Sparrow Documentation

Release 0.1

Evert Heylen

May 16, 2017

Contents

Quick Links:

- Github: https://github.com/evertheylen/Sparrow

Sparrow stands for "Single Page Application Real-time Relational Object Wrapper".

It doesn't have to be used for a SPA, but it has JSON support out of the box to send objects inside messages (I personally prefer websockets).

I could have added an extra 'a' somewhere to signal it is asynchronous.

Contents:

CHAPTER 1

Basic usage

Define some classes and create a `SparrowModel`:

```
from sparrow import *

class User(RTEntity):
    username = Property(str, sql_extra="UNIQUE")
    password = Property(str, constraint = lambda p: len(p) > 8)
    key = UID = KeyProperty()

class Message(Entity):
    msg = Property(str)
    from = Reference(User)
    to = RTReference(User)
    key = MID = KeyProperty()

model = SparrowModel(ioloop, {"dbname": "Example"}, [User, Message])

class MyListener:
    # For example, inside a websocket connection
    def new_reference(self, obj, ref_obj):
        # Send the user (registered to this connection) the new message
        self.send(ref_obj, ref_obj.to_json())
```

Dependencies

Sparrow depends on `psycopg2` and `momoko`. The examples may use Tornado for an ioloop, but this is not directly required. However, `momoko` depends on Tornado.

Changing the underlying database library (and async wrapper) should not be too difficult.

CHAPTER 2

Model

The model binds everything together (and might form the actual model of your MVC app). You don't *have* to use this, but it is highly encouraged.

Reference

class sparrow.model.**SparrowModel**(*ioloop*, *db_args*, *classes*, *debug=True*, *db=None*, *set_global_db=False*)

Bases: object

The central class that keeps everything together.

add_sql_statement(*stat: sparrow.sql.Sql*)
Add an *Sql* that you want printed on *info()*.

all_sql_statements()
Return all *Sql* statements that have been added or are automatically generated.

install()
Set up database, only once for each "install" of the model

json_info()
Print all JSON info (as organized as possible). Send this to frontend devs.

sql_for_class(*cls*)

sql_info()
Print all SQL statements (as organized as possible).

uninstall()
Very brutal operation, drops all tables.

sparrow.model.**indent**(*s*, *i=4*, *code=True*)

sparrow.model.**inline**(*s*)

CHAPTER 3

Entities

How? Simple, inherit from Entity and define some properties. Define a property with its type and possibly an extra string that gets included in the 'CREATE TABLE' statement. Example:

```
class User(Entity):
    name = Property(str)
    mail = Property(str, sql_extra="UNIQUE")
```

(In what follows I will often skip properties that are not essential to exemplify the topic at hand.)

Keys

If you try the above example, sparrow will complain, because there is no key. An Entity class can only have 1 key, and you define it with `key`. So if we for example want to have a `UID` attribute and define it as key, we would do:

```
class User(Entity):
    name = Property(str)
    UID = Property(int)
    key = Key(UID)
```

However, this would be irritating. We would have to come up with a unique UID every time we create a User. To solve this problem, we have a `KeyProperty`. This will use Postgres' `SERIAL` type to create the property. Use it like this:

```
class User(Entity):
    name = Property(str)
    UID = KeyProperty()
    key = UID
```

We can shorten this even further:

```
class User(Entity):
    name = Property(str)
    key = UID = KeyProperty()
```

Note that for a `KeyProperty`, its value will be `None` until you insert the object into the database (which will provide the value for it).

A `Key` can also contain multiple properties:

```
class User(Entity):
    firstname = Property(str)
    lastname = Property(str)
    key = Key(firstname, lastname)
```

You can always use a `key` attribute of an object as if it was an entity:

```
u = User(...)
u.key = ("Evert", "Heylen")
```

In case of multiple properties, you need to put it in a tuple. Otherwise (also in the case of a `KeyProperty`, it has to be a simple value.

Constraints

There are two types of constraints: constraints for properties and object-wide constraints. The latter is only checked when calling `__init__`, `update` and `insert`. An example:

```
class User(Entity):
    name = Property(str)
    password = Property(str, constraint=lambda p: len(p) >= 8)  # Password of minimum length

    constraint = lambda u: u.name != u.password  # Don't use your name as password
```

References

Often, you want to keep a reference to other objects. In sparrow, you use a `Reference` to do so. A Reference will automatically create properties to fully save a key of another Entity class. Note that a `Reference` can not be constrained, but it can be used in a `Key`.:

```
class User(Entity):
    firstname = Property(str)
    lastname = Property(str)
    key = Key(firstname, lastname)

class Message(Entity):
    msg = Property(str)
    to = Reference(User)
    from = Reference(User)
```

In this case, the table created for `Message` will have 5 attributes: `msg`, `to_firstname`, `to_lastname`, `from_firstname` and `from_lastname`. It will also be constrained (in the DB) so so that always refers to a `User` in the database. However, you should not set these attributes directly, you should rather use `to` and `from` as if it were properties.:

```
>>> msg = Message(msg="Test", to=some_user.key, from=other_user.key)
>>> # Or after initializing
>>> msg.to = another_user.key
```

Remember to always refer to the key of an object, not the object itself.

