers = ('MySQLdb', 'pymysql', 'mysqlconnector')

prepare(**args*, ***kwargs*)

rollback_prepared(**args*, ***kwargs*)

support_distributed_transaction = True

pydal.adapters.oracle module

class pydal.adapters.oracle.**Oracle**(**args*, ***kwargs*)
Bases: *pydal.adapters.base.SQLAdapter*

after_connection()

cmd_fix = <_sre.SRE_Pattern object>

connector()

create_sequence_and_triggers(*query*, *table*, ***args*)

dbengine = 'oracle'

drivers = ('cx_Oracle',)

execute(**args*, ***kwargs*)

fetchall()

insert(*table*, *fields*)

lastrowid(*table*)

sqlsafe_table(*tablename*, *original_tablename=None*)

test_connection()

pydal.adapters.postgres module

class pydal.adapters.postgres.**JDBCPostgre**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)

Bases: *pydal.adapters.postgres.Postgre*

REGEX_URI = <_sre.SRE_Pattern object at 0x1851430>

after_connection()

connector()

drivers = ('zxJDBC',)

class pydal.adapters.postgres.**Postgre**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)

Bases: *pydal.adapters.base.SQLAdapter*

REGEX_URI = <_sre.SRE_Pattern object at 0x18439f0>

after_connection()

commit_prepared(**args*, ***kwargs*)

connector()

dbengine = 'postgres'

drivers = ('psycopg2', 'pg8000')

lastrowid(*table*)

prepare(**args*, ***kwargs*)

rollback_prepared(**args*, ***kwargs*)

support_distributed_transaction = True

class pydal.adapters.postgres.**PostgreBoolean**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)

Bases: *pydal.adapters.postgres.PostgreNew*

class pydal.adapters.postgres.**PostgreMeta**

Bases: *pydal.adapters.AdapterMeta*

class pydal.adapters.postgres.**PostgreNew**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)

Bases: *pydal.adapters.postgres.Postgre*

class pydal.adapters.postgres.**PostgrePG8000**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)

Bases: *pydal.adapters.postgres.Postgre*

adapt(*obj*)

drivers = ('pg8000',)

execute(**args*, ***kwargs*)

class pydal.adapters.postgres.**PostgrePG8000Boolean**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)

Bases: *pydal.adapters.postgres.PostgrePG8000New*

class pydal.adapters.postgres.**PostgrePG8000New**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)

Bases: *pydal.adapters.postgres.PostgrePG8000*

class pydal.adapters.postgres.**PostgrePsyco**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)

Bases: *pydal.adapters.postgres.Postgre*

adapt(*obj*)

drivers = ('psycopg2',)

class pydal.adapters.postgres.**PostgrePsycoBoolean**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)

Bases: *pydal.adapters.postgres.PostgrePsycoNew*

class pydal.adapters.postgres.**PostgrePsycoNew**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)

Bases: *pydal.adapters.postgres.PostgrePsyco*

pydal.adapters.sapdb module

pydal.adapters.sqlite module

class pydal.adapters.sqlite.**JDBCSQLite**(**args*, ***kwargs*)
Bases: *pydal.adapters.sqlite.SQLite*

after_connection()

connector()

drivers = ('zxJDBC_sqlite',)

class pydal.adapters.sqlite.**SQLite**(**args*, ***kwargs*)
Bases: *pydal.adapters.base.SQLAdapter*

after_connection()

connector()

dbengine = 'sqlite'

delete(*tablename*, *query*)

drivers = ('sqlite2', 'sqlite3')

select(*query*, *fields*, *attributes*)

static web2py_extract(*lookup*, *s*)

static web2py_regexp(*expression*, *item*)

class pydal.adapters.sqlite.**Spatialite**(**args*, ***kwargs*)
Bases: *pydal.adapters.sqlite.SQLite*

SPATIALLIBS = {'Windows': 'mod_spatialite.dll', 'Darwin': 'libspatialite.dylib', 'Linux': 'libspatialite.so'}

after_connections()

dbengine = 'spatialite'

pydal.adapters.teradata module

class pydal.adapters.teradata.**Teradata**(**args*, ***kwargs*)
Bases: *pydal.adapters.base.SQLAdapter*

close()

connector()

dbengine = ''

drivers = ('pyodbc',)

lastrowid(*table*)

Module contents

class pydal.adapters.**AdapterMeta**
Bases: type

Metaclass to support manipulation of adapter classes.

At the moment is used to intercept *entity_quoting* argument passed to DAL.

class pydal.adapters.**Adapters**(*namespace=None*)
Bases: pydal.helpers._internals.Dispatcher

get_for(*uri*)

register_for(**uris*)

pydal.adapters.**with_connection**(*f*)

pydal.adapters.**with_connection_or_raise**(*f*)

