pyDAL Documentation

Release 15.5.29

web2py-developers

October 07, 2015

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.**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.base.**BaseAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)
Bases: pydal.connection.ConnectionPool

ADD(*first*, *second*)

AGGREGATE(*first*, *what*)

ALLOW_NULL()

AND(*first*, *second*)

AS(*first*, *second*)

BELONGS(*first*, *second*)

CASE(*query*, *t*, *f*)

CAST(*first*, *second*)

COALESCE(*first*, *second*)

COALESCE_ZERO (*first*)

COMMA (*first*, *second*)

CONCAT (**items*)

CONTAINS (*first*, *second*, *case_sensitive=True*)

COUNT (*first*, *distinct=None*)

DIV (*first*, *second*)

ENDSWITH (*first*, *second*)

EPOCH (*first*)

EQ (*first*, *second=None*)

EXTRACT (*first*, *what*)

FALSE = 'F'

FALSE_exp = '0'

GE (*first*, *second=None*)

GT (*first*, *second=None*)

ILIKE (*first*, *second*)
Case insensitive like operator

INVERT (*first*)

JOIN ()

LE (*first*, *second=None*)

LEFT_JOIN ()

LENGTH (*first*)

LIKE (*first*, *second*)
Case sensitive like operator

LOWER (*first*)

LT (*first*, *second=None*)

MOD (*first*, *second*)

MUL (*first*, *second*)

NE (*first*, *second=None*)

NOT (*first*)

NOT_NULL (*default*, *field_type*)

ON (*first*, *second*)

OR (*first*, *second*)

PRIMARY_KEY (*key*)

QUOTE_TEMPLATE = "'%s'"

RANDOM ()

RAW (*first*)

REGEXP (*first*, *second*)
Regular expression operator

REPLACE (*first*, *tup*)

STARTSWITH (*first*, *second*)

SUB (*first*, *second*)

SUBSTRING (*field*, *parameters*)

TRUE = 'T'

TRUE_exp = '1'

T_SEP = ' '

UPPER (*first*)

adapt (*obj*)

alias (*table*, *alias*)
Given a table object, makes a new table object with alias name.

build_parsemap ()

bulk_insert (*table*, *items*)

can_join ()

can_select_for_update = True

close_connection ()

commit ()

commit_on_alter_table = False

commit_prepared (*key*)

common_filter (*query*, *tablenames*)

concat_add (*tablename*)

connection = None

connector (**args*, ***kwargs*)

constraint_name (*table*, *fieldname*)

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

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

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

dbpath = None

delete (*tablename*, *query*)

distributed_transaction_begin (*key*)

driver = None

driver_auto_json = []

driver_name = None

drivers = ()

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

execute(**a*, ***b*)

expand(*expression*, *field_type=None*, *colnames=False*)

expand_all(*fields*, *tablenames*)

file_close(*fileobj*)

file_delete(*filename*)

file_exists(*filename*)

file_open(*filename*, *mode='rb'*, *lock=True*)

find_driver(*adapter_args*, *uri=None*)

folder = None

get_table(*query*)

id_query(*table*)

insert(*table*, *fields*)

isOperationalError(*exception*)

isProgrammingError(*exception*)

is_numerical_type(*ftype*)

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

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

lastrowid(*table*)

log(*message*, *table=None*)
Logs migrations

It will not log changes if logfile is not specified. Defaults to sql.log

log_execute(**a*, ***b*)

migrate_table(*table*, *sql_fields*, *sql_fields_old*, *sql_fields_aux*, *logfile*, *fake_migrate=False*)

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

parse_blob(*value*, *field_type*)

parse_boolean(*value*, *field_type*)

parse_date(*value*, *field_type*)

parse_datetime(*value*, *field_type*)

parse_decimal(*value*, *field_type*)

parse_double(*value*, *field_type*)

parse_id(*value*, *field_type*)

parse_integer(*value*, *field_type*)

parse_json(*value*, *field_type*)

parse_list_integers(*value*, *field_type*)

parse_list_references(*value*, *field_type*)

parse_list_strings(*value*, *field_type*)

parse_reference(*value*, *field_type*)

parse_time(*value*, *field_type*)

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

prepare(*key*)

represent(*obj*, *fieldtype*)

represent_exceptions(*obj*, *fieldtype*)

rollback()

rollback_prepared(*key*)

rowslice(*rows*, *minimum=0*, *maximum=None*)
By default this function does nothing; overload when db does not do slicing.

save_dbt(*table*, *sql_fields_current*)

select(*query*, *fields*, *attributes*)
Always returns a Rows object, possibly empty.

select_limitby(*sql_s*, *sql_f*, *sql_t*, *sql_w*, *sql_o*, *limitby*)

sequence_name(*tablename*)

smart_adapt(*obj*)

sqlsafe_field(*fieldname*)

