drunken boat Documentation

Release 0.0.1

Yohann Gabory

May 15, 2015

Contents

Drunken boat is a performance based webframework under heavy active developpment. It support python2.x and python3.x

It offer Routing, View management and a projection based ORM, schema less and eventualy agnostic.

The ORM is inspired by the POMM project: http://www.pomm-project.org/ by Grégoire HUBERT (https://github.com/chanmix51/Pomm) Many thanks for his thoughts

CHAPTER 1

A simple Hello World

first, install *drunken-boat* see http://drunken_boat.readthedocs.org/en/stable/install.html. Once drunken_boat installed you can boostrap your first application with:

```
drunken_run.py bootstrap example_blog
```

This will create for you all you need to start:

```
cd /home/yohann/Dev/drunken_boat/example_blog
python application.py
```

then visit http://localhost:5000/

1.1 Project Layout

drunken_run.py bootstrap example_blog create a new *example_blog* directory with base file structure to start working:

```
example_blog/
  -- __init__.py
  -- application.py
  -- router.py
  -- views.py
  -- projection.py
  -- config.py
```

content of *application.py*:

```
from drunken_boat import Application
from example_blog.router import MainRouter

app = Application(
        MainRouter("/")
    )

if __name__ == '__main__':
    from werkzeug.serving import run_simple
    run_simple('127.0.0.1', 5000, app, use_debugger=True, use_reloader=True)
```

application only need an Application instance with a *Router* responsible for routing the incomming requests.

content of *router.py*:

```python
from drunken_boat.router import Router
from example_blog.views import MainView


class MainRouter(Router):
    view = MainView
```

a router can be as simple as this one but obviously you can add more endpoints using *Router.patterns*. *Router* can take a *View* attribute to compute the *Response* to return

content of *view.py*:

```python
from drunken_boat.views import View
from werkzeug.wrappers import Response


class MainView(View):
    def get(self, request, **kwargs):
        response = Response('Hello World!', mimetype='text/plain')
        return response
```

Every request on "/" will return a "Hello World!" a lot more can be done in *View* check the documentation on how to manage much more with *MiddleWare*, *Projection* for database access and else.

Continue reading Models: database management the easy way

See http://drunken_boat.readthedocs.org/ for full documentation

CHAPTER 2

Contents:

2.1 Install

Due to the active developpment of the project, no pakages have been uploaded to pip. However you can git pull the main repository:

```
git clone https://github.com/boblefrag/drunken_boat
```

Then *cd* to drunken_boat directory:

```
cd drunken_boat
```

And install with:

```
python setup.py install
```

2.2 ORM

2.2.1 Models: database management the easy way

Basicaly Models are simply a thin layer on top of *Projections* see Database Management for a full documentation on *Projections*. With models, on can easily create, update and delete database objects using a clear syntax.

Of course you also get all the power of *Projections* when you need them.

Initialize

To create a model, you need at least a *Database* and a *Projection* A basic projection can be something like:

```
from drunken_boat.db.postgresql.projections import Projection

class ExampleProjection(Projection):
    """
    A basic projection
    """
    age = Timestamp(db_name="age(birthday)", virtual=True)
    birthday = Timestamp()

    class Meta:
        table = "test"
```

The underlying database table can be something like:

```
Table : test

  id serial PRIMARY KEY,
  num integer NOT NULL,
  data varchar NOT NULL,
  birthday timestamp
```

you can now import your database configuration:

```
from example_blog.config import DATABASE
from drunken_boat.db.postgresql import DB

db = DB(**DATABASE)
```

You are now ready to use the *Models* API:

```
from drunken_boat.db.postgresql.models import Model
my_model = Model(db, projections={"example": ExampleProjection})
```

Manipulation of Model objects

With a *Model* instance, you can request an object for creating data in your database in a very pythonic way. To requert an object, just do:

```
obj = model.object()
```

Then you can set data on this object:

```
obj.num = 10
obj.data = "something"
```

And, of course save this object in the database:

```
obj.save()
```

Once saved, the object can be updated:

```
obj.num = 25
obj.update()
```

Or deleted:

```
obj.delete()
```

How things works

Models are nothing more than just a very thin layer around *Projections* and database objets. When you create a Model with some projections, we will look for a “default” projection. If this projection does not exist, the first projection will be set as the “default” one.

In order to implement update and delete on model object, we look for the primary key on the “default” projection. If the primary key is not on the projection, it will be automaticaly added when you call the save() method of object.

After that, because we get the primary key of your object we are able to update and delete the object using this pimary key.

All the projections you a model contains are available in Model.projections. For example, to get the default projection you just have to write:

```
my_model.projections.default
```

If you do not want to use the default projection when requesting a Model object, you can ask for a particular projection:

```
obj = my_model.object(projection=my_model.projections.other)
```

Using a projection you do not define in your model is absolutly ok because there is no magic in how Model work:

```
obj = my_model.object(projection=AnOtherProjection)
```

The only limit is that your projection **must** be on the same table. (same primary key field)

Returning

Because underlying projections offers returning on insert, update and delete, Model objects offer this behavior too. Simply add the returning argument to your save, update or delete method just like with projections:

```
obj.save(returning="title")
```

2.2.2 Database Management

Drunken Boat is focused on performances. Most of current applications lack performances due to ORM. Yes ORM are great when you need Object Oriented programation but they lack a lot of features you can find in modern database like PostgreSQL.