1.1.2 pydal.helpers package

Submodules

pydal.helpers.classes module

class pydal.helpers.classes.**BasicStorage**(**args*, ***kwargs*)
Bases: object

clear(**args*, ***kwargs*)

copy(**args*, ***kwargs*)

get(*key*, *default=None*)

has_key(*key*)

items()

iteritems()

iterkeys()

itervalues()

keys()

pop(**args*, ***kwargs*)

update(**args*, ***kwargs*)

values()

class pydal.helpers.classes.**DatabaseStoredFile**(*db*, *filename*, *mode*)

close()

close_connection()

escape(*obj*)

static **exists**(*db*, *filename*)

static **is_operational_error**(*db*, *error*)

static **is_programming_error**(*db*, *error*)

read(*bytes=None*)

readline()

static **try_create_web2py_filesystem**(*db*)

web2py_filesystems = set([])

write(*data*)

class pydal.helpers.classes.**ExecutionHandler**(*adapter*)
Bases: object

after_execute(*command*)

before_execute(*command*)

class pydal.helpers.classes.**FakeCursor**
Bases: object

The Python Database API Specification has a cursor() method, which NoSql drivers generally don't support. If the exception in this function is taken then it likely means that some piece of functionality has not yet been implemented in the driver. And something is using the cursor.

https://www.python.org/dev/peps/pep-0249/

warn_bad_usage(*attr*)

class pydal.helpers.classes.**FakeDriver**(**args*, ***kwargs*)
Bases: *pydal.helpers.classes.BasicStorage*

close()

commit()

cursor()

class pydal.helpers.classes.**MethodAdder**(*table*)
Bases: object

register(*method_name=None*)

class pydal.helpers.classes.**NullCursor**
Bases: *pydal.helpers.classes.FakeCursor*

lastrowid = 1

class pydal.helpers.classes.**NullDriver**(**args*, ***kwargs*)
Bases: *pydal.helpers.classes.FakeDriver*

class pydal.helpers.classes.**RecordDeleter**(*colset*, *table*, *id*)
Bases: *pydal.helpers.classes.RecordOperator*

class pydal.helpers.classes.**RecordOperator**(*colset*, *table*, *id*)
Bases: object

class pydal.helpers.classes.**RecordUpdater**(*colset*, *table*, *id*)
Bases: *pydal.helpers.classes.RecordOperator*

class pydal.helpers.classes.**Reference**
Bases: long

get(*key*, *default=None*)

pydal.helpers.classes.**Reference_pickler**(*data*)

pydal.helpers.classes.**Reference_unpickler**(*data*)

class pydal.helpers.classes.**SQLALL**(*table*)
Bases: object

Helper class providing a comma-separated string having all the field names (prefixed by table name and '.')

normally only called from within gluon.dal

class pydal.helpers.classes.**SQLCallableList**
Bases: list

class pydal.helpers.classes.**SQLCustomType**(*type='string'*, *native=None*, *encoder=None*, *decoder=None*, *validator=None*, *_class=None*, *widget=None*, *represent=None*)
Bases: object

Allows defining of custom SQL types

Parameters

- **type** – the web2py type (default = 'string')
- **native** – the backend type
- **encoder** – how to encode the value to store it in the backend
- **decoder** – how to decode the value retrieved from the backend
- **validator** – what validators to use (default = None, will use the default validator for type)

Example:: Define as:

decimal = SQLCustomType(type ='double', native ='integer', encoder =(lambda x: int(float(x) * 100)), decoder = (lambda x: Decimal("0.00") + Decimal(str(float(x)/100))))

db.define_table('example', Field('value', type=decimal))

endswith(*text=None*)

startswith(*text=None*)

class pydal.helpers.classes.**Serializable**
Bases: object

as_dict(*flat=False*, *sanitize=True*)

as_json(*sanitize=True*)

as_xml(*sanitize=True*)

as_yaml(*sanitize=True*)

class pydal.helpers.classes.**TimingHandler**(*adapter*)
Bases: *pydal.helpers.classes.ExecutionHandler*

MAXSTORAGE = 100

after_execute(*command*)

before_execute(*command*)

timings

pydal.helpers.classes.**pickle_basicstorage**(*s*)

pydal.helpers.methods module

pydal.helpers.methods.**archive_record**(*qset*, *fs*, *archive_table*, *current_record*)

pydal.helpers.methods.**auto_represent**(*field*)

pydal.helpers.methods.**auto_validators**(*field*)

pydal.helpers.methods.**bar_decode_integer**(*value*)

pydal.helpers.methods.**bar_decode_string**(*value*)

pydal.helpers.methods.**bar_encode**(*items*)

pydal.helpers.methods.**bar_escape**(*item*)

pydal.helpers.methods.**cleanup**(*text*)

Validates that the given text is clean: only contains [**0-9a-zA-Z_**]

pydal.helpers.methods.**geoLine**(**line*)

pydal.helpers.methods.**geoPoint**(*x*, *y*)

pydal.helpers.methods.**geoPolygon**(**line*)

pydal.helpers.methods.**hide_password**(*uri*)

pydal.helpers.methods.**int2uuid**(*n*)

pydal.helpers.methods.**list_represent**(*values*, *row=None*)