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

support_distributed_transaction = False

table_alias(*tbl*)

tables(**queries*)

test_query = 'SELECT 1;'

trigger_name(*tablename*)

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

types = {'reference': 'INTEGER REFERENCES %(foreign_key)s ON DELETE %(on_delete_action)s', 'text': 'TEXT',

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

uploads_in_blob = False

varquote(*name*)

class pydal.adapters.base.**NoSQLAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)

Bases: pydal.adapters.base.BaseAdapter

ADD(*first*, *second*)

AGGREGATE(*first*, *what*)

AND(*first*, *second*)

AS(*first*, *second*)

DIV(*first*, *second*)

ENDSWITH (*first*, *second=None*)

EXTRACT (*first*, *what*)

ILIKE (*first*, *second*)

LEFT_JOIN ()

LENGTH (*first*)

LOWER (*first*)

MUL (*first*, *second*)

ON (*first*, *second*)

OR (*first*, *second*)

PRIMARY_KEY (*key*)

QUOTE_TEMPLATE = '%s'

RANDOM ()

STARTSWITH (*first*, *second=None*)

SUB (*first*, *second*)

SUBSTRING (*field*, *parameters*)

UPPER (*first*)

can_join ()

can_select_for_update = False

close_connection ()
remember: no transactions on many NoSQL

commit ()
remember: no transactions on many NoSQL

commit_prepared (*key*)

concat_add (*table*)

constraint_name (*table*, *fieldname*)

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

distributed_transaction_begin (*key*)

drop (*table*, *mode*)

execute (**a*, ***b*)

id_query (*table*)

lastrowid (*table*)

log_execute (**a*, ***b*)

migrate_table (**a*, ***b*)

parse_list_integers (*value*, *field_type*)

parse_list_references (*value*, *field_type*)

parse_list_strings (*value*, *field_type*)

prepare (*key*)

represent(*obj*, *fieldtype*)

represent_exceptions(*obj*, *fieldtype*)

rollback()
remember: no transactions on many NoSQL

rollback_prepared(*key*)

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

pydal.adapters.couchdb module

class pydal.adapters.couchdb.**CouchDBAdapter**(*db*, *uri='couchdb://127.0.0.1:5984'*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)

Bases: pydal.adapters.base.NoSQLAdapter

AND(*first*, *second*)

COMMA(*first*, *second*)

EQ(*first*, *second*)

NE(*first*, *second*)

OR(*first*, *second*)

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

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

delete(*tablename*, *query*)

drivers = **('couchdb',)**

expand(*expression*, *field_type=None*)

file_close(*fileobj*)

file_exists(*filename*)

file_open(*filename*, *mode='rb'*, *lock=True*)

insert(*table*, *fields*)

represent(*obj*, *fieldtype*)

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

types = **{'string': <type 'str'>, 'reference': <type 'long'>, 'text': <type 'str'>, 'id': <type 'long'>, 'float': <type 'float'>,**

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

uploads_in_blob = **True**

pydal.adapters.cubrid module

class pydal.adapters.cubrid.**CubridAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)

Bases: pydal.adapters.mysql.MySQLAdapter

REGEX_URI = <_sre.SRE_Pattern object at 0x29c4a30>

after_connection()

drivers = ('cubriddb',)

pydal.adapters.db2 module

class pydal.adapters.db2.**DB2Adapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)

Bases: pydal.adapters.base.BaseAdapter

LEFT_JOIN()

RANDOM()

drivers = ('ibm_db_dbi', 'pyodbc')

execute(*command*, *placeholders=None*)

lastrowid(*table*)

represent_exceptions(*obj*, *fieldtype*)

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

select_limitby(*sql_s*, *sql_f*, *sql_t*, *sql_w*, *sql_o*, *limitby*)

types = {'reference': 'INT, FOREIGN KEY (%(field_name)s) REFERENCES %(foreign_key)s ON DELETE %(on_del

pydal.adapters.firebird module

class pydal.adapters.firebird.**FireBirdAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)

Bases: pydal.adapters.base.BaseAdapter

CONTAINS(*first*, *second*, *case_sensitive=False*)

EPOCH(*first*)

LENGTH(*first*)

NOT_NULL(*default*, *field_type*)

RANDOM()

REGEX_URI = <_sre.SRE_Pattern object at 0x29dd5e0>

SUBSTRING(*field*, *parameters*)

commit_on_alter_table = False

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

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

lastrowid(*table*)

select_limitby(*sql_s*, *sql_f*, *sql_t*, *sql_w*, *sql_o*, *limitby*)

sequence_name(*tablename*)

support_distributed_transaction = True

trigger_name(*tablename*)

types = {'reference': 'INTEGER REFERENCES %(foreign_key)s ON DELETE %(on_delete_action)s', 'text': 'BLOB S

class pydal.adapters.firebird.**FireBirdEmbeddedAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)

Bases: pydal.adapters.firebird.FireBirdAdapter

REGEX_URI = <_sre.SRE_Pattern object at 0x29accd0>

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

pydal.adapters.google_adapters module

Adapter for GAE

pydal.adapters.imap module

class pydal.adapters.imap.**IMAPAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)

Bases: pydal.adapters.base.NoSQLAdapter

IMAP server adapter

This class is intended as an interface with email IMAP servers to perform simple queries in the web2py DAL query syntax, so email read, search and other related IMAP mail services (as those implemented by brands like Google(r), and Yahoo!(r) can be managed from web2py applications.