Drunken Boat want to help you write powerful applications where you can use the most of your database and still use Object Oriented programmation.

This is the reason why Drunken Boat does not force you to create your database nor managing table schema in his ORM. Sure it gives you some helpful methods and functions to create database, schema, make ALTER TABLE on your databases but it's absolutely up to you to manage them the way you like.

Configuration & Table creation

In the project created by drunken_run.py the file config.py contains the base detail of a database connection. Change the DATABASE with connection informations of your database.

Even if drunken_boat don't force you to create table from python, for this tutorial you can use this simple script to generate the table you will use in the next step:

```
#projection.py
from drunken_boat.db.postgresql import DB
from example_blog.config import DATABASE

def create_tables():
    db = DB(**DATABASE)
    cur = db.cursor()
    cur.execute(
        """select exists(
            select * from information_schema.tables where table_name=%s)
        """,
        ('test',))
    if not cur.fetchone()[0]:
        cur.execute("""CREATE TABLE test (
        id serial PRIMARY KEY,
        num integer,
        data varchar,
        birthday timestamp)""");
        db.conn.commit()
        print("table created")
        return
    print("table already exists")
```

Projections

Projections are the object based representation of the *result* of a database query. See them as what you expect from the database.

Let say you make this query:

```
select name, age(birthdate) from user;
```

the corresponding projection will just fit:

```
class UserNameAge(Projection):

    name = CharField()
    age = Timestamp(name="age(birthdate)")

    class Meta:
        table = "user"

projection = UserNameAge(DB(**connection_params))
```

And you can get your results as easily as:

```
>> users = projection.select()
>> users[0].age
datetime.timedelta(13850, 50160)
```

results are list of *DataBaseObject*. because DataBaseObject are objects, you can attach any method you want on it. For example:

```
from drunken_boat.db.postgresql import DB
from config import DATABASE
from drunken_boat.db.postgresql.fields import Timestamp
from drunken_boat.db.postgresql.projections import (Projection,
                                                    DataBaseObject)
class ExampleDataBaseObject(DataBaseObject):

    def display_birthyear_and_days(self):
        days = self.age.days
        year = self.birthdate.year
        return "{} days since {}".format(days, year)

class ExampleProjection(Projection):
    """
    Here you can write your real projections
    """
    age = Timestamp(db_name="age(birthday)", virtual=True)
    birthdate = Timestamp()

    class Meta:
        table = "test"
        database_object = ExampleDataBaseObject

example_projection = ExampleProjection(DB(**DATABASE))


>>> from projections import example_projection
>>> t = example_projection.select()
>>> t[0].display_birthyear_and_days()
'13850 days since 1977'
```

Where

One thing you will surely do very often is to use *Projection* with WHERE clause. Where clause are defined with 2 sides. First side is the clause and the comparison operator, the other side is the parameter.

For example, in the statement:

WHERE id > 4;

id is the clause, > is the comparison operator, and 4 is the parameter.

The first an easier way to make a query with a WHERE clause is simply adding where and parameter to the select statement:

```
>>> projection.select(where='id=%s', params=(1,))
```

If it's perfectly ok to do so, but sometimes you will need to store a WHERE clause to use it in many places in your code. For this the Where object is here to help you.

A where object take a clause, an operator and a value:

```
from drunken_boat.db.postgresql.query import Where
where = Where("id", "=", "%s")
```

As you can see a Where object is very similar to the select version. The difference is that you do not define a parameter yet. The parameter will be define when calling the select method of your *Projection*:

```
>>> projection.select(where=where, params=(1,))
```

Multiple Where

It's also possible to use multiple where in a single select using biwise operations. AND, OR and NOT are supported:

AND:

```
>>> where = Where("id", "=", "%s") & Where("title", "=", "%s")
```

OR:

```
>>> where = Where("id", "=", "%s") | Where("title", "=", "%s")
```

NOT:

```
>>> where = Where("id", "=", "%s") & ~Where("title", "=", "%s")
```

NOT can be used as is to make exclude clause:

```
>>> where = ~Where("title", "=", "%s")
```

You can also define priorities with parenthesis:

```
>>> where = Where("id", "=", "%s") | (Where("title", "=", "%s") & Where("intro", "=", "%s"))
```

this will be rendered as:

```
id = %s OR (title = %s AND intro = %s)
```

Insert

Even if you do not describe the table schema of your tables, drunken_boat introspect your table schema to give you automatic validation of data before even hitting the database.