pydal.helpers.methods.**pluralize**(*singular, rules=[(<_sre.SRE_Pattern object at 0x7f634c4bf030>, <_sre.SRE_Pattern object at 0x7f634c4bf030>, 'children'), (<_sre.SRE_Pattern object at 0x7f634c59ef50>, <_sre.SRE_Pattern object at 0x7f634c59ef50>, 'eet'), (<_sre.SRE_Pattern object at 0x7f634c4a13f0>, <_sre.SRE_Pattern object at 0x7f634c4a13f0>, 'eeth'), (<_sre.SRE_Pattern object at 0x7f634c4a14b0>, <_sre.SRE_Pattern object at 0x7f634c4bf100>, 'f\\1aves'), (<_sre.SRE_Pattern object at 0x7f634c446030>, <_sre.SRE_Pattern object at 0x7f634c446030>, 'ses'), (<_sre.SRE_Pattern object at 0x7f634c4460e0>, <_sre.SRE_Pattern object at 0x7f634c4460e0>, 'men'), (<_sre.SRE_Pattern object at 0x7f634c446190>, <_sre.SRE_Pattern object at 0x7f634c446190>, 'ives'), (<_sre.SRE_Pattern object at 0x7f634c446240>, <_sre.SRE_Pattern object at 0x7f634c446240>, 'eaux'), (<_sre.SRE_Pattern object at 0x7f634c49b350>, <_sre.SRE_Pattern object at 0x7f634c49b350>, 'lves'), (<_sre.SRE_Pattern object at 0x7f634c4bf1d0>, <_sre.SRE_Pattern object at 0x7f634cb60098>, 'es'), (<_sre.SRE_Pattern object at 0x7f634c503570>, <_sre.SRE_Pattern object at 0x7f634cb60098>, 'es'), (<_sre.SRE_Pattern object at 0x7f634c5ba9e0>, <_sre.SRE_Pattern object at 0x7f634c4a0db0>, 'ies'), (<_sre.SRE_Pattern object at 0x7f634cb60098>, <_sre.SRE_Pattern object at 0x7f634cb60098>, 's')]*)

pydal.helpers.methods.**smart_query**(*fields*, *text*)

pydal.helpers.methods.**use_common_filters**(*query*)

pydal.helpers.methods.**uuid2int**(*uuidv*)

pydal.helpers.methods.**varquote_aux**(*name*, *quotestr='%s'*)

pydal.helpers.methods.**xorify**(*orderby*)

pydal.helpers.regex module

Module contents

1.2 Submodules

1.3 pydal.base module

This file is part of the web2py Web Framework
Copyrighted by Massimo Di Pierro <mdipierro@cs.depaul.edu>
License: LGPLv3 (http://www.gnu.org/licenses/lgpl.html)

This file contains the DAL support for many relational databases, including:

- SQLite & SpatiaLite
- MySQL
- Postgres
- Firebird
- Oracle
- MS SQL
- DB2
- Interbase
- Ingres
- Informix (9+ and SE)
- SapDB (experimental)
- Cubrid (experimental)
- CouchDB (experimental)
- MongoDB (in progress)
- Google:nosql
- Google:sql
- Teradata
- IMAP (experimental)

Example of usage:

```
>>> # from dal import DAL, Field

### create DAL connection (and create DB if it doesn't exist)
>>> db = DAL(('sqlite://storage.sqlite','mysql://a:b@localhost/x'),
... folder=None)

### define a table 'person' (create/alter as necessary)
>>> person = db.define_table('person',Field('name','string'))
```

```
### insert a record
>>> id = person.insert(name='James')

### retrieve it by id
>>> james = person(id)

### retrieve it by name
>>> james = person(name='James')

### retrieve it by arbitrary query
>>> query = (person.name=='James') & (person.name.startswith('J'))
>>> james = db(query).select(person.ALL)[0]

### update one record
>>> james.update_record(name='Jim')
<Row {'id': 1, 'name': 'Jim'}>

### update multiple records by query
>>> db(person.name.like('J%')).update(name='James')
1

### delete records by query
>>> db(person.name.lower() == 'jim').delete()
0

### retrieve multiple records (rows)
>>> people = db(person).select(orderby=person.name,
... groupby=person.name, limitby=(0,100))

### further filter them
>>> james = people.find(lambda row: row.name == 'James').first()
>>> print james.id, james.name
1 James

### check aggregates
>>> counter = person.id.count()
>>> print db(person).select(counter).first()(counter)
1

### delete one record
>>> james.delete_record()
1

### delete (drop) entire database table
>>> person.drop()
```

Supported DAL URI strings:

```
'sqlite://test.db'
'spatialite://test.db'
'sqlite:memory'
'spatialite:memory'
'jdbc:sqlite://test.db'
'mysql://root:none@localhost/test'
'postgres://mdipierro:password@localhost/test'
'postgres:psycopg2://mdipierro:password@localhost/test'
'postgres:pg8000://mdipierro:password@localhost/test'
'jdbc:postgres://mdipierro:none@localhost/test'
```

```
'mssql://web2py:none@A64X2/web2py_test'
'mssql2://web2py:none@A64X2/web2py_test' # alternate mappings
'mssql3://web2py:none@A64X2/web2py_test' # better pagination (requires >= 2005)
'mssql4://web2py:none@A64X2/web2py_test' # best pagination (requires >= 2012)
'oracle://username:password@database'
'firebird://user:password@server:3050/database'
'db2:ibm_db_dbi://DSN=dsn;UID=user;PWD=pass'
'db2:pyodbc://driver=DB2;hostname=host;database=database;uid=user;pwd=password;port=port'
'firebird://username:password@hostname/database'
'firebird_embedded://username:password@c://path'
'informix://user:password@server:3050/database'
'informixu://user:password@server:3050/database' # unicode informix
'ingres://database'  # or use an ODBC connection string, e.g. 'ingres://dsn=dsn_name'
'google:datastore' # for google app engine datastore (uses ndb by default)
'google:sql' # for google app engine with sql (mysql compatible)
'teradata://DSN=dsn;UID=user;PWD=pass; DATABASE=database' # experimental
'imap://user:password@server:port' # experimental
'mongodb://user:password@server:port/database' # experimental
```