The code uses examples by Yuji Tomita on this post: http://yuji.wordpress.com/2011/06/22/python-imaplib-imap-example-with-gmail/#comment-1137 and is based in docs for Python imaplib, python email and email IETF's (i.e. RFC2060 and RFC3501)

This adapter was tested with a small set of operations with Gmail(r). Other services requests could raise command syntax and response data issues.

It creates its table and field names "statically", meaning that the developer should leave the table and field definitions to the DAL instance by calling the adapter's .define_tables() method. The tables are defined with the IMAP server mailbox list information.

.define_tables() returns a dictionary mapping dal tablenames to the server mailbox names with the following structure:

{<tablename>: str <server mailbox name>}

Here is a list of supported fields:

Field	Type	Description
uid	string	
answered	boolean	Flag
created	date	
content	list:string	A list of dict text or html parts
to	string	
cc	string	
bcc	string	
size	integer	the amount of octets of the message*
deleted	boolean	Flag
draft	boolean	Flag
flagged	boolean	Flag
sender	string	
recent	boolean	Flag
seen	boolean	Flag
subject	string	
mime	string	The mime header declaration
email	string	The complete RFC822 message (*)
attachments	list	Each non text part as dict
encoding	string	The main detected encoding

(*) At the application side it is measured as the length of the RFC822 message string

WARNING: As row id's are mapped to email sequence numbers, make sure your imap client web2py app does not delete messages during select or update actions, to prevent updating or deleting different messages. Sequence numbers change whenever the mailbox is updated. To avoid this sequence numbers issues, it is recommended the use of uid fields in query references (although the update and delete in separate actions rule still applies).

```
# This is the code recommended to start imap support
# at the app's model:

imapdb = DAL("imap://user:password@server:port", pool_size=1) # port 993 for ssl
imapdb.define_tables()
```

Here is an (incomplete) list of possible imap commands:

```
# Count today's unseen messages
# smaller than 6000 octets from the
# inbox mailbox

q = imapdb.INBOX.seen == False
q &= imapdb.INBOX.created == datetime.date.today()
q &= imapdb.INBOX.size < 6000
unread = imapdb(q).count()

# Fetch last query messages
rows = imapdb(q).select()
```

```
# it is also possible to filter query select results with limitby and
# sequences of mailbox fields

set.select(<fields sequence>, limitby=(<int>, <int>))

# Mark last query messages as seen
messages = [row.uid for row in rows]
seen = imapdb(imapdb.INBOX.uid.belongs(messages)).update(seen=True)

# Delete messages in the imap database that have mails from mr. Gumby

deleted = 0
for mailbox in imapdb.tables
    deleted += imapdb(imapdb[mailbox].sender.contains("gumby")).delete()

# It is possible also to mark messages for deletion instead of ereasing them
# directly with set.update(deleted=True)


# This object give access
# to the adapter auto mailbox
# mapped names (which native
# mailbox has what table name)

imapdb.mailboxes <dict> # tablename, server native name pairs

# To retrieve a table native mailbox name use:
imapdb.<table>.mailbox

### New features v2.4.1:

# Declare mailboxes statically with tablename, name pairs
# This avoids the extra server names retrieval

imapdb.define_tables({"inbox": "INBOX"})

# Selects without content/attachments/email columns will only
# fetch header and flags

imapdb(q).select(imapdb.INBOX.sender, imapdb.INBOX.subject)
```

AND (*first*, *second*)

BELONGS (*first*, *second*)

CONTAINS (*first*, *second*, *case_sensitive=False*)

EQ (*first*, *second*)

GE (*first*, *second*)

GT (*first*, *second*)

LE (*first*, *second*)

LT (*first*, *second*)

NE (*first*, *second=None*)

NOT (*first*)

OR (*first*, *second*)

REGEX_URI = <_sre.SRE_Pattern object at 0x7f0c9d37fa80>

convert_date(*date*, *add=None*, *imf=False*)

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

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

dbengine = 'imap'

define_tables(*mailbox_names=None*)
Auto create common IMAP fileds

This function creates fields definitions "statically" meaning that custom fields as in other adapters should not be supported and definitions handled on a service/mode basis (local syntax for Gmail(r), Ymail(r)

Returns a dictionary with tablename, server native mailbox name pairs.