To demonstrate this behavior let's create another table:

```
Table : test

  id serial PRIMARY KEY,
  num integer NOT NULL,
  data varchar NOT NULL,
  birthday timestamp
```

And another projection:

```
class ExampleProjection(Projection):
    """
    Here you can write your real projections
    """
    age = Timestamp(db_name="age(birthday)", virtual=True)
    birthday = Timestamp()

    class Meta:
        table = "test"
        database_object = ExampleDataBaseObject

example_projection = ExampleProjection(DB(**DATABASE))
```

Now, with a shell try to insert some data in the table:

```
>>> from projections import example_projection
>>> example_projection.insert({"birthday": datetime.datetime.now()})
ValueError: num of type integer is required
data of type character varying is required
```

Now that you know wich data you must use to insert data you can type:

```
>>> example_projection.insert({"num": 10,
...                            "data": "some data"})
```

You can check that your record is saved in the database:

```
>>> example_projection.select()
... [<projections.DataBaseObject at 0x7f2ac0447c10>]
```

Returning

You can feel a bit disturbing to do not have a hint on what's the result of your insert. If you want to get results, you can use *returning* parameter to get a result from the database:

```
>>> example_projection.insert({"num": 10,
...                                "data": "some data"},
...                               returning="id, num, data")
(6, 10, 'some data')
```

Last but not least, you can even ask drunken_boat to return the object corresponding to the projection you actually use:

```
>>> import datetime
>>> obj = example_projection.insert(
...                           {"data": "hello",
...                            "num": "6",
...                            "birthday": datetime.datetime.now()},
...                           returning="self")
>>> obj.age
datetime.timedelta(-1, 33857, 32595)
>>> obj.birthday
datetime.datetime(2015, 5, 1, 14, 35, 42, 967405)
```

Update

Updating is similar o insert but the main difference is that when you commonly insert a single row, when you update a table, you can update a lot of rows in a single query on the database.

To reflect this, the syntax of update is where clause, updated column and parameters for the where. For example, if you want to change all the example_projection object where data is "hello" to goodbye, you will write:

```
>>> example_projection.update("data=%s", {"data": "goodbye"}, ("hello",))
```

obviously you can use a Where object to make things more readable:

```
>>> example_projection.update(Where("data", "=" "%s"),
...     {"data": "goodbye"}, ("hello",))
```

Last bt not least, like with insert you can ask the database for returning:

```
>>> example_projection.update(Where("data", "=" "%s"),
...     {"data": "goodbye"}, ("hello",), returning="id, num, data")
```

or

```
>>> example_projection.update(Where("data", "=" "%s"),
...     {"data": "goodbye"}, ("hello",), returning="self")
```

Delete

With delete, you do not need to specify what will be changed. So the api of delete is like update but without changing columns:

```
>>> example_projection.delete(Where("data", "=" "%s"),("hello",))
```

Like for *update* and *insert* you can use *returning* on delete:

```
>>> example_projection.delete(Where("data", "=" "%s"),("hello",),
...     returning="self")
```

2.2.3 Relations

Foreignkey

When you need to manage relation between objects (ForeignKey), you will need a way to tell the Database wich fields of the related table you want to retreive. You will also need to tell the database how to handle the relation. Of course with projections it's really easy to do.

Of course you need to create the tables in your database. For this purpose you can use something like this:

```
db = DB(**DATABASE)
cur = db.cursor()
cur.execute(
"""CREATE TABLE author (id serial PRIMARY KEY,
                        first_name = varchar(250) NOT NULL,
                        last_name = varchar(250) NOT NULL)
""")
db.conn.commit()
cur.execute(
"""CREATE TABLE blog_post (id serial PRIMARY KEY,
                           title varchar(250),
                           introduction text,
                           body text,
                           created_at timestamp default now(),
                           last_edited_at default now(),
                           author_id integer NOT NULL,
                           published boolean default False)
""")
db.conn.commit()
cur.execute(
"alter table blog_post add foreign key(author_id)
references author"
)
db.conn.commit()
```

Then you can create two new projections:

```
class AuthorProjection(Projection):
    first_name = CharField()
    last_name = CharField()
    birthdate = Timestamp()

    class Meta:
        table = "author"

author_projection = AuthorProjection(DB(**DATABASE))

class PostProjection(Projection):
    title = CharField()
    introduction = Text()
    body = Text()
    created_at = Timestamp()
    last_edited_at = Timestamp()
    author = ForeignKey(join=["author_id", "id"],
                              projection=AuthorProjection)
    published = Boolean()

    class Meta:
        table = "blog_post"

post_projection = PostProjection(DB(**DATABASE))
```

ForeignKey take two mandatory parameters, join and projection.

- **join: This is a list of 2 elements. First element is the field on** the table you're working on. Second element is the field on the related table.
- projection:: The projection to use to render the field.

Usage of projections with foreignkeys are straitforward:

```
>>> from projections import post_projection
>>> post = post_projection.select()[0]
>>> post.__dict__
{'author': <drunken_boat.db.postgresql.projections.DataBaseObject at 0x7f7170187490>,
 'body': None,
 'created_at': datetime.datetime(2015, 5, 1, 17, 18, 20, 95226),
 'introduction': 'Pouet Pouet PimPim',
 'last_edited_at': datetime.datetime(2015, 5, 1, 17, 18, 20, 95226),
 'published': False,
 'title': 'hello'}
>>> post.author.__dict__
{'birthdate': None, 'first_name': 'Paul', 'last_name': 'Eluard'}
```

ReverseForeignkey

Another cas you will encounter a lot is when you want to reverse the relation. In our example, this can be :

How to get the authors with their corresponding posts ?