JSON

To get a JSON representation, simply call `obj.to_json()`. Some options are available to change this output. You can override `json_repr`, which has to return some datatype that is convertible to JSON. By default, it returns a dictionary of all properties. This too you can control:

```
class User(Entity):
    name = Property(str)
    password = Property(str, json=False)  # We don't want to send passwords
```

Real-time

Sparrow has excellent support for real-time updates. Or you could call it live updates but 'spalrow' is not a word. Anyway, in its simplest form, just inherit from `RTEntity` instead of `Entity`. This will allow you to call `add_listener` (and `remove_listener`) on the object:

```
class User(RTEntity):
    name = Property(str)
    key = UID = KeyProperty()
```

Whenever `update` or `delete` is called, all listeners will get notified of this.

A `RTEntity` gets an extra method `send_update` which will trigger all listeners to be notified of an update without actually writing to the database.

Real-time references

The real fancy stuff is a `RTReference` though. This will make sure that whenever some object refers to another object, it will automatically call `add_reference` on all listeners of the referencing object. For example, with a few modifications we can add live messaging to our `Message` class of before:

```
class Message(Entity):
    msg = Property(str)
    to = RTReference(User)   #
    from = Reference(User)   # Assuming the sender knows it has sent a message, it doesn't
                             # need to know it has sent a message again.
```

A `RTReference` requires a `RTEntity` as referencing class.

Both `RTReference` and `RTEntity` add some overhead, so only use it when necessary.

The listeners need to following a certain interface, more info about that in `RTEntity`.

Database

More info about this can be found in the docs for the file `sql.py`. However, some `Entity`-specific things are not explained there. A list of the possibilities

- **Classmethods (where `Cls` is a subclass of `Entity`):**
  - `Cls.get(Cls.prop1 == val, Cls.prop2 <= Cls.prop3)`: returns a `SELECT` query.
  - `Cls.raw("SELECT * FROM ...")`: returns whatever query you want.
  - `obj = await Cls.find_by_key(some_key, db)`: returns **an instance** with that key.
- **Methods (where `obj` is an instance of a subclass of `Entity`):**
  - `await obj.insert(db)`: inserts the object in the database. Will also fill in the `KeyProperty` if it exists.
  - `await obj.update(db)`: update in the database.

- `await obj.delete(db)`: delete from the database.

Caching

It's there, and it's magic. All objects in memory with the same key will, in fact, be the exact same object (I'm talking about `is` equality, not == equality). It will regulate itself and you shouldn't really care about it. However, I'd like to mention that if an object is in the cache, it will be crazy fast to call `find_by_key`, as it will not use the database at all.

Reference

exception `sparrow.entity.`**`CantSetProperty`**(*obj*, *propnames*)
Bases: *sparrow.util.Error*

Raised when trying to set properties you may not edit. Mainly when editing/setting through edit_from_json or Cls(json_dict=...).

`args`

`with_traceback`()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class `sparrow.entity.`**`ConstrainedProperty`**(*typ*, *constraint: function=None*, *sql_extra: str=''*, *required: bool=True*, *json: bool=True*)
Bases: *sparrow.entity.Property*

`sql_def`()

`type_sql_def`()

class `sparrow.entity.`**`Entity`**(**args*, ***kwargs*)
Bases: `object`

Central class for an Entity. WARNING: Be careful with changing the key as it will fuck with caching. Basically, don't do it.

`check`()
Check object-wide constraint.

`constraint` = **None**

`delete`(*db=None*)
Delete object from the database.

`edit_from_json`(*dct: dict*)

classmethod `find_by_key`(*key*, *db: sparrow.sql.Database=None*) → 'cls'
Works different from *get*, as it will immediatly return the object

classmethod `get`(**where_clauses: list*) → sparrow.sql.Sql

`insert`(*db: sparrow.sql.Database=None*, *replace=False*)
Insert in database.

`json_repr`() → dict
Returns a dictionary of all properties that don't contain *json = False*. When overriding this method, you can return anything you want as long as it is convertible to JSON.

classmethod raw(*text: str*, *dct={}*) → sparrow.sql.RawSql
Return a RawSql where the results will be interpreted as objects of *cls*.

to_json() → str

update(*db=None*)
Update object in the database.

class sparrow.entity.**Enum**(**args*)
Bases: *sparrow.entity.Type*

constraint

from_sql(*obj*)

to_sql(*obj: 'self.python_type'*)

class sparrow.entity.**Key**(**props*)
Bases: *sparrow.entity.Queryable*

A reference to other properties that define the key of this object.

referencing_props()

sql_constraint() → str
Returns the SQL needed to define the PRIMARY KEY constraint.

class sparrow.entity.**KeyProperty**
Bases: *sparrow.entity.SingleKey*, *sparrow.entity.Property*

A specifically created property to be used as a key. Type in postgres is SERIAL.

referencing_props()

sql_constraint() → str

sql_def()

type_sql_def()

class sparrow.entity.**List**(*inner_type*)
Bases: *sparrow.entity.Type*

constraint = None

from_sql(*obj*)

to_sql(*obj: 'self.python_type'*, *f=<built-in function repr>*)

class sparrow.entity.**Listener**
Bases: object

Interface for a listener to be used with *RTEntity*. Main use is documentation, not functionality.