delete(*tablename*, *query*)

drivers = ('imaplib',)

encode_text(*text*, *charset*, *errors='replace'*)
convert text for mail to unicode

get_charset(*message*)

get_last_message(*tablename*)

get_mailboxes()
Query the mail database for mailbox names

get_query_mailbox(*query*)

get_uid_bounds(*tablename*)

static **header_represent**(*f*, *r*)

insert(*table*, *fields*)

is_flag(*flag*)

reconnect(*f=None*, *cursor=True*)
IMAP4 Pool connection method

imap connection lacks of self cursor command. A custom command should be provided as a replacement for connection pooling to prevent uncaught remote session closing

select(*query*, *fields*, *attributes*)
Searches and Fetches records and return web2py rows

types = {'boolean': <type 'bool'>, 'string': <type 'str'>, 'list:string': <type 'str'>, 'integer': <type 'int'>, 'date': <type '

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

uri = None
MESSAGE is an identifier for sequence number

pydal.adapters.informix module

class pydal.adapters.informix.**InformixAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)

Bases: pydal.adapters.base.BaseAdapter

NOT_NULL(*default*, *field_type*)

RANDOM()

REGEX_URI = <_sre.SRE_Pattern object at 0x7f0c9cc13030>

drivers = ('informixdb',)

execute(*command*)

lastrowid(*table*)

represent_exceptions(*obj*, *fieldtype*)

select_limitby(*sql_s*, *sql_f*, *sql_t*, *sql_w*, *sql_o*, *limitby*)

types = {'reference': 'INTEGER REFERENCES %(foreign_key)s ON DELETE %(on_delete_action)s', 'text': 'BLOB S

class pydal.adapters.informix.**InformixSEAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)

Bases: pydal.adapters.informix.InformixAdapter

work in progress

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

select_limitby(*sql_s*, *sql_f*, *sql_t*, *sql_w*, *sql_o*, *limitby*)

pydal.adapters.ingres module

class pydal.adapters.ingres.**IngresAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)

Bases: pydal.adapters.base.BaseAdapter

LEFT_JOIN()

RANDOM()

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

drivers = ('pyodbc',)

lastrowid(*table*)

select_limitby(*sql_s*, *sql_f*, *sql_t*, *sql_w*, *sql_o*, *limitby*)

types = {'reference': 'INT, FOREIGN KEY (%(field_name)s) REFERENCES %(foreign_key)s ON DELETE %(on_del

class pydal.adapters.ingres.**IngresUnicodeAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)

Bases: pydal.adapters.ingres.IngresAdapter

drivers = ('pyodbc',)

types = {'reference': 'INTEGER4, FOREIGN KEY (%(field_name)s) REFERENCES %(foreign_key)s ON DELETE %

pydal.adapters.mongo module

class pydal.adapters.mongo.**Binary**

Bases: object

class pydal.adapters.mongo.**MongoBlob**

Bases: pydal.adapters.mongo.Binary

MONGO_BLOB_BYTES = 0

MONGO_BLOB_NON_UTF8_STR = 1

static decode(*value*)

class pydal.adapters.mongo.**MongoDBAdapter**(*db*, *uri='mongodb://127.0.0.1:5984/db'*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)

Bases: pydal.adapters.base.NoSQLAdapter

ADD(*first*, *second*)

AND(*first*, *second*)

AS(*first*, *second*)

BELONGS(*first*, *second*)

COMMA(*first*, *second*)

CONTAINS(*first*, *second*, *case_sensitive=True*)

DIV(*first*, *second*)

ENDSWITH(*first*, *second*)

EQ(*first*, *second=None*)

GE(*first*, *second=None*)

GT(*first*, *second*)

ILIKE(*first*, *second*)

INVERT(*first*)

LE(*first*, *second=None*)

LIKE(*first*, *second*, *case_sensitive=True*)

LT (*first*, *second=None*)

MOD (*first*, *second*)

MUL (*first*, *second*)

NE (*first*, *second=None*)

NOT (*first*)

ON (*first*, *second*)

OR (*first*, *second*)

STARTSWITH (*first*, *second*)

SUB (*first*, *second*)

bulk_insert (*table*, *items*)

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

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

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

driver_auto_json = ['loads', 'dumps']

drivers = ('pymongo',)

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

error_messages = {'javascript_needed': 'This must yet be replaced with javascript in order to work.'}

expand (*expression*, *field_type=None*)

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

parse_blob (*value*, *field_type*)

parse_id (*value*, *field_type*)

parse_reference (*value*, *field_type*)

represent (*obj*, *fieldtype*)

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

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

types = {'string': <type 'str'>, 'reference': <type 'long'>, 'text': <type 'str'>, 'id': <type 'long'>, 'float': <type 'float'>,

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

uploads_in_blob = False

pydal.adapters.mssql module

class pydal.adapters.mssql.**MSSQL2Adapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)