To solve this case we have to retreive all the posts belonging to one of the author and then dispatch the posts to the corresponding author representation.

ReverseForeign is a type of *Field* created for this job.

It need to know the related column on the "from" side and the related column on the "to" side. Exactly the opposite of *ForeignKey*.

In our example we want all the post with an *author_id* equal to the *author.id*.

We also need to tell *ReverseForeign* wich *Projection* to use for rendering the posts. Here is an example:

```
class PostProjectionRelated(Projection):
    title = CharField()
    introduction = Text()

    class Meta:
        table = "blog_post"

post_projection_related = PostProjectionRelated(DB(**DATABASE))


class AuthorProjectionWithPost(AuthorProjection):
    posts = ReverseForeign(join=["id", "author_id"],
                            projection=PostProjectionRelated)

author_projection_with_post = AuthorProjectionWithPost(DB(**DATABASE))
```

author_projection_with_post.select() will return a list of Author with the attribute posts containing all the posts of this author:

```
>>> for author in author_projection_with_post.select():
...     print(author.id, [post.__dict__ for post in author.posts])
1, [],
2, [{"title": "a title", "introduction": "an introduction",
"author_id": 2}, {"title": "another title", "introduction": "another
introduction", "author_id":2 ] ...
```

If the first element of ReverseForeign.join is not in the projection, (*id* in the example) it will be automaticaly added.

The same go for the ReverseForeign.projection wich will gain the second part of ReverseForeign.join (*author_id* in the example).

This is the reason why we can get *author.id* even if *id* is not on the *AuthorProjectionWithPost.fields* and *post.author_id* even if *author_id* is not on *PostProjectionRelated.fields*

Filter reverse foreignkey

Sometimes getting the related objects is not enought and you will need to filter the related objects.

To do so, *drunken_boat* offer a simple API. You only need to give to the *select* method a related argument to hold every related fields where and params:

```
>>> projection = author_projection_with_post.select(
...     related={'posts':
...         'where': 'title=%s',
...         'params': ('a title')})
>>> print post.__dict__ for post in projection[1].posts]
[{"title": "a title", "introduction": "an introduction",
"author_id": 2}]
```

Many To Many

With *ReverseForeign* and *ForeignKey* you can already implement Many to many relations. To do so, let say you have the following tables in your database:

```
CREATE TABLE product (
  id serial PRIMARY KEY
, product    text NOT NULL
, price      numeric NOT NULL DEFAULT 0
);

CREATE TABLE bill (
  id  serial PRIMARY KEY
, bill     text NOT NULL
, billdate date NOT NULL DEFAULT now()::date
);

CREATE TABLE bill_product (
  bill_id    int REFERENCES bill (bill_id) ON UPDATE CASCADE ON DELETE CASCADE
, product_id int REFERENCES product (product_id) ON UPDATE CASCADE
, amount      numeric NOT NULL DEFAULT 1
, CONSTRAINT bill_product_pkey PRIMARY KEY (bill_id, product_id)
);
```

You can then create the following projections:

```
class Product(Projection):
    product = CharField()
    price = Integer()

    class Meta:
        table = "product"

product_projection = Product(DB(**DATABASE))


class BillProduct(Projection):
    product = ForeignKey(join=["product_id", "id"],
                         projection=Product)
    class Meta:
        table = "bill_product"

productbill_projection = BillProduct(DB(**DATABASE))


class Bill(Projection):
    bill = CharField()
    billdate = Timestamp()
    products = ReverseForeign(join=["id", "bill_id"],
                              projection=BillProduct)
    class Meta:
        table = "bill"

bill_projection = Bill(DB(**DATABASE))
```

With a select on *bill_projection* you will retreive all the *BillProduct* matching your select. *BillProduct* will then retreive the corresponding *Product*. This will give you the following results:

```
[
  {"bill": "<text>",
   "billdate": <a date>,
   "products": [
     {"product":
       {"product": <text>, "price": <a price>},
     {"product":
       {"product": <text>, "price": <a price>},
     ...
   ]
 ...
]
```

create a bill with product you need to create the corresponding bill_product record. This can be done using *Returning*:

```
import datetime
bill = bill_projection.insert(
    {"bill": "a bill",
     "billdate": datetime.datetime.now()
    }, returning="id")
```

```
# because we use returning, bill is the bill.id of the object just
# saved

product = product_projection.insert(
    {"product": "screwdriver",
     "price": 10},
     returning="id"
    )

productbill_projection.insert({"bill_id": bill, "product_id": product})
```

2.3 ORM Philosophy

It's common in the ORM world to write your tables schema in your python code. This cause majors issues.

First of the is duplication. Your schema is in your database AND in your python code. Every time one chage, the other has to change.