Most implementations will want to define a *set* of objects they are listening to (*listenees*).

delete(*obj: sparrow.entity.RTEntity*)
Handle deletions of the object.

new_reference(*obj: sparrow.entity.RTEntity*, *ref_obj: sparrow.entity.Entity*)
Handle a new reference from *ref_obj* to *obj*. *ref_obj* does not have to be a *RTEntity*.

remove_reference(*obj: sparrow.entity.RTEntity*, *ref_obj: sparrow.entity.Entity*)
Handle the removal of a reference from *ref_obj* to *obj*. *ref_obj* does not have to be a *RTEntity*.

update(*obj: sparrow.entity.RTEntity*)
Handle updates to the object.

class sparrow.entity.**MetaEntity**
Bases: type

Metaclass for *Entity*. This does a whole lot of stuff you should not care about as user of this library. If you do want to know how it works I suggest you read the code.

mro() → list
return a type's method resolution order

exception sparrow.entity.**ObjectConstraintFail**(*obj*)
Bases: *sparrow.util.Error*

Raised when an object failed to follow its constraint.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class sparrow.entity.**Property**(*typ*, *constraint: function=None*, *sql_extra: str=''*, *required: bool=True*, *json: bool=True*)
Bases: *sparrow.entity.Queryable*

sql_def()

type_sql_def()

exception sparrow.entity.**PropertyConstraintFail**(*obj*, *prop*)
Bases: *sparrow.util.Error*

Raised when a property failed to follow its constraint.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class sparrow.entity.**Queryable**
Bases: object

class sparrow.entity.**RTEntity**(**args*, ***kwargs*)
Bases: *sparrow.entity.Entity*

Subclass of Entity that sends live updates! Listeners should follow the interface of *Listener*.

add_listener(*l: 'Listener'*)
Add listeners to this object.

check()
Check object-wide constraint.

constraint = None

delete(*db=None*)

edit_from_json(*dct: dict*)

find_by_key(*key*, *db: sparrow.sql.Database=None*) → 'cls'
Works different from *get*, as it will immediatly return the object

get(**where_clauses: list*) → sparrow.sql.Sql

insert(*db: sparrow.sql.Database=None*, *replace=False*)
Insert in database.

json_repr() → dict
Returns a dictionary of all properties that don't contain *json = False*. When overriding this method, you can return anything you want as long as it is convertible to JSON.

new_reference(*ref*, *ref_obj*)

raw(*text: str*, *dct={}*) → sparrow.sql.RawSql
Return a RawSql where the results will be interpreted as objects of *cls*.

remove_all_listeners()

remove_listener(*l: 'Listener'*)

remove_reference(*ref*, *ref_obj*)

send_update(*db=None*)

to_json() → str

update(*db: sparrow.sql.Database=None*)

class sparrow.entity.**RTReference**(*ref: 'MetaEntity'*)
Bases: *sparrow.entity.Reference*

Reference that automatically notifies the referencing object.

classmethod single_upgrade()

sql_constraint() → str
Will only generate the SQL constraint. The metaclass will take care of the properties.

class sparrow.entity.**RTSingleReference**(*ref: 'MetaEntity'*)
Bases: *sparrow.entity.RTReference*, *sparrow.entity.SingleReference*

Version of RTReference with only one referencing property. (Don't directly use this, it will be automatic.)

single_upgrade()

sql_constraint() → str
Will only generate the SQL constraint. The metaclass will take care of the properties.

class sparrow.entity.**Reference**(*ref: 'MetaEntity'*, *json=True*, *cascade=True*)
Bases: *sparrow.entity.Queryable*

A reference to another Entity type.

classmethod single_upgrade()

sql_constraint() → str
Will only generate the SQL constraint. The metaclass will take care of the properties.

class sparrow.entity.**SingleKey**(**props*)
Bases: *sparrow.entity.Key*

Version of Key with only one property. (Don't directly use this, it will be automatic.)

referencing_props()

sql_constraint() → str
Returns the SQL needed to define the PRIMARY KEY constraint.

class sparrow.entity.**SingleReference**(*ref: 'MetaEntity'*, *json=True*, *cascade=True*)
Bases: *sparrow.entity.Reference*

Version of Reference with only one referencing property. (Don't directly use this, it will be automatic.)

single_upgrade()

sql_constraint() → str
Will only generate the SQL constraint. The metaclass will take care of the properties.

class sparrow.entity.**StaticType**(*python_type*, *sql_type=None*)
Bases: *sparrow.entity.Type*

constraint = None

from_sql(*obj*)

to_sql(*obj: 'self.python_type'*)

class sparrow.entity.**Type**(*python_type*, *sql_type=None*)
Bases: object

constraint = None

from_sql(*obj*)

to_sql(*obj: 'self.python_type'*)

sparrow.entity.**classitems**(*dct*, *bases*)
Helper function to allow for inheritance

sparrow.entity.**create_order**(*op*)

sparrow.entity.**create_where_comparison**(*op*)

Database and Queries

Example:

```
res = await User.get(User.mail == Unsafe(usermail)).exec(db)
u = res.single()
```

Shorter example:

```
u = await User.get(User.mail == Unsafe(usermail)).single(db)
```

For preprocessing a query, you can translate it into `RawSql`, this is faster:

```
users_query = User.get(User.mail == Field("mail")).to_raw()
users = await users_query.with_data(mail = usermail).all(db)
```

Raw requests are also possible:

```
query = User.raw("SELECT * FROM table_User WHERE UID = %(name)s")
```

Or if you don't like pyformat:

```
query = User.raw("SELECT * FROM table_User WHERE UID = {}".format(Field("name")))
u = await query.with_data(name = "evert").single(db)
```

You don't have to mention a class:

```
query = RawSql("SELECT * FROM users WHERE name = 'Evert'")
```

More general and safer example:

```
query = RawSql("SELECT * FROM users WHERE name = %(name)s", {"name": some_user_data})
```

Reference

class sparrow.sql.**And**(**conds*)

Bases: *sparrow.sql.MultiCondition*

all()

Returns all objects in the query.