Bases: pydal.adapters.mssql.MSSQLAdapter

ILIKE(*first*, *second*)

drivers = ('pyodbc',)

execute(**a*, ***b*)

represent(*obj*, *fieldtype*)

types = {'reference': 'INT, CONSTRAINT %(constraint_name)s FOREIGN KEY (%(field_name)s) REFERENCES %(

class pydal.adapters.mssql.**MSSQL3Adapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)

Bases: pydal.adapters.mssql.MSSQLAdapter

Experimental support for pagination in MSSQL

Requires MSSQL >= 2005, uses *ROW_NUMBER()*

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

select_limitby(*sql_s*, *sql_f*, *sql_t*, *sql_w*, *sql_o*, *limitby*)

types = {'reference': 'INT NULL, CONSTRAINT %(constraint_name)s FOREIGN KEY (%(field_name)s) REFEREN

class pydal.adapters.mssql.**MSSQL3NAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)

Bases: pydal.adapters.mssql.MSSQLNAdapter

drivers = ('pyodbc',)

Experimental support for pagination in MSSQL Experimental: see MSSQLNAdapter docstring for warnings

Requires MSSQL >= 2005, uses *ROW_NUMBER()*

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

select_limitby(*sql_s*, *sql_f*, *sql_t*, *sql_w*, *sql_o*, *limitby*)

types = {'reference': 'INT NULL, CONSTRAINT %(constraint_name)s FOREIGN KEY (%(field_name)s) REFEREN

class pydal.adapters.mssql.**MSSQL4Adapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)

Bases: pydal.adapters.mssql.MSSQLAdapter

Support for "native" pagination

Requires MSSQL >= 2012, uses *OFFSET ... ROWS ... FETCH NEXT ... ROWS ONLY*

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

select_limitby(*sql_s*, *sql_f*, *sql_t*, *sql_w*, *sql_o*, *limitby*)

types = {'reference': 'INT NULL, CONSTRAINT %(constraint_name)s FOREIGN KEY (%(field_name)s) REFEREN

class pydal.adapters.mssql.**MSSQL4NAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)

Bases: pydal.adapters.mssql.MSSQLNAdapter

Experimental: see MSSQLNAdapter docstring for warnings Support for "native" pagination

Unicode-compatible version Requires MSSQL >= 2012, uses *OFFSET ... ROWS ... FETCH NEXT ... ROWS ONLY* After careful testing, this should be the de-facto adapter for recent MSSQL backends

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

select_limitby(*sql_s*, *sql_f*, *sql_t*, *sql_w*, *sql_o*, *limitby*)

types = {'reference': 'INT NULL, CONSTRAINT %(constraint_name)s FOREIGN KEY (%(field_name)s) REFEREN

class pydal.adapters.mssql.**MSSQLAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)

Bases: pydal.adapters.base.BaseAdapter

AGGREGATE(*first*, *what*)

ALLOW_NULL()

CAST(*first*, *second*)

CONCAT(**items*)

EPOCH(*first*)

EXTRACT(*field*, *what*)

FALSE = 0

LEFT_JOIN()

LENGTH(*first*)

PRIMARY_KEY(*key*)

QUOTE_TEMPLATE = "'%s'"

RANDOM()

REGEX_ARGPATTERN = <_sre.SRE_Pattern object at 0x7f0c9d146440>

REGEX_DSN = <_sre.SRE_Pattern object at 0x7f0c9cc15d50>

REGEX_URI = <_sre.SRE_Pattern object at 0x29ce2d0>

ST_ASTEXT(*first*)

ST_CONTAINS(*first*, *second*)

ST_DISTANCE(*first*, *second*)

ST_EQUALS(*first*, *second*)

ST_INTERSECTS(*first*, *second*)

ST_OVERLAPS(*first*, *second*)

ST_TOUCHES(*first*, *second*)

ST_WITHIN(*first*, *second*)

SUBSTRING(*field*, *parameters*)

TRUE = 1

T_SEP = 'T'

concat_add(*tablename*)

drivers = ('pyodbc',)

lastrowid(*table*)

represent(*obj*, *fieldtype*)

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

select_limitby(*sql_s*, *sql_f*, *sql_t*, *sql_w*, *sql_o*, *limitby*)

types = {'reference': 'INT NULL, CONSTRAINT %(constraint_name)s FOREIGN KEY (%(field_name)s) REFEREN

varquote(*name*)

class pydal.adapters.mssql.**MSSQLNAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)

Bases: pydal.adapters.mssql.MSSQLAdapter

ILIKE(*first*, *second*)

drivers = ('pyodbc',)

Experimental – base class for handling unicode in MSSQL by default. Needs lots of testing. Try this on a fresh (or on a legacy) database. Using this in a database handled previously with non-unicode aware adapter is NOT supported

execute(**a*, ***b*)

represent(*obj*, *fieldtype*)

types = {'reference': 'INT, CONSTRAINT %(constraint_name)s FOREIGN KEY (%(field_name)s) REFERENCES %

class pydal.adapters.mssql.**SybaseAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)