For more info:

```
help(DAL)
help(Field)
```

class pydal.base.**DAL**(*uri='sqlite://dummy.db'*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *check_reserved=None*, *migrate=True*, *fake_migrate=False*, *migrate_enabled=True*, *fake_migrate_all=False*, *decode_credentials=False*, *driver_args=None*, *adapter_args=None*, *attempts=5*, *auto_import=False*, *bigint_id=False*, *debug=False*, *lazy_tables=False*, *db_uid=None*, *do_connect=True*, *after_connection=None*, *tables=None*, *ignore_field_case=True*, *entity_quoting=True*, *table_hash=None*)

Bases: *pydal.helpers.classes.Serializable*, *pydal.helpers.classes.BasicStorage*

An instance of this class represents a database connection

Parameters

- **uri** (*str*) – contains information for connecting to a database. Defaults to *'sqlite://dummy.db'*

  Note: experimental: you can specify a dictionary as uri parameter i.e. with:

  ```
  db = DAL({"uri": "sqlite://storage.sqlite",
            "tables": {...}, ...})
  ```

  for an example of dict input you can check the output of the scaffolding db model with

  db.as_dict()

  Note that for compatibility with Python older than version 2.6.5 you should cast your dict input keys to str due to a syntax limitation on kwarg names. for proper DAL dictionary input you can use one of:

  ```
  obj = serializers.cast_keys(dict, [encoding="utf-8"])
  #or else (for parsing json input)
  obj = serializers.loads_json(data, unicode_keys=False)
  ```

- **pool_size** – How many open connections to make to the database object.

- **folder** – where .table files will be created. Automatically set within web2py. Use an explicit path when using DAL outside web2py
- **db_codec** – string encoding of the database (default: 'UTF-8')
- **table_hash** – database identifier with .tables. If your connection hash change you can still using old .tables if they have db_hash as prefix
- **check_reserved** – list of adapters to check tablenames and column names against sql/nosql reserved keywords. Defaults to *None*
  - 'common' List of sql keywords that are common to all database types such as "SELECT, INSERT". (recommended)
  - 'all' Checks against all known SQL keywords
  - '<adapternname>" Checks against the specific adapters list of keywords
  - '<adapternname>_nonreserved' Checks against the specific adapters list of nonreserved keywords. (if available)
- **migrate** – sets default migrate behavior for all tables
- **fake_migrate** – sets default fake_migrate behavior for all tables
- **migrate_enabled** – If set to False disables ALL migrations
- **fake_migrate_all** – If set to True fake migrates ALL tables
- **attempts** – Number of times to attempt connecting
- **auto_import** – If set to True, tries import automatically table definitions from the databases folder (works only for simple models)
- **bigint_id** – If set, turn on bigint instead of int for id and reference fields
- **lazy_tables** – delays table definition until table access
- **after_connection** – can a callable that will be executed after the connection

Example

Use as:

```
db = DAL('sqlite://test.db')
```

or:

```
db = DAL(**{"uri": ..., "tables": [...]...}) # experimental

db.define_table('tablename', Field('fieldname1'),
                             Field('fieldname2'))
```

class Row (**args*, ***kwargs*)

Bases: *pydal.helpers.classes.BasicStorage*

A dictionary that lets you do d['a'] as well as d.a this is only used to store a *Row*

as_dict (*datetime_to_str=False*, *custom_types=None*)

as_json (*mode='object'*, *default=None*, *colnames=None*, *serialize=True*, ***kwargs*)

serializes the row to a JSON object kwargs are passed to .as_dict method only "object" mode supported

serialize = *False* used by Rows.as_json

TODO: return array mode with query column order

mode and colnames are not implemented

as_xml(*row_name='row'*, *colnames=None*, *indent=' '*)

get(*key*, *default=None*)

class DAL.**Rows**(*db=None*, *records=[]*, *colnames=[]*, *compact=True*, *rawrows=None*)
Bases: *pydal.objects.BasicRows*

A wrapper for the return value of a select. It basically represents a table. It has an iterator and each row is represented as a *Row* dictionary.

append(*row*)

column(*column=None*)

exclude(*f*)
Removes elements from the calling Rows object, filtered by the function *f*, and returns a new Rows object containing the removed elements

find(*f*, *limitby=None*)
Returns a new Rows object, a subset of the original object, filtered by the function *f*

first()

group_by_value(**fields*, ***args*)
Regroups the rows, by one of the fields

insert(*position*, *row*)

join(*field*, *name=None*, *constraint=None*, *fields=[]*, *orderby=None*)

last()

render(*i=None*, *fields=None*)
Takes an index and returns a copy of the indexed row with values transformed via the "represent" attributes of the associated fields.