Wrapped version, first argument is the database.

amount(*i: int*)

Returns a given number of objects in the query.

Wrapped version, first argument is the database.

check(*what*)
Helper function to *parse* parts of a query and insert their data in *self.data*. Handles *Unsafe*, *Field*, other *Sql* instances and tuples. It will simply return all others types.

cls = None

copy()
Create a copy of the statement, by default uses *copy.deepcopy*.

count()
Return the number of rows found in the query.

Wrapped version, first argument is the database.

exec(*db: sparrow.sql.Database=None*)
Execute the SQL statement on the given database.

raw()

Returns the raw (single) result, without any interpreting. If you want to do more than getting a single raw value, consider accessing self.cursor directly.

Wrapped version, first argument is the database.

raw_all()
Returns all raw values.

Wrapped version, first argument is the database.

single()
Returns a single object (and raises NotSingle if there is not only one.

Wrapped version, first argument is the database.

to_raw()
Compile this to a *RawSql* instance for more performance!

with_data(***kwargs*)
This function creates a copy of the statement with added data, passed as keyword arguments.

class sparrow.sql.**ClassedSql**(*cls: type*, *data={}*)
Bases: *sparrow.sql.Sql*

Version of *Sql* that also saves a given class. *SqlResult* will later try to parse its result as instances of this class.

all()
Returns all objects in the query.

Wrapped version, first argument is the database.

amount(*i: int*)
Returns a given number of objects in the query.

Wrapped version, first argument is the database.

check(*what*)
Helper function to *parse* parts of a query and insert their data in *self.data*. Handles *Unsafe*, *Field*, other *Sql* instances and tuples. It will simply return all others types.

cls = None

copy()
Create a copy of the statement, by default uses *copy.deepcopy*.

count()
Return the number of rows found in the query.

Wrapped version, first argument is the database.

exec(*db: sparrow.sql.Database=None*)
Execute the SQL statement on the given database.

raw()

Returns the raw (single) result, without any interpreting. If you want to do more than getting a single raw value, consider accessing self.cursor directly.

Wrapped version, first argument is the database.

raw_all()
Returns all raw values.

Wrapped version, first argument is the database.

single()
Returns a single object (and raises NotSingle if there is not only one.

Wrapped version, first argument is the database.

to_raw()

with_data(***kwargs*)
This function creates a copy of the statement with added data, passed as keyword arguments.

class sparrow.sql.**Command**(*cls: type*, *data={}*)
Bases: *sparrow.sql.ClassedSql*

For INSERT, DELETE, UPDATE, CREATE TABLE, DROP TABLE, ... statements.

all()
Returns all objects in the query.

Wrapped version, first argument is the database.

amount(*i: int*)
Returns a given number of objects in the query.

Wrapped version, first argument is the database.

check(*what*)
Helper function to *parse* parts of a query and insert their data in *self.data*. Handles *Unsafe*, *Field*, other *Sql* instances and tuples. It will simply return all others types.

cls = None

copy()
Create a copy of the statement, by default uses *copy.deepcopy*.

count()
Return the number of rows found in the query.

Wrapped version, first argument is the database.

exec(*db: sparrow.sql.Database=None*)
Execute the SQL statement on the given database.

raw()

Returns the raw (single) result, without any interpreting. If you want to do more than getting a single raw value, consider accessing self.cursor directly.

Wrapped version, first argument is the database.

raw_all()
Returns all raw values.

Wrapped version, first argument is the database.

single()
Returns a single object (and raises NotSingle if there is not only one.

Wrapped version, first argument is the database.

to_raw()

with_data(***kwargs*)
This function creates a copy of the statement with added data, passed as keyword arguments.

class sparrow.sql.**Condition**(*data={}*)
Bases: *sparrow.sql.Sql*

all()
Returns all objects in the query.

Wrapped version, first argument is the database.

amount(*i: int*)
Returns a given number of objects in the query.

Wrapped version, first argument is the database.

check(*what*)
Helper function to *parse* parts of a query and insert their data in *self.data*. Handles *Unsafe*, *Field*, other *Sql* instances and tuples. It will simply return all others types.

cls = None

copy()
Create a copy of the statement, by default uses *copy.deepcopy*.

count()
Return the number of rows found in the query.

Wrapped version, first argument is the database.

exec(*db: sparrow.sql.Database=None*)
Execute the SQL statement on the given database.

raw()

Returns the raw (single) result, without any interpreting. If you want to do more than getting a single raw value, consider accessing self.cursor directly.

Wrapped version, first argument is the database.

raw_all()
Returns all raw values.

Wrapped version, first argument is the database.

single()
Returns a single object (and raises NotSingle if there is not only one.