Bases: pydal.adapters.mssql.MSSQLAdapter

drivers = 'Sybase'

types = {'reference': 'INT NULL, CONSTRAINT %(constraint_name)s FOREIGN KEY (%(field_name)s) REFEREN

class pydal.adapters.mssql.**VerticaAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)

Bases: pydal.adapters.mssql.MSSQLAdapter

EXTRACT(*first*, *what*)

T_SEP = ' '

drivers = ('pyodbc',)

execute(*a*)

lastrowid(*table*)

select_limitby(*sql_s*, *sql_f*, *sql_t*, *sql_w*, *sql_o*, *limitby*)

types = {'big-reference': 'BIGINT REFERENCES %(foreign_key)s ON DELETE %(on_delete_action)s', 'string': 'VA

pydal.adapters.mysql module

class pydal.adapters.mysql.**MySQLAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)

Bases: pydal.adapters.base.BaseAdapter

CAST(*first*, *second*)

CONCAT(**items*)

EPOCH(*first*)

QUOTE_TEMPLATE = "%s"

RANDOM()

REGEXP(*first*, *second*)

REGEX_URI = <_sre.SRE_Pattern object at 0x29c4a30>

SUBSTRING(*field*, *parameters*)

after_connection()

commit_on_alter_table = True

commit_prepared(*key*)

distributed_transaction_begin(*key*)

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

prepare(*key*)

rollback_prepared(*key*)

support_distributed_transaction = True

types = {'reference': 'INT, INDEX %(index_name)s (%(field_name)s), FOREIGN KEY (%(field_name)s) REFERENC

varquote(*name*)

pydal.adapters.oracle module

class pydal.adapters.oracle.**OracleAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)

Bases: pydal.adapters.base.BaseAdapter

LEFT_JOIN()

NOT_NULL(*default*, *field_type*)

RANDOM()

REGEXP(*first*, *second*)

after_connection()

commit_on_alter_table = False

constraint_name(*tablename*, *fieldname*)

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

drivers = ('cx_Oracle',)

execute(*command*, *args=None*)

lastrowid(*table*)

oracle_fix = <_sre.SRE_Pattern object at 0x7f0c9d44cc70>

represent_exceptions(*obj*, *fieldtype*)

select_limitby(*sql_s*, *sql_f*, *sql_t*, *sql_w*, *sql_o*, *limitby*)

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

trigger_name(*tablename*)

types = {'reference': 'NUMBER, CONSTRAINT %(constraint_name)s FOREIGN KEY (%(field_name)s) REFERENC

pydal.adapters.postgres module

class pydal.adapters.postgres.**JDBCPostgreSQLAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)

Bases: pydal.adapters.postgres.PostgreSQLAdapter

REGEX_URI = <_sre.SRE_Pattern object at 0x7f0c9cc13030>

after_connection()

drivers = ('zxJDBC',)

class pydal.adapters.postgres.**NewPostgreSQLAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)

Bases: pydal.adapters.postgres.PostgreSQLAdapter

ANY(*first*)

CONTAINS(*first*, *second*, *case_sensitive=True*)

EQ(*first*, *second=None*)

ILIKE(*first*, *second*)

drivers = ('psycopg2', 'pg8000')

parse_list_integers(*value*, *field_type*)

parse_list_references(*value*, *field_type*)

parse_list_strings(*value*, *field_type*)

represent(*obj*, *fieldtype*)

types = {'reference': 'INTEGER REFERENCES %(foreign_key)s ON DELETE %(on_delete_action)s', 'text': 'TEXT'

class pydal.adapters.postgres.**PostgreSQLAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)

Bases: pydal.adapters.base.BaseAdapter

ADD(*first*, *second*)

ILIKE(*first*, *second*)

LIKE(*first*, *second*)

QUOTE_TEMPLATE = '"%s"'

RANDOM()

REGEXP(*first*, *second*)

REGEX_URI = <_sre.SRE_Pattern object at 0x2976000>

ST_ASGEOJSON(*first*, *second*)
http://postgis.org/docs/ST_AsGeoJSON.html

ST_ASTEXT(*first*)
http://postgis.org/docs/ST_AsText.html

ST_CONTAINS(*first*, *second*)
http://postgis.org/docs/ST_Contains.html

ST_DISTANCE(*first*, *second*)
http://postgis.org/docs/ST_Distance.html

ST_DWITHIN(*first*, *tup*)
http://postgis.org/docs/ST_DWithin.html

ST_EQUALS(*first*, *second*)
http://postgis.org/docs/ST_Equals.html

ST_INTERSECTS(*first*, *second*)
http://postgis.org/docs/ST_Intersects.html

ST_OVERLAPS(*first*, *second*)
http://postgis.org/docs/ST_Overlaps.html

ST_SIMPLIFY(*first*, *second*)
http://postgis.org/docs/ST_Simplify.html

ST_SIMPLIFYPRESERVETOPOLOGY(*first*, *second*)
http://postgis.org/docs/ST_SimplifyPreserveTopology.html