Second is static schema. Database are not bound to a schema, they are bound to projections. Here is an example, let say you have this database schema:

Table bookstore_store:

```
id          | integer                | not NULL default, nextval('bookstore_store_id_seq'::regclass)
name        | character varying(250) | not NULL
close_time  | integer                | not NULL
open_time   | integer                | not NULL
open_date   | date                   | not NULL
location_id | integer                | not NULL
```

Table bookstore_location:

```
id       | integer                | not NULL default, nextval('bookstore_location_id_seq'::regclass)
name     | character varying(250) | non NULL
```

a store object will always have the representation

```
Store:
    id
    name
    close_time
    open_time
    open_date
    location_id
```

Let say you only need the *name* and the *location name* you will write something like:

```
for store in stores:
    store.name
    store.location.name  # Your ORM without telling you anything
                         # will make a query on location for each
                         # store
```

In Django for example, you will need to specify a select_related argument to your query to retreive location.name when querying the store table. You can't get only the store.name and the location.name without loosing the objects paradigm (or using the "only" parameter wich will not raise anything but make a query for each field you forget)

Because database can manage this in an admirable manner, and much more, we decide to create a schemaless ORM without breaking the Object Oriented paradigm. Seems interesting? Let's take the ride!

2.4 Models: database management the easy way

Basicaly Models are simply a thin layer on top of *Projections* see `database` for a full documentation on *Projections*. With models, on can easily create, update and delete database objects using a clear syntax.

Of course you also get all the power of *Projections* when you need them.

2.4.1 Initialize

To create a model, you need at least a *Database* and a *Projection* A basic projection can be something like:

```python
from drunken_boat.db.postgresql.projections import Projection

class ExampleProjection(Projection):
    """
    A basic projection
    """
    age = Timestamp(db_name="age(birthday)", virtual=True)
    birthday = Timestamp()

    class Meta:
        table = "test"
```

The underlying database table can be something like:

```
Table : test

  id serial PRIMARY KEY,
  num integer NOT NULL,
  data varchar NOT NULL,
  birthday timestamp
```

you can now import your database configuration:

```python
from example_blog.config import DATABASE
from drunken_boat.db.postgresql import DB

db = DB(**DATABASE)
```

You are now ready to use the *Models* API:

```python
from drunken_boat.db.postgresql.models import Model
my_model = Model(db, projections={"example": ExampleProjection})
```

2.4.2 Manipulation of Model objects

With a *Model* instance, you can request an object for creating data in your database in a very pythonic way. To requert an object, just do:

```python
obj = model.object()
```

Then you can set data on this object:

```
obj.num = 10
obj.data = "something"
```

And, of course save this object in the database:

```
obj.save()
```

Once saved, the object can be updated:

```
obj.num = 25
obj.update()
```

Or deleted:

```
obj.delete()
```

2.4.3 How things works

Models are nothing more than just a very thin layer around *Projections* and database objets. When you create a Model with some projections, we will look for a "default" projection. If this projection does not exist, the first projection will be set as the "default" one.

In order to implement update and delete on model object, we look for the primary key on the "default" projection. If the primary key is not on the projection, it will be automaticaly added when you call the save() method of object.

After that, because we get the primary key of your object we are able to update and delete the object using this pimary key.

All the projections you a model contains are available in Model.projections. For example, to get the default projection you just have to write:

```
my_model.projections.default
```

If you do not want to use the default projection when requestiong a Model object, you can ask for a particular projection:

```
obj = my_model.object(projection=my_model.projections.other)
```

Using a projection you do not define in your model is absolutly ok because there is no magic in how Model work:

```
obj = my_model.object(projection=AnOtherProjection)
```

The only limit is that your projection **must** be on the same table. (same primary key field)

2.4.4 Returning

Because underlying projections offers returning on insert, update and delete, Model objects offer this behavior too. Simply add the returning argument to your save, update or delete method just like with projections:

```
obj.save(returning="title")
```

2.5 Database Management

Drunken Boat is focused on performances. Most of current applications lack performances due to ORM. Yes ORM are great when you need Object Oriented programation but they lack a lot of features you can find in modern database like PostgreSQL.

Drunken Boat want to help you write powerful applications where you can use the most of your database and still use Object Oriented programmation.

This is the reason why Drunken Boat does not force you to create your database nor managing table schema in his ORM. Sure it gives you some helpful methods and functions to create database, schema, make ALTER TABLE on your databases but it's absolutely up to you to manage them the way you like.

2.5.1 Configuration & Table creation

In the project created by drunken_run.py the file config.py contains the base detail of a database connection. Change the DATABASE with connection informations of your database.

Even if drunken_boat don't force you to create table from python, for this tutorial you can use this simple script to generate the table you will use in the next step:

```
#projection.py
from drunken_boat.db.postgresql import DB
from example_blog.config import DATABASE

def create_tables():
    db = DB(**DATABASE)
    cur = db.cursor()
    cur.execute(
        """select exists(
            select * from information_schema.tables where table_name=%s)
        """,
        ('test',))
    if not cur.fetchone()[0]:
        cur.execute("""CREATE TABLE test (
        id serial PRIMARY KEY,
        num integer,
        data varchar,
        birthday timestamp)""");
        db.conn.commit()
        print("table created")
        return
    print("table already exists")
```

2.5.2 Projections

Projections are the object based representation of the *result* of a database query. See them as what you expect from the database.