Wrapped version, first argument is the database.

to_raw()
Compile this to a *RawSql* instance for more performance!

with_data(***kwargs*)
This function creates a copy of the statement with added data, passed as keyword arguments.

class sparrow.sql.**CreateTable**(*cls*)
Bases: *sparrow.sql.Command*

For CREATE TABLE statements.

all()
Returns all objects in the query.

Wrapped version, first argument is the database.

amount(*i: int*)
Returns a given number of objects in the query.

Wrapped version, first argument is the database.

check(*what*)
Helper function to *parse* parts of a query and insert their data in *self.data*. Handles *Unsafe*, *Field*, other *Sql* instances and tuples. It will simply return all others types.

cls = None

copy()
Create a copy of the statement, by default uses *copy.deepcopy*.

count()
Return the number of rows found in the query.

Wrapped version, first argument is the database.

exec(*db: sparrow.sql.Database=None*)
Execute the SQL statement on the given database.

raw()

Returns the raw (single) result, without any interpreting. If you want to do more than getting a single raw value, consider accessing self.cursor directly.

Wrapped version, first argument is the database.

raw_all()
Returns all raw values.

Wrapped version, first argument is the database.

single()
Returns a single object (and raises NotSingle if there is not only one.

Wrapped version, first argument is the database.

to_raw()

with_data(***kwargs*)
This function creates a copy of the statement with added data, passed as keyword arguments.

class sparrow.sql.**Database**(*ioloop*, *dbname*, *user='postgres'*, *password='postgres'*, *host='localhost'*, *port=5432*, *momoko_poolsize=5*)
Bases: object

Class for Postgres database.

get_cursor(*statement: 'Sql'*, *unsafe_dict: dict*)

class sparrow.sql.**Delete**(*what*)
Bases: *sparrow.sql.Command*

all()
Returns all objects in the query.

Wrapped version, first argument is the database.

amount(*i: int*)
Returns a given number of objects in the query.

Wrapped version, first argument is the database.

check(*what*)
Helper function to *parse* parts of a query and insert their data in *self.data*. Handles *Unsafe*, *Field*, other *Sql* instances and tuples. It will simply return all others types.

cls = None

copy()
Create a copy of the statement, by default uses *copy.deepcopy*.

count()
Return the number of rows found in the query.

Wrapped version, first argument is the database.

exec(*db: sparrow.sql.Database=None*)
Execute the SQL statement on the given database.

raw()

Returns the raw (single) result, without any interpreting. If you want to do more than getting a single raw value, consider accessing self.cursor directly.

Wrapped version, first argument is the database.

raw_all()
Returns all raw values.

Wrapped version, first argument is the database.

single()
Returns a single object (and raises NotSingle if there is not only one.

Wrapped version, first argument is the database.

to_raw()

with_data(***kwargs*)
This function creates a copy of the statement with added data, passed as keyword arguments.

class sparrow.sql.**DropTable**(*cls*)
Bases: *sparrow.sql.Command*

For DROP TABLE statements.

all()
Returns all objects in the query.

Wrapped version, first argument is the database.

amount(*i: int*)
Returns a given number of objects in the query.

Wrapped version, first argument is the database.

check(*what*)
Helper function to *parse* parts of a query and insert their data in *self.data*. Handles *Unsafe*, *Field*, other *Sql* instances and tuples. It will simply return all others types.

cls = None

copy()
Create a copy of the statement, by default uses *copy.deepcopy*.

count()
Return the number of rows found in the query.

Wrapped version, first argument is the database.

exec(*db: sparrow.sql.Database=None*)
Execute the SQL statement on the given database.

raw()

Returns the raw (single) result, without any interpreting. If you want to do more than getting a single raw value, consider accessing self.cursor directly.

Wrapped version, first argument is the database.

raw_all()
Returns all raw values.

Wrapped version, first argument is the database.

single()
Returns a single object (and raises NotSingle if there is not only one.

Wrapped version, first argument is the database.

to_raw()

with_data(***kwargs*)
This function creates a copy of the statement with added data, passed as keyword arguments.

class sparrow.sql.**EntityCommand**(*cls: type*, *data={}*)
Bases: *sparrow.sql.Command*

Class for commands that work for both classes (will require inserting data later on) as objects.

all()
Returns all objects in the query.

Wrapped version, first argument is the database.

amount(*i: int*)
Returns a given number of objects in the query.

Wrapped version, first argument is the database.

check(*what*)
Helper function to *parse* parts of a query and insert their data in *self.data*. Handles *Unsafe*, *Field*, other *Sql* instances and tuples. It will simply return all others types.

cls = None

copy()
Create a copy of the statement, by default uses *copy.deepcopy*.

count()
Return the number of rows found in the query.

Wrapped version, first argument is the database.

exec(*db: sparrow.sql.Database=None*)
Execute the SQL statement on the given database.

raw()

Returns the raw (single) result, without any interpreting. If you want to do more than getting a single raw value, consider accessing self.cursor directly.

Wrapped version, first argument is the database.

raw_all()
Returns all raw values.

Wrapped version, first argument is the database.

single()
Returns a single object (and raises NotSingle if there is not only one.

Wrapped version, first argument is the database.

to_raw()

with_data(***kwargs*)
This function creates a copy of the statement with added data, passed as keyword arguments.

class sparrow.sql.**Field**(*name*)
Bases: object

Wrapper for data that is to be inserted into the query (with Sql.with_data) later on.

class sparrow.sql.**GlobalDb**
Bases: object

db = None

classmethod get()

classmethod globalize(*db*)

classmethod set(*db*)

class sparrow.sql.**Insert**(*what*, *returning=None*, *replace=False*)
Bases: *sparrow.sql.EntityCommand*

For INSERT statements.

all()
Returns all objects in the query.