ST_TOUCHES(*first*, *second*)
http://postgis.org/docs/ST_Touches.html

ST_WITHIN(*first*, *second*)
http://postgis.org/docs/ST_Within.html

ST_X(*first*)
http://postgis.org/docs/ST_X.html

ST_Y(*first*)
http://postgis.org/docs/ST_Y.html

adapt(*obj*)

after_connection()

commit_prepared(*key*)

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

distributed_transaction_begin(*key*)

drivers = ('psycopg2', 'pg8000')

lastrowid(*table=None*)

prepare(*key*)

represent(*obj*, *fieldtype*)

rollback_prepared(*key*)

sequence_name(*table*)

support_distributed_transaction = True

try_json()

types = {'reference': 'INTEGER REFERENCES %(foreign_key)s ON DELETE %(on_delete_action)s', 'text': 'TEXT',

varquote(*name*)

pydal.adapters.sapdb module

class pydal.adapters.sapdb.**SAPDBAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)

Bases: pydal.adapters.base.BaseAdapter

REGEX_URI = <_sre.SRE_Pattern object at 0x2976000>

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

drivers = ('sapdb',)

lastrowid (*table*)

select_limitby (*sql_s*, *sql_f*, *sql_t*, *sql_w*, *sql_o*, *limitby*)

sequence_name (*table*)

support_distributed_transaction = False

types = {'reference': 'INT, FOREIGN KEY (%(field_name)s) REFERENCES %(foreign_key)s ON DELETE %(on_del

pydal.adapters.sqlite module

class pydal.adapters.sqlite.**JDBCSQLiteAdapter** (*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)

Bases: pydal.adapters.sqlite.SQLiteAdapter

after_connection ()

drivers = ('zxJDBC_sqlite',)

execute (*a*)

class pydal.adapters.sqlite.**SQLiteAdapter** (*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)

Bases: pydal.adapters.base.BaseAdapter

EXTRACT (*field*, *what*)

REGEXP (*first*, *second*)

after_connection ()

can_select_for_update = None

delete (*tablename*, *query*)

drivers = ('sqlite2', 'sqlite3')

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

Simulate *SELECT ... FOR UPDATE* with *BEGIN IMMEDIATE TRANSACTION*. Note that the entire database, rather than one record, is locked (it will be locked eventually anyway by the following UPDATE).

static web2py_extract (*lookup*, *s*)

static web2py_regexp (*expression*, *item*)

class pydal.adapters.sqlite.**SpatiaLiteAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *srid=4326*, *after_connection=None*)

Bases: pydal.adapters.sqlite.SQLiteAdapter

ST_ASGEOJSON(*first*, *second*)

ST_ASTEXT(*first*)

ST_CONTAINS(*first*, *second*)

ST_DISTANCE(*first*, *second*)

ST_EQUALS(*first*, *second*)

ST_INTERSECTS(*first*, *second*)

ST_OVERLAPS(*first*, *second*)

ST_SIMPLIFY(*first*, *second*)

ST_TOUCHES(*first*, *second*)

ST_WITHIN(*first*, *second*)

after_connection()

drivers = ('sqlite3', 'sqlite2')

represent(*obj*, *fieldtype*)

types = {'string': 'CHAR(%(length)s)', 'reference': 'INTEGER REFERENCES %(foreign_key)s ON DELETE %(on_c

pydal.adapters.teradata module

class pydal.adapters.teradata.**TeradataAdapter**(*db*, *uri*, *pool_size=0*, *folder=None*, *db_codec='UTF-8'*, *credential_decoder=<function IDENTITY at 0x7f0c9d438f50>*, *driver_args={}*, *adapter_args={}*, *do_connect=True*, *after_connection=None*)

Bases: pydal.adapters.base.BaseAdapter

LEFT_JOIN()

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

drivers = ('pyodbc',)

select_limitby(*sql_s*, *sql_f*, *sql_t*, *sql_w*, *sql_o*, *limitby*)

types = {'reference': 'INT', 'text': 'VARCHAR(2000)', 'float': 'REAL', 'datetime': 'TIMESTAMP', 'bigint': 'BIGINT'

Module contents

pydal.adapters.**GoogleDatastoreAdapter = None**

make the import available for BaseAdapter

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(*item*)

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*)

read(*bytes*)

readline()

static **try_create_web2py_filesystem**(*db*)

web2py_filesystems = set([])

write(*data*)

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

register(*method_name=None*)

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

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

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.**UseDatabaseStoredFile**

file_close(*fileobj*)

file_delete(*filename*)

file_exists(*filename*)

file_open(*filename*, *mode='rb'*, *lock=True*)