Parameters

- **i** – index. If not specified, a generator is returned for iteration over all the rows.
- **fields** – a list of fields to transform (if None, all fields with "represent" attributes will be transformed)

setvirtualfields(***keyed_virtualfields*)
For reference:

```
db.define_table('x', Field('number', 'integer'))
if db(db.x).isempty(): [db.x.insert(number=i) for i in range(10)]

from gluon.dal import lazy_virtualfield

class MyVirtualFields(object):
    # normal virtual field (backward compatible, discouraged)
    def normal_shift(self): return self.x.number+1
    # lazy virtual field (because of @staticmethod)
    @lazy_virtualfield
    def lazy_shift(instance, row, delta=4): return row.x.number+delta
db.x.virtualfields.append(MyVirtualFields())
```

```
        for row in db(db.x).select():
            print row.number, row.normal_shift, row.lazy_shift(delta=7)
```

sort(*f*, *reverse=False*)
Returns a list of sorted elements (not sorted in place)

class DAL.**Table**(*db*, *tablename*, **fields*, ***args*)
Bases: *pydal.helpers.classes.Serializable*, *pydal.helpers.classes.BasicStorage*

Represents a database table

Example::

You can create a table as:: db = DAL(...) db.define_table('users', Field('name'))

And then:

```
        db.users.insert(name='me') # print db.users._insert(...) to see SQL
        db.users.drop()
```

as_dict(*flat=False*, *sanitize=True*)

bulk_insert(*items*)
here items is a list of dictionaries

create_index(*name*, **fields*, ***kwargs*)

drop(*mode=''*)

drop_index(*name*)

fields

import_from_csv_file(*csvfile*, *id_map=None*, *null='<NULL>'*, *unique='uuid'*, *id_offset=None*, *transform=None*, **args*, ***kwargs*)
Import records from csv file. Column headers must have same names as table fields. Field 'id' is ignored. If column names read 'table.file' the 'table.' prefix is ignored.
- 'unique' argument is a field which must be unique (typically a uuid field)
- 'restore' argument is default False; if set True will remove old values in table first.
- 'id_map' if set to None will not map ids

The import will keep the id numbers in the restored table. This assumes that there is an field of type id that is integer and in incrementing order. Will keep the id numbers in restored table.

insert(***fields*)

on(*query*)

sqlsafe

sqlsafe_alias

truncate(*mode=''*)

update(**args*, ***kwargs*)

update_or_insert(*_key=<function <lambda>>*, ***values*)

validate_and_insert(***fields*)

validate_and_update(*_key=<function <lambda>>*, ***fields*)

validate_and_update_or_insert(*_key=<function <lambda>>*, ***fields*)

with_alias(*alias*)

DAL.**as_dict**(*flat=False*, *sanitize=True*)

DAL.**can_join**()

DAL.**check_reserved_keyword**(*name*)
Validates *name* against SQL keywords Uses self.check_reserve which is a list of operators to use.

DAL.**close**()

DAL.**commit**()

DAL.**define_table**(*tablename*, **fields*, ***args*)

static DAL.**distributed_transaction_begin**(**instances*)

static DAL.**distributed_transaction_commit**(**instances*)

DAL.**executesql**(*query*, *placeholders=None*, *as_dict=False*, *fields=None*, *colnames=None*, *as_ordered_dict=False*)
Executes an arbitrary query

Parameters

- **query** (*str*) – the query to submit to the backend
- **placeholders** – is optional and will always be None. If using raw SQL with placeholders, placeholders may be a sequence of values to be substituted in or, (if supported by the DB driver), a dictionary with keys matching named placeholders in your SQL.
- **as_dict** – will always be None when using DAL. If using raw SQL can be set to True and the results cursor returned by the DB driver will be converted to a sequence of dictionaries keyed with the db field names. Results returned with as_dict=True are the same as those returned when applying .to_list() to a DAL query. If "as_ordered_dict"=True the behaviour is the same as when "as_dict"=True with the keys (field names) guaranteed to be in the same order as returned by the select name executed on the database.
- **fields** – list of DAL Fields that match the fields returned from the DB. The Field objects should be part of one or more Table objects defined on the DAL object. The "fields" list can include one or more DAL Table objects in addition to or instead of including Field objects, or it can be just a single table (not in a list). In that case, the Field objects will be extracted from the table(s).

  Note: if either *fields* or *colnames* is provided, the results will be converted to a DAL *Rows* object using the *db._adapter.parse()* method

- **colnames** – list of field names in tablename.fieldname format

Note: It is also possible to specify both "fields" and the associated "colnames". In that case, "fields" can also include DAL Expression objects in addition to Field objects. For Field objects in "fields", the associated "colnames" must still be in tablename.fieldname format. For Expression objects in "fields", the associated "colnames" can be any arbitrary labels.

DAL Table objects referred to by "fields" or "colnames" can be dummy tables and do not have to represent any real tables in the database. Also, note that the "fields" and "colnames" must be in the same order as the fields in the results cursor returned from the DB.