Wrapped version, first argument is the database.

amount(*i: int*)
Returns a given number of objects in the query.

Wrapped version, first argument is the database.

check(*what*)
Helper function to *parse* parts of a query and insert their data in *self.data*. Handles *Unsafe*, *Field*, other *Sql* instances and tuples. It will simply return all others types.

cls = None

copy()
Create a copy of the statement, by default uses *copy.deepcopy*.

count()
Return the number of rows found in the query.

Wrapped version, first argument is the database.

exec(*db: sparrow.sql.Database=None*)
Execute the SQL statement on the given database.

raw()

Returns the raw (single) result, without any interpreting. If you want to do more than getting a single raw value, consider accessing self.cursor directly.

Wrapped version, first argument is the database.

raw_all()
Returns all raw values.

Wrapped version, first argument is the database.

returning(*prop*)
Set what the database should return. Can be chained.

single()
Returns a single object (and raises NotSingle if there is not only one.

Wrapped version, first argument is the database.

to_raw()

with_data(***kwargs*)
This function creates a copy of the statement with added data, passed as keyword arguments.

class sparrow.sql.**MultiCondition**(**conds*)
Bases: *sparrow.sql.Condition*

all()
Returns all objects in the query.

Wrapped version, first argument is the database.

amount(*i: int*)
Returns a given number of objects in the query.

Wrapped version, first argument is the database.

check(*what*)
Helper function to *parse* parts of a query and insert their data in *self.data*. Handles *Unsafe*, *Field*, other *Sql* instances and tuples. It will simply return all others types.

cls = None

copy()
Create a copy of the statement, by default uses *copy.deepcopy*.

count()
Return the number of rows found in the query.

Wrapped version, first argument is the database.

exec(*db: sparrow.sql.Database=None*)
Execute the SQL statement on the given database.

raw()

Returns the raw (single) result, without any interpreting. If you want to do more than getting a single raw value, consider accessing self.cursor directly.

Wrapped version, first argument is the database.

raw_all()
Returns all raw values.

Wrapped version, first argument is the database.

single()
Returns a single object (and raises NotSingle if there is not only one.

Wrapped version, first argument is the database.

to_raw()
Compile this to a *RawSql* instance for more performance!

with_data(***kwargs*)
This function creates a copy of the statement with added data, passed as keyword arguments.

class sparrow.sql.**Not**(*cond*)
Bases: *sparrow.sql.Condition*

all()
Returns all objects in the query.

Wrapped version, first argument is the database.

amount(*i: int*)
Returns a given number of objects in the query.

Wrapped version, first argument is the database.

check(*what*)
Helper function to *parse* parts of a query and insert their data in *self.data*. Handles *Unsafe*, *Field*, other *Sql* instances and tuples. It will simply return all others types.

cls = None

copy()
Create a copy of the statement, by default uses *copy.deepcopy*.

count()
Return the number of rows found in the query.

Wrapped version, first argument is the database.

exec(*db: sparrow.sql.Database=None*)
Execute the SQL statement on the given database.

raw()

Returns the raw (single) result, without any interpreting. If you want to do more than getting a single raw value, consider accessing self.cursor directly.

Wrapped version, first argument is the database.

raw_all()
Returns all raw values.

Wrapped version, first argument is the database.

single()
Returns a single object (and raises NotSingle if there is not only one.

Wrapped version, first argument is the database.

to_raw()
Compile this to a *RawSql* instance for more performance!

with_data(***kwargs*)
This function creates a copy of the statement with added data, passed as keyword arguments.

exception sparrow.sql.**NotSingle**
Bases: *sparrow.util.Error*

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class sparrow.sql.**Or**(**conds*)
Bases: *sparrow.sql.MultiCondition*

all()
Returns all objects in the query.

Wrapped version, first argument is the database.

amount(*i: int*)
Returns a given number of objects in the query.

Wrapped version, first argument is the database.

check(*what*)
Helper function to *parse* parts of a query and insert their data in *self.data*. Handles *Unsafe*, *Field*, other *Sql* instances and tuples. It will simply return all others types.

cls = None

copy()
Create a copy of the statement, by default uses *copy.deepcopy*.

count()
Return the number of rows found in the query.

Wrapped version, first argument is the database.

exec(*db: sparrow.sql.Database=None*)
Execute the SQL statement on the given database.

raw()

Returns the raw (single) result, without any interpreting. If you want to do more than getting a single raw value, consider accessing self.cursor directly.

Wrapped version, first argument is the database.

raw_all()
Returns all raw values.

Wrapped version, first argument is the database.

single()
Returns a single object (and raises NotSingle if there is not only one.

Wrapped version, first argument is the database.

to_raw()
Compile this to a *RawSql* instance for more performance!

with_data(***kwargs*)
This function creates a copy of the statement with added data, passed as keyword arguments.

class sparrow.sql.**Order**(*field*, *op: str*, *data={}*)
Bases: *sparrow.sql.Sql*

Encodes an 'ORDER BY' clause.

all()
Returns all objects in the query.