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 0x7f0ca0613e00>, <_sre.SRE_Pattern object at 0x7f0ca0613e00>, 'children'), (<_sre.SRE_Pattern object at 0x7f0c9d41b450>, <_sre.SRE_Pattern object at 0x7f0c9d41b450>, 'eet'), (<_sre.SRE_Pattern object at 0x7f0ca05da570>, <_sre.SRE_Pattern object at 0x7f0ca05da570>, 'eeth'), (<_sre.SRE_Pattern object at 0x7f0ca05da630>, <_sre.SRE_Pattern object at 0x7f0ca0613ed0>, 'l\\1aves'), (<_sre.SRE_Pattern object at 0x7f0c9d41b500>, <_sre.SRE_Pattern object at 0x7f0c9d41b500>, 'ses'), (<_sre.SRE_Pattern object at 0x7f0c9d41b5b0>, <_sre.SRE_Pattern object at 0x7f0c9d41b5b0>, 'men'), (<_sre.SRE_Pattern object at 0x7f0c9d41b660>, <_sre.SRE_Pattern object at 0x7f0c9d41b660>, 'ives'), (<_sre.SRE_Pattern object at 0x7f0c9d41b710>, <_sre.SRE_Pattern object at 0x7f0c9d41b710>, 'eaux'), (<_sre.SRE_Pattern object at 0x7f0c9d3be030>, <_sre.SRE_Pattern object at 0x7f0c9d3be030>, 'lves'), (<_sre.SRE_Pattern object at 0x7f0c9d3bf030>, <_sre.SRE_Pattern object at 0x7f0c9db10238>, 'es'), (<_sre.SRE_Pattern object at 0x7f0c9d3b42d0>, <_sre.SRE_Pattern object at 0x7f0c9db10238>, 'es'), (<_sre.SRE_Pattern object at 0x7f0ca05cc8e8>, <_sre.SRE_Pattern object at 0x7f0c9d3a5150>, 'ies'), (<_sre.SRE_Pattern object at 0x7f0c9db10238>, <_sre.SRE_Pattern object at 0x7f0c9db10238>, '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=False*, *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** – delaya 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 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

drop(*mode=''*)

fields

import_from_csv_file(*csvfile*, *id_map=None*, *null='<NULL>'*, *unique='uuid'*, *id_offset=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=None*)

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

update_or_insert(*_key=<function <lambda> at 0x7f0c9d438ed8>*, ***values*)

validate_and_insert(***fields*)

validate_and_update(*_key=<function <lambda> at 0x7f0c9d438ed8>*, ***fields*)

validate_and_update_or_insert(*_key=<function <lambda> at 0x7f0c9d438ed8>*, ***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.**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 at 0x7f0c9d36fe10>**

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

Example

Use as:

```
db.define_table('person',Field('name'),Field('info'))
db.define_table('pet',
    Field('ownedby',db.person),
    Field('name'),Field('info')
)

@request.restful()
def index():
    def GET(*args,**vars):
        patterns = [
            "/friends[person]",
            "/{person.name}/:field",
            "/{person.name}/pets[pet.ownedby]",
            "/{person.name}/pets[pet.ownedby]/{pet.name}",
            "/{person.name}/pets[pet.ownedby]/{pet.name}/:field",
            ("/dogs[pet]", db.pet.info=='dog'),
            ("/dogs[pet]/{pet.name.startswith}", db.pet.info=='dog'),
            ]
        parser = db.parse_as_rest(patterns,args,vars)
        if parser.status == 200:
            return dict(content=parser.response)
        else:
            raise HTTP(parser.status,parser.error)

    def POST(table_name,**vars):
        if table_name == 'person':
            return db.person.validate_and_insert(**vars)
        elif table_name == 'pet':
            return db.pet.validate_and_insert(**vars)
        else:
            raise HTTP(400)
    return locals()
```

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

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

find_or_make_work_folder()

reconnect(*f=None*, *cursor=True*)
Defines: *self.connection* and *self.cursor* (if cursor is True) 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

static set_folder(*folder*)

1.5 pydal.objects module

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

Abstract class for Rows and IterRows

__str__()
Serializes the table into a csv file

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*)

len()

like(*value*, *case_sensitive=True*)

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> at 0x7f0c9d438ed8>*, *required=False*, *requires=<function <lambda> at 0x7f0c9d438ed8>*, *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*)

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.

__iter__()
Iterator over records

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

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*)

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

drop(*mode=''*)

fields

import_from_csv_file(*csvfile*, *id_map=None*, *null='<NULL>'*, *unique='uuid'*, *id_offset=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=None*)

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

update_or_insert(*_key=<function <lambda> at 0x7f0c9d438ed8>*, ***values*)

validate_and_insert(***fields*)

validate_and_update(*_key=<function <lambda> at 0x7f0c9d438ed8>*, ***fields*)

validate_and_update_or_insert(*_key=<function <lambda> at 0x7f0c9d438ed8>*, ***fields*)

with_alias(*alias*)

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

Bases: object

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

1.6 Module contents

p

Symbols

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