Let say you make this query:

```
select name, age(birthdate) from user;
```

the corresponding projection will just fit:

```
class UserNameAge(Projection):

    name = CharField()
    age = Timestamp(name="age(birthdate)")

    class Meta:
        table = "user"
```

```
projection = UserNameAge(DB(**connection_params))
```

And you can get your results as easily as:

```
>> users = projection.select()
>> users[0].age
datetime.timedelta(13850, 50160)
```

results are list of *DataBaseObject*. because DataBaseObject are objects, you can attach any method you want on it. For example:

```
from drunken_boat.db.postgresql import DB
from config import DATABASE
from drunken_boat.db.postgresql.fields import Timestamp
from drunken_boat.db.postgresql.projections import (Projection,
                                                    DataBaseObject)
class ExampleDataBaseObject(DataBaseObject):

    def display_birthyear_and_days(self):
        days = self.age.days
        year = self.birthdate.year
        return "{} days since {}".format(days, year)

class ExampleProjection(Projection):
    """
    Here you can write your real projections
    """

    age = Timestamp(db_name="age(birthday)", virtual=True)
    birthdate = Timestamp()

    class Meta:
        table = "test"
        database_object = ExampleDataBaseObject

example_projection = ExampleProjection(DB(**DATABASE))


>>> from projections import example_projection
>>> t = example_projection.select()
>>> t[0].display_birthyear_and_days()
'13850 days since 1977'
```

2.5.3 Where

One thing you will surely do very often is to use *Projection* with WHERE clause. Where clause are defined with 2 sides. First side is the clause and the comparison operator, the other side is the parameter.

For example, in the statement:

WHERE id > 4;

id is the clause, > is the comparison operator, and 4 is the parameter.

The first an easier way to make a query with a WHERE clause is simply adding where and parameter to the select statement:

```
>>> projection.select(where='id=%s', params=(1,))
```

If it's perfectly ok to do so, but sometimes you will need to store a WHERE clause to use it in many places in your code. For this the Where object is here to help you.

A where object take a clause, an operator and a value:

```
from drunken_boat.db.postgresql.query import Where
where = Where("id", "=", "%s")
```

As you can see a Where object is very similar to the select version. The difference is that you do not define a parameter yet. The parameter will be define when calling the select method of your *Projection*:

```
>>> projection.select(where=where, params=(1,))
```

Multiple Where

It's also possible to use multiple where in a single select using biwise operations. AND, OR and NOT are supported:

AND:

```
>>> where = Where("id", "=", "%s") & Where("title", "=", "%s")
```

OR:

```
>>> where = Where("id", "=", "%s") | Where("title", "=", "%s")
```

NOT:

```
>>> where = Where("id", "=", "%s") & ~Where("title", "=", "%s")
```

NOT can be used as is to make exclude clause:

```
>>> where = ~Where("title", "=", "%s")
```

You can also define priorities with parenthesis:

```
>>> where = Where("id", "=", "%s") | (Where("title", "=", "%s") & Where("intro", "=", "%s"))
```

this will be rendered as:

```
id = %s OR (title = %s AND intro = %s)
```

2.5.4 Insert

Even if you do not describe the table schema of your tables, drunken_boat introspect your table schema to give you automatic validation of data before even hitting the database.

To demonstrate this behavior let's create another table:

```
Table : test

  id serial PRIMARY KEY,
  num integer NOT NULL,
  data varchar NOT NULL,
  birthday timestamp
```

And another projection:

```
class ExampleProjection(Projection):
    """
    Here you can write your real projections
    """
    age = Timestamp(db_name="age(birthday)", virtual=True)
    birthday = Timestamp()

    class Meta:
        table = "test"
        database_object = ExampleDataBaseObject

example_projection = ExampleProjection(DB(**DATABASE))
```

Now, with a shell try to insert some data in the table:

```
>>> from projections import example_projection
>>> example_projection.insert({"birthday": datetime.datetime.now()})
ValueError: num of type integer is required
data of type character varying is required
```

Now that you know wich data you must use to insert data you can type:

```
>>> example_projection.insert({"num": 10,
...                            "data": "some data"})
```

You can check that your record is saved in the database:

```
>>> example_projection.select()
... [<projections.DataBaseObject at 0x7f2ac0447c10>]
```

2.5.5 Returning

You can feel a bit disturbing to do not have a hint on what's the result of your insert. If you want to get results, you can use *returning* parameter to get a result from the database:

```
>>> example_projection.insert({"num": 10,
...                            "data": "some data"},
...                           returning="id, num, data")
(6, 10, 'some data')
```

Last but not least, you can even ask drunken_boat to return the object corresponding to the projection you actually use:

```
>>> import datetime
>>> obj = example_projection.insert(
...                       {"data": "hello",
...                        "num": "6",
...                        "birthday": datetime.datetime.now()},
...                       returning="self")
>>> obj.age
datetime.timedelta(-1, 33857, 32595)
>>> obj.birthday
datetime.datetime(2015, 5, 1, 14, 35, 42, 967405)
```

2.5.6 Update

Updating is similar o insert but the main difference is that when you commonly insert a single row, when you update a table, you can update a lot of rows in a single query on the database.