Wrapped version, first argument is the database.

amount(*i: int*)
Returns a given number of objects in the query.

Wrapped version, first argument is the database.

check(*what*)
Helper function to *parse* parts of a query and insert their data in *self.data*. Handles *Unsafe*, *Field*, other *Sql* instances and tuples. It will simply return all others types.

cls = None

copy()
Create a copy of the statement, by default uses *copy.deepcopy*.

count()
Return the number of rows found in the query.

Wrapped version, first argument is the database.

exec(*db: sparrow.sql.Database=None*)
Execute the SQL statement on the given database.

raw()

Returns the raw (single) result, without any interpreting. If you want to do more than getting a single raw value, consider accessing self.cursor directly.

Wrapped version, first argument is the database.

raw_all()
Returns all raw values.

Wrapped version, first argument is the database.

single()
Returns a single object (and raises NotSingle if there is not only one.

Wrapped version, first argument is the database.

to_raw()
Compile this to a *RawSql* instance for more performance!

with_data(***kwargs*)
This function creates a copy of the statement with added data, passed as keyword arguments.

class sparrow.sql.**RawClassedSql**(*cls*, *text*, *data={}*)
Bases: *sparrow.sql.RawSql*, *sparrow.sql.ClassedSql*

Version of *RawSql* that also saves a given class like *ClassedSql*.

all()
Returns all objects in the query.

Wrapped version, first argument is the database.

amount(*i: int*)
Returns a given number of objects in the query.

Wrapped version, first argument is the database.

check(*what*)
Helper function to *parse* parts of a query and insert their data in *self.data*. Handles *Unsafe*, *Field*, other *Sql* instances and tuples. It will simply return all others types.

cls = None

copy()

count()
Return the number of rows found in the query.

Wrapped version, first argument is the database.

exec(*db: sparrow.sql.Database=None*)
Execute the SQL statement on the given database.

raw()

Returns the raw (single) result, without any interpreting. If you want to do more than getting a single raw value, consider accessing self.cursor directly.

Wrapped version, first argument is the database.

raw_all()
Returns all raw values.

Wrapped version, first argument is the database.

single()
Returns a single object (and raises NotSingle if there is not only one.

Wrapped version, first argument is the database.

to_raw()
Already raw, just return self.

with_data(***kwargs*)
This function creates a copy of the statement with added data, passed as keyword arguments.

class sparrow.sql.**RawSql**(*text*, *data={}*)
Bases: *sparrow.sql.Sql*

Simply saves a string, and also some data. This is in contrast with *Sql*, which may save the query in a more abstract way.

all()
Returns all objects in the query.

Wrapped version, first argument is the database.

amount(*i: int*)
Returns a given number of objects in the query.

Wrapped version, first argument is the database.

check(*what*)
Helper function to *parse* parts of a query and insert their data in *self.data*. Handles *Unsafe*, *Field*, other *Sql* instances and tuples. It will simply return all others types.

cls = None

copy()
More optimized version of copy.

count()
Return the number of rows found in the query.

Wrapped version, first argument is the database.

exec(*db: sparrow.sql.Database=None*)
Execute the SQL statement on the given database.

raw()

Returns the raw (single) result, without any interpreting. If you want to do more than getting a single raw value, consider accessing self.cursor directly.

Wrapped version, first argument is the database.

raw_all()
Returns all raw values.

Wrapped version, first argument is the database.

single()
Returns a single object (and raises NotSingle if there is not only one.

Wrapped version, first argument is the database.

to_raw()
Already raw, just return self.

with_data(***kwargs*)
This function creates a copy of the statement with added data, passed as keyword arguments.

class sparrow.sql.**Select**(*cls*, *where_clauses=[]*, *order: sparrow.sql.Order=None*, *offset=None*, *limit=None*)
Bases: *sparrow.sql.ClassedSql*

Encodes a 'SELECT' query.

all()
Returns all objects in the query.

Wrapped version, first argument is the database.

amount(*i: int*)
Returns a given number of objects in the query.

Wrapped version, first argument is the database.

check(*what*)
Helper function to *parse* parts of a query and insert their data in *self.data*. Handles *Unsafe*, *Field*, other *Sql* instances and tuples. It will simply return all others types.

cls = None

copy()
Create a copy of the statement, by default uses *copy.deepcopy*.

count()
Return the number of rows found in the query.

Wrapped version, first argument is the database.

exec(*db: sparrow.sql.Database=None*)
Execute the SQL statement on the given database.

limit(*l*)
'LIMIT' the query. Can be used for chaining.

offset(*o*)
'OFFSET' the query. Can be used for chaining.

order(*_order: sparrow.sql.Order*)
'ORDER' the query. Can be used for chaining.

raw()

Returns the raw (single) result, without any interpreting. If you want to do more than getting a single raw value, consider accessing self.cursor directly.

Wrapped version, first argument is the database.

raw_all()
Returns all raw values.

Wrapped version, first argument is the database.

single()
Returns a single object (and raises NotSingle if there is not only one.

Wrapped version, first argument is the database.

to_raw()

where(**clauses*)
'OFFSET' the query. Can be used for chaining.

Parameters: a number of Where clauses.

with_data(***kwargs*)
This function creates a copy of the statement with added data, passed as keyword arguments.

class sparrow.sql.**Sql**(*data={}*)
Bases: object

Main class to save a given SQL query/command. Do not use directly, use the subclasses.

all()
Returns all objects in the query.