DAL.**execution_handlers = [<class 'pydal.helpers.classes.TimingHandler'>]**

DAL.**export_to_csv_file**(*ofile*, **args*, ***kwargs*)

static DAL.**get_instances**()
Returns a dictionary with uri as key with timings and defined tables:

```
        {'sqlite://storage.sqlite': {
            'dbstats': [(select auth_user.email from auth_user, 0.02009)],
            'dbtables': {
                'defined': ['auth_cas', 'auth_event', 'auth_group',
                    'auth_membership', 'auth_permission', 'auth_user'],
                'lazy': '[]'
                }
            }
        }
```

DAL.**has_representer**(*name*)

DAL.**import_from_csv_file**(*ifile*, *id_map=None*, *null='<NULL>'*, *unique='uuid'*, *map_tablenames=None*, *ignore_missing_tables=False*, **args*, ***kwargs*)

DAL.**import_table_definitions**(*path*, *migrate=False*, *fake_migrate=False*, *tables=None*)

DAL.**lazy_define_table**(*tablename*, **fields*, ***args*)

DAL.**logger = <logging.Logger object>**

DAL.**parse_as_rest**(*patterns*, *args*, *vars*, *queries=None*, *nested_select=True*)

DAL.**record_operators = {'update_record': <class 'pydal.helpers.classes.RecordUpdater'>, 'delete_record': <class '**

DAL.**represent**(*name*, **args*, ***kwargs*)

DAL.**representers = {}**

DAL.**rollback**()

DAL.**serializers = None**

static DAL.**set_folder**(*folder*)

DAL.**smart_query**(*fields*, *text*)

DAL.**tables**

DAL.**uuid**(*x*)

DAL.**validators = None**

DAL.**validators_method = None**

DAL.**where**(*query=None*, *ignore_common_filters=None*)

pydal.base.**DAL_pickler**(*db*)

pydal.base.**DAL_unpickler**(*db_uid*)

class pydal.base.**MetaDAL**

Bases: type

1.4 pydal.connection module

class pydal.connection.**ConnectionPool**

Bases: object

POOLS = {}

after_connection()

after_connection_hook()
Hook for the after_connection parameter

check_active_connection = True

close(*action='commit'*, *really=True*)

static **close_all_instances**(*action*)
to close cleanly databases in a multithreaded environment

close_cursor(*cursor*)

connection

cursor

cursors

lock_cursor(*cursor*)

reconnect()
Defines: *self.connection* and *self.cursor* if *self.pool_size>0* it will try pull the connection from the pool if the connection is not active (closed by db server) it will loop if not *self.pool_size* or no active connections in pool makes a new one

release_cursor(*cursor*)

static **set_folder**(*folder*)

1.5 pydal.objects module

class pydal.objects.**BasicRows**
Bases: object

Abstract class for Rows and IterRows

as_csv()
Serializes the table into a csv file

as_dict(*key='id'*, *compact=True*, *storage_to_dict=True*, *datetime_to_str=False*, *custom_types=None*)
Returns the data as a dictionary of dictionaries (storage_to_dict=True) or records (False)

Parameters

- **key** – the name of the field to be used as dict key, normally the id
- **compact** – ? (default True)
- **storage_to_dict** – when True returns a dict, otherwise a list(default True)
- **datetime_to_str** – convert datetime fields as strings (default False)

as_json(*mode='object'*, *default=None*)
Serializes the rows to a JSON list or object with objects mode='object' is not implemented (should return a nested object structure)

as_list(*compact=True*, *storage_to_dict=True*, *datetime_to_str=False*, *custom_types=None*)
Returns the data as a list or dictionary.

Parameters

- **storage_to_dict** – when True returns a dict, otherwise a list

- **datetime_to_str** – convert datetime fields as strings

as_trees(*parent_name='parent_id'*, *children_name='children'*, *render=False*)
returns the data as list of trees.

Parameters

- **parent_name** – the name of the field holding the reference to the parent (default parent_id).
- **children_name** – the name where the children of each row will be stored as a list (default children).
- **render** – whether we will render the fields using their represent (default False) can be a list of fields to render or True to render all.

as_xml(*row_name='row'*, *rows_name='rows'*)

export_to_csv_file(*ofile*, *null='<NULL>'*, **args*, ***kwargs*)
Exports data to csv, the first line contains the column names

Parameters

- **ofile** – where the csv must be exported to
- **null** – how null values must be represented (default '<NULL>')
- **delimiter** – delimiter to separate values (default ',')
- **quotechar** – character to use to quote string values (default '"')
- **quoting** – quote system, use csv.QUOTE_*** (default csv.QUOTE_MINIMAL)
- **represent** – use the fields .represent value (default False)
- **colnames** – list of column names to use (default self.colnames)

This will only work when exporting rows objects!!!! DO NOT use this with db.export_to_csv()

json(*mode='object'*, *default=None*)
Serializes the rows to a JSON list or object with objects mode='object' is not implemented (should return a nested object structure)

xml(*strict=False*, *row_name='row'*, *rows_name='rows'*)
Serializes the table using sqlhtml.SQLTABLE (if present)

class pydal.objects.**Expression**(*db*, *op*, *first=None*, *second=None*, *type=None*, ***optional_args*)
Bases: object

abs()

avg()

belongs(**value*, ***kwattr*)
Accepts the following inputs:

```
field.belongs(1, 2)
field.belongs((1, 2))
field.belongs(query)
```

Does NOT accept:

field.belongs(1)