To reflect this, the syntax of update is where clause, updated column and parameters for the where. For example, if you want to change all the example_projection object where data is "hello" to goodbye, you will write:

```
>>> example_projection.update("data=%s", {"data": "goodbye"}, ("hello",))
```

obviously you can use a Where object to make things more readable:

```
>>> example_projection.update(Where("data", "=" "%s"),
...     {"data": "goodbye"}, ("hello",))
```

Last bt not least, like with insert you can ask the database for returning:

```
>>> example_projection.update(Where("data", "=" "%s"),
...     {"data": "goodbye"}, ("hello",), returning="id, num, data")
```

or

```
>>> example_projection.update(Where("data", "=" "%s"),
...     {"data": "goodbye"}, ("hello",), returning="self")
```

2.5.7 Delete

With delete, you do not need to specify what will be changed. So the api of delete is like update but without changing columns:

```
>>> example_projection.delete(Where("data", "=" "%s"),("hello",))
```

Like for *update* and *insert* you can use *returning* on delete:

```
>>> example_projection.delete(Where("data", "=" "%s"),("hello",),
...     returning="self")
```

2.6 Relations

2.6.1 Foreignkey

When you need to manage relation between objects (ForeignKey), you will need a way to tell the Database wich fields of the related table you want to retreive. You will also need to tell the database how to handle the relation. Of course with projections it's really easy to do.

Of course you need to create the tables in your database. For this purpose you can use something like this:

```
db = DB(**DATABASE)
cur = db.cursor()
cur.execute(
"""CREATE TABLE author (id serial PRIMARY KEY,
                        first_name = varchar(250) NOT NULL,
                        last_name = varchar(250) NOT NULL)
""")
db.conn.commit()
cur.execute(
"""CREATE TABLE blog_post (id serial PRIMARY KEY,
                           title varchar(250),
                           introduction text,
                           body text,
                           created_at timestamp default now(),
                            last_edited_at default now(),
                            author_id integer NOT NULL,
                            published boolean default False)
""")
db.conn.commit()
cur.execute(
"alter table blog_post add foreign key(author_id)
references author"
)
db.conn.commit()
```

Then you can create two new projections:

```
class AuthorProjection(Projection):
    first_name = CharField()
    last_name = CharField()
    birthdate = Timestamp()

    class Meta:
        table = "author"

author_projection = AuthorProjection(DB(**DATABASE))

class PostProjection(Projection):
    title = CharField()
    introduction = Text()
    body = Text()
    created_at = Timestamp()
    last_edited_at = Timestamp()
    author = ForeignKey(join=["author_id", "id"],
                        projection=AuthorProjection)
    published = Boolean()

    class Meta:
        table = "blog_post"

post_projection = PostProjection(DB(**DATABASE))
```

ForeignKey take two mandatory parameters, join and projection.

- **join: This is a list of 2 elements. First element is the field on** the table you're working on. Second element is the field on the related table.
- projection:: The projection to use to render the field.

Usage of projections with foreignkeys are straitforward:

```
>>> from projections import post_projection
>>> post = post_projection.select()[0]
>>> post.__dict__
{'author': <drunken_boat.db.postgresql.projections.DataBaseObject at 0x7f7170187490>,
 'body': None,
 'created_at': datetime.datetime(2015, 5, 1, 17, 18, 20, 95226),
 'introduction': 'Pouet Pouet PimPim',
 'last_edited_at': datetime.datetime(2015, 5, 1, 17, 18, 20, 95226),
 'published': False,
 'title': 'hello'}
>>> post.author.__dict__
{'birthdate': None, 'first_name': 'Paul', 'last_name': 'Eluard'}
```

2.6.2 ReverseForeignkey

Another cas you will encounter a lot is when you want to reverse the relation. In our example, this can be :

How to get the authors with their corresponding posts ?

To solve this case we have to retreive all the posts belonging to one of the author and then dispatch the posts to the corresponding author representation.

ReverseForeign is a type of *Field* created for this job.

It need to know the related column on the "from" side and the related column on the "to" side. Exactly the opposite of *ForeignKey*.

In our example we want all the post with an *author_id* equal to the *author.id*.