Wrapped version, first argument is the database.

amount(*i: int*)
Returns a given number of objects in the query.

Wrapped version, first argument is the database.

check(*what*)
Helper function to *parse* parts of a query and insert their data in *self.data*. Handles *Unsafe*, *Field*, other *Sql* instances and tuples. It will simply return all others types.

cls = None

copy()
Create a copy of the statement, by default uses *copy.deepcopy*.

count()
Return the number of rows found in the query.

Wrapped version, first argument is the database.

exec(*db: sparrow.sql.Database=None*)
Execute the SQL statement on the given database.

raw()

Returns the raw (single) result, without any interpreting. If you want to do more than getting a single raw value, consider accessing self.cursor directly.

Wrapped version, first argument is the database.

raw_all()
Returns all raw values.

Wrapped version, first argument is the database.

single()
Returns a single object (and raises NotSingle if there is not only one.

Wrapped version, first argument is the database.

to_raw()
Compile this to a *RawSql* instance for more performance!

with_data(***kwargs*)
This function creates a copy of the statement with added data, passed as keyword arguments.

exception sparrow.sql.**SqlError**(*err: psycopg2.Error*, *query: 'Sql'*, *data: dict*)
Bases: *sparrow.util.Error*

Exception raised while executing a query (or command). Wraps a psycopg2 error to also include the query that went wrong.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class sparrow.sql.**SqlResult**(*cursor*, *query: 'Sql'*)
Bases: object

Class wrapping a database cursor. Note that most of the time, you can use the 'easier' methods in Sql. Instead of:

```
>>> res = await User.get(User.name == "Evert").exec(db)
>>> u = res.single()
```

You can do:

```
>>> u = await User.get(User.name == "Evert").single(db)
```

This doesn't work for scrolling and getting raw data.

The methods *single*, *all* and *amount* will try to interpret the result as object(s) of the given class in *self.query.cls*, don't try them if it that class is *None*.

all()
Returns all objects in the query.

amount(*i: int*)
Returns a given number of objects in the query.

count()
Return the number of rows found in the query.

raw()
Returns the raw (single) result, without any interpreting. If you want to do more than getting a single raw value, consider accessing self.cursor directly.

raw_all()
Returns all raw values.

scroll(*i: int*)
Scroll the cursor *i* steps. *i* can be negative. This method is chainable.

single()
Returns a single object (and raises NotSingle if there is not only one.

class sparrow.sql.**Unsafe**(*value*)
Bases: object

Wrapper for unsafe data. (For data that needs to inserted later, use Field.)

class sparrow.sql.**Update**(*what*)
Bases: *sparrow.sql.EntityCommand*

all()
Returns all objects in the query.

Wrapped version, first argument is the database.

amount(*i: int*)
Returns a given number of objects in the query.

Wrapped version, first argument is the database.

check(*what*)
Helper function to *parse* parts of a query and insert their data in *self.data*. Handles *Unsafe*, *Field*, other *Sql* instances and tuples. It will simply return all others types.

cls = None

copy()
Create a copy of the statement, by default uses *copy.deepcopy*.

count()
Return the number of rows found in the query.

Wrapped version, first argument is the database.

exec(*db: sparrow.sql.Database=None*)
Execute the SQL statement on the given database.

raw()

Returns the raw (single) result, without any interpreting. If you want to do more than getting a single raw value, consider accessing self.cursor directly.

Wrapped version, first argument is the database.

raw_all()
Returns all raw values.

Wrapped version, first argument is the database.

single()
Returns a single object (and raises NotSingle if there is not only one.

Wrapped version, first argument is the database.

to_raw()

with_data(***kwargs*)
This function creates a copy of the statement with added data, passed as keyword arguments.

class sparrow.sql.**Where**(*lfield*, *op: str*, *rfield*, *data={}*)
Bases: *sparrow.sql.Condition*

Encodes a single 'WHERE' clause.

all()
Returns all objects in the query.

Wrapped version, first argument is the database.

amount(*i: int*)
Returns a given number of objects in the query.

Wrapped version, first argument is the database.

check(*what*)
Helper function to *parse* parts of a query and insert their data in *self.data*. Handles *Unsafe*, *Field*, other *Sql* instances and tuples. It will simply return all others types.

cls = None

copy()
Create a copy of the statement, by default uses *copy.deepcopy*.

count()
Return the number of rows found in the query.

Wrapped version, first argument is the database.

exec(*db: sparrow.sql.Database=None*)
Execute the SQL statement on the given database.

raw()

Returns the raw (single) result, without any interpreting. If you want to do more than getting a single raw value, consider accessing self.cursor directly.

Wrapped version, first argument is the database.

raw_all()
Returns all raw values.

Wrapped version, first argument is the database.

single()
Returns a single object (and raises NotSingle if there is not only one.

Wrapped version, first argument is the database.

to_raw()
Compile this to a *RawSql* instance for more performance!

with_data(***kwargs*)
This function creates a copy of the statement with added data, passed as keyword arguments.

CHAPTER 5

Utilities etc

Not so much here so far.

Reference

exception sparrow.util.**Error**
Bases: Exception

Base class for all exceptions of Sparrow

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

S

A

C

D

E

F

G

I

J

K

L

M

N

O

P

Q

R

S

T

U

W