If the set you want back includes *None* values, you can do:

```
        field.belongs((1, None), null=True)
```

coalesce(**others*)

coalesce_zero()

contains(*value*, *all=False*, *case_sensitive=False*)
For GAE contains() is always case sensitive

day()

endswith(*value*)

epoch()

hour()

ilike(*value*, *escape=None*)

len()

like(*value*, *case_sensitive=True*, *escape=None*)

lower()

max()

min()

minutes()

month()

regexp(*value*)

replace(*a*, *b*)

seconds()

st_asgeojson(*precision=15*, *options=0*, *version=1*)

st_astext()

st_contains(*value*)

st_distance(*other*)

st_dwithin(*value*, *distance*)

st_equals(*value*)

st_intersects(*value*)

st_overlaps(*value*)

st_simplify(*value*)

st_simplifypreservetopology(*value*)

st_touches(*value*)

st_within(*value*)

st_x()

st_y()

startswith(*value*)

sum()

upper()

with_alias(*alias*)

year()

class pydal.objects.**Field**(*fieldname*, *type='string'*, *length=None*, *default=<function <lambda>>*, *required=False*, *requires=<function <lambda>>*, *ondelete='CASCADE'*, *notnull=False*, *unique=False*, *uploadfield=True*, *widget=None*, *label=None*, *comment=None*, *writable=True*, *readable=True*, *update=None*, *authorize=None*, *autodelete=False*, *represent=None*, *uploadfolder=None*, *uploadseparate=False*, *uploadfs=None*, *compute=None*, *custom_store=None*, *custom_retrieve=None*, *custom_retrieve_file_properties=None*, *custom_delete=None*, *filter_in=None*, *filter_out=None*, *custom_qualifier=None*, *map_none=None*, *rname=None*)

Bases: *pydal.objects.Expression*, *pydal.helpers.classes.Serializable*

Lazy

Represents a database field

Example

Usage:

```
a = Field(name, 'string', length=32, default=None, required=False,
    requires=IS_NOT_EMPTY(), ondelete='CASCADE',
    notnull=False, unique=False,
    uploadfield=True, widget=None, label=None, comment=None,
    uploadfield=True, # True means store on disk,
                      # 'a_field_name' means store in this field in db
                      # False means file content will be discarded.
    writable=True, readable=True, update=None, authorize=None,
    autodelete=False, represent=None, uploadfolder=None,
    uploadseparate=False # upload to separate directories by uuid_keys
                         # first 2 character and tablename.fieldname
                         # False - old behavior
                         # True - put uploaded file in
                         #   <uploaddir>/<tablename>.<fieldname>/uuid_key[:2]
                         #        directory)
    uploadfs=None        # a pyfilesystem where to store upload
    )
```

to be used as argument of *DAL.define_table*

alias of *FieldMethod*

Method

alias of *FieldMethod*

Virtual

alias of *FieldVirtual*

as_dict(*flat=False*, *sanitize=True*)

clone(*point_self_references_to=False*, ***args*)

count(*distinct=None*)

formatter(*value*)

retrieve(*name*, *path=None*, *nameonly=False*)
If *nameonly==True* return (filename, fullfilename) instead of (filename, stream)

retrieve_file_properties(*name*, *path=None*)

set_attributes(**args*, ***attributes*)

sqlsafe

sqlsafe_name

store(*file*, *filename=None*, *path=None*)

validate(*value*)

class pydal.objects.**FieldMethod**(*name*, *f=None*, *handler=None*)
Bases: object

class pydal.objects.**FieldVirtual**(*name*, *f=None*, *ftype='string'*, *label=None*, *table_name=None*)
Bases: object

class pydal.objects.**IterRows**(*db*, *sql*, *fields*, *colnames*, *blob_decode*, *cacheable*)
Bases: *pydal.objects.BasicRows*

first()

next()

class pydal.objects.**LazyReferenceGetter**(*table*, *id*)
Bases: object

class pydal.objects.**LazySet**(*field*, *id*)
Bases: object

count(*distinct=None*, *cache=None*)

delete()

delete_uploaded_files(*upload_fields=None*)

isempty()

nested_select(**fields*, ***attributes*)

select(**fields*, ***attributes*)

update(***update_fields*)

update_naive(***update_fields*)

validate_and_update(***update_fields*)

where(*query*, *ignore_common_filters=False*)

class pydal.objects.**Query**(*db*, *op*, *first=None*, *second=None*, *ignore_common_filters=False*, ***optional_args*)
Bases: *pydal.helpers.classes.Serializable*

Necessary to define a set. It can be stored or can be passed to *DAL.__call__()* to obtain a *Set*

Example

Use as:

```
query = db.users.name=='Max'
set = db(query)
records = set.select()
```

as_dict (*flat=False*, *sanitize=True*)
Experimental stuff

This allows to return a plain dictionary with the basic query representation. Can be used with json/xml services for client-side db I/O

Example

Usage:

```
q = db.auth_user.id != 0
q.as_dict(flat=True)
{
"op": "NE",
"first":{
    "tablename": "auth_user",
    "fieldname": "id"
    },
"second":0
}
```

case (*t=1*, *f=0*)

class pydal.objects.**Row** (**args*, ***kwargs*)
Bases: *pydal.helpers.classes.BasicStorage*

A dictionary that lets you do d['a'] as well as d.a this is only used to store a *Row*

as_dict (*datetime_to_str=False*, *custom_types=None*)

as_json (*mode='object'*, *default=None*, *colnames=None*, *serialize=True*, ***kwargs*)
serializes the row to a JSON object kwargs are passed to .as_dict method only "object" mode supported

serialize = *False* used by Rows.as_json

TODO: return array mode with query column order

mode and colnames are not implemented

as_xml (*row_name='row'*, *colnames=None*, *indent=' '*)

get (*key*, *default=None*)

class pydal.objects.**Rows** (*db=None*, *records=[]*, *colnames=[]*, *compact=True*, *rawrows=None*)
Bases: *pydal.objects.BasicRows*

A wrapper for the return value of a select. It basically represents a table. It has an iterator and each row is represented as a *Row* dictionary.

append (*row*)

column (*column=None*)

exclude (*f*)
Removes elements from the calling Rows object, filtered by the function *f*, and returns a new Rows object containing the removed elements

find(*f*, *limitby=None*)
Returns a new Rows object, a subset of the original object, filtered by the function *f*

first()

group_by_value(**fields*, ***args*)
Regroups the rows, by one of the fields

insert(*position*, *row*)

join(*field*, *name=None*, *constraint=None*, *fields=[]*, *orderby=None*)

last()

render(*i=None*, *fields=None*)
Takes an index and returns a copy of the indexed row with values transformed via the "represent" attributes of the associated fields.

Parameters

- **i** – index. If not specified, a generator is returned for iteration over all the rows.
- **fields** – a list of fields to transform (if None, all fields with "represent" attributes will be transformed)

setvirtualfields(***keyed_virtualfields*)
For reference:

```
db.define_table('x', Field('number', 'integer'))
if db(db.x).isempty(): [db.x.insert(number=i) for i in range(10)]

from gluon.dal import lazy_virtualfield

class MyVirtualFields(object):
    # normal virtual field (backward compatible, discouraged)
    def normal_shift(self): return self.x.number+1
    # lazy virtual field (because of @staticmethod)
    @lazy_virtualfield
    def lazy_shift(instance, row, delta=4): return row.x.number+delta
db.x.virtualfields.append(MyVirtualFields())

for row in db(db.x).select():
    print row.number, row.normal_shift, row.lazy_shift(delta=7)
```

sort(*f*, *reverse=False*)
Returns a list of sorted elements (not sorted in place)

class pydal.objects.**Set**(*db*, *query*, *ignore_common_filters=None*)
Bases: *pydal.helpers.classes.Serializable*

Represents a set of records in the database. Records are identified by the *query=Query(...)* object. Normally the Set is generated by *DAL.__call__(Query(...))*

Given a set, for example:

```
myset = db(db.users.name=='Max')
```

you can:

```
myset.update(db.users.name='Massimo')
myset.delete() # all elements in the set
myset.select(orderby=db.users.id, groupby=db.users.name, limitby=(0, 10))
```

and take subsets:

subset = myset(db.users.id<5)

as_dict(*flat=False*, *sanitize=True*)

build(*d*)
Experimental: see .parse()

count(*distinct=None*, *cache=None*)

delete()

delete_uploaded_files(*upload_fields=None*)

isempty()

iterselect(**fields*, ***attributes*)

nested_select(**fields*, ***attributes*)

parse(*dquery*)
Experimental: Turn a dictionary into a Query object

select(**fields*, ***attributes*)

update(***update_fields*)

update_naive(***update_fields*)
Same as update but does not call table._before_update and _after_update

validate_and_update(***update_fields*)

where(*query*, *ignore_common_filters=False*)

class pydal.objects.**Table**(*db*, *tablename*, **fields*, ***args*)
Bases: *pydal.helpers.classes.Serializable*, *pydal.helpers.classes.BasicStorage*

Represents a database table

Example::

You can create a table as:: db = DAL(...) db.define_table('users', Field('name'))

And then:

```
db.users.insert(name='me') # print db.users._insert(...) to see SQL
db.users.drop()
```

as_dict(*flat=False*, *sanitize=True*)

bulk_insert(*items*)
here items is a list of dictionaries

create_index(*name*, **fields*, ***kwargs*)

drop(*mode=''*)

drop_index(*name*)

fields

import_from_csv_file(*csvfile*, *id_map=None*, *null='<NULL>'*, *unique='uuid'*, *id_offset=None*, *transform=None*, **args*, ***kwargs*)
Import records from csv file. Column headers must have same names as table fields. Field 'id' is ignored. If column names read 'table.file' the 'table.' prefix is ignored.

- 'unique' argument is a field which must be unique (typically a uuid field)
- 'restore' argument is default False; if set True will remove old values in table first.

- 'id_map' if set to None will not map ids

The import will keep the id numbers in the restored table. This assumes that there is an field of type id that is integer and in incrementing order. Will keep the id numbers in restored table.

insert(***fields*)

on(*query*)

sqlsafe

sqlsafe_alias

truncate(*mode=''*)

update(**args*, ***kwargs*)

update_or_insert(*_key=<function <lambda>>*, ***values*)

validate_and_insert(***fields*)

validate_and_update(*_key=<function <lambda>>*, ***fields*)

validate_and_update_or_insert(*_key=<function <lambda>>*, ***fields*)

with_alias(*alias*)

class pydal.objects.**VirtualCommand**(*method*, *row*)

Bases: object

pydal.objects.**pickle_row**(*s*)

1.6 Module contents

p

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y