We also need to tell *ReverseForeign* wich *Projection* to use for rendering the posts. Here is an example:

```
class PostProjectionRelated(Projection):
    title = CharField()
    introduction = Text()

    class Meta:
        table = "blog_post"

post_projection_related = PostProjectionRelated(DB(**DATABASE))


class AuthorProjectionWithPost(AuthorProjection):
    posts = ReverseForeign(join=["id", "author_id"],
                           projection=PostProjectionRelated)

author_projection_with_post = AuthorProjectionWithPost(DB(**DATABASE))
```

author_projection_with_post.select() will return a list of Author with the attribute posts containing all the posts of this author:

```
>>> for author in author_projection_with_post.select():
...     print(author.id, [post.__dict__ for post in author.posts])
1, [],
2, [{"title": "a title", "introduction": "an introduction",
"author_id": 2}, {"title": "another title", "introduction": "another
introduction", "author_id":2 ] ...
```

If the first element of ReverseForeign.join is not in the projection, (*id* in the example) it will be automaticaly added.

The same go for the ReverseForeign.projection wich will gain the second part of ReverseForeign.join (*author_id* in the example).

This is the reason why we can get *author.id* even if *id* is not on the *AuthorProjectionWithPost.fields* and *post.author_id* even if *author_id* is not on *PostProjectionRelated.fields*

2.6.3 Filter reverse foreignkey

Sometimes getting the related objects is not enought and you will need to filter the related objects.

To do so, *drunken_boat* offer a simple API. You only need to give to the *select* method a related argument to hold every related fields where and params:

```
>>> projection = author_projection_with_post.select(
...     related={'posts':
...         'where': 'title=%s',
...         'params': ('a title')})
>>> print post.__dict__ for post in projection[1].posts]
[{"title": "a title", "introduction": "an introduction",
"author_id": 2}]
```

2.6.4 Many To Many

With *ReverseForeign* and *ForeignKey* you can already implement Many to many relations. To do so, let say you have the following tables in your database:

```
CREATE TABLE product (
  id serial PRIMARY KEY
, product    text NOT NULL
, price      numeric NOT NULL DEFAULT 0
);

CREATE TABLE bill (
  id  serial PRIMARY KEY
, bill     text NOT NULL
, billdate date NOT NULL DEFAULT now()::date
);

CREATE TABLE bill_product (
  bill_id    int REFERENCES bill (bill_id) ON UPDATE CASCADE ON DELETE CASCADE
, product_id int REFERENCES product (product_id) ON UPDATE CASCADE
, amount     numeric NOT NULL DEFAULT 1
, CONSTRAINT bill_product_pkey PRIMARY KEY (bill_id, product_id)
);
```

You can then create the following projections:

```
class Product(Projection):
    product = CharField()
    price = Integer()

    class Meta:
        table = "product"

product_projection = Product(DB(**DATABASE))


class BillProduct(Projection):
    product = ForeignKey(join=["product_id", "id"],
                         projection=Product)
    class Meta:
        table = "bill_product"

productbill_projection = BillProduct(DB(**DATABASE))


class Bill(Projection):
    bill = CharField()
    billdate = Timestamp()
    products = ReverseForeign(join=["id", "bill_id"],
                                  projection=BillProduct)
    class Meta:
        table = "bill"

bill_projection = Bill(DB(**DATABASE))
```

With a select on *bill_projection* you will retreive all the *BillProduct* matching your select. *BillProduct* will then retreive the corresponding *Product*. This will give you the following results:

```
[
  {"bill": "<text>",
   "billdate": <a date>,
   "products": [
     {"product":
       {"product": <text>, "price": <a price>},
     {"product":
       {"product": <text>, "price": <a price>},
     ...
   ]
 ...
]
```

create a bill with product you need to create the corresponding bill_product record. This can be done using *Returning*:

```
import datetime
bill = bill_projection.insert(
    {"bill": "a bill",
     "billdate": datetime.datetime.now()
    }, returning="id")

# because we use returning, bill is the bill.id of the object just
# saved

product = product_projection.insert(
    {"product": "screwdriver",
     "price": 10},
     returning="id"
    )

productbill_projection.insert({"bill_id": bill, "product_id": product})
```

2.7 Tutorial

2.7.1 The inevitable blog example

First, install *drunken-boat* see Install. Once drunken_boat installed you can boostrap your first application with:

```
drunken_run.py bootstrap example_blog
```

This will create for you all you need to start:

```
cd /home/yohann/Dev/drunken_boat/example_blog
python application.py
```

then visit http://localhost:5000/

If everything is fine you should see a bare “Hello World!” This it a first victory but there is much more to do.

2.8 Contributing

2.8.1 Contributing

This document provides guidelines for people who want to contribute to the *drunken-boat* project.

Create tickets

Please use drunken-boat bugtracker [1] **before** starting some work:

- check if the bug or feature request has already been filed. It may have been answered too!
- else create a new ticket.
- if you plan to contribute, tell us, so that we are given an opportunity to give feedback as soon as possible.
- Then, in your commit messages, reference the ticket with some `refs #TICKET-ID` syntax.

Use topic branches

- Work in branches.
- Please never push in `master` directly.
- Prefix your branch with one the following keyword `feature/` when adding a new feature and `fix/` when working on a fix. You can also add the ticket ID corresponding to the issue to be explicit.
- If you work in a development branch and want to refresh it with changes from master, please rebase [2] or merge-based rebase [2], i.e. do not merge master.

Fork, clone

Clone *drunken-boat* repository (adapt to use your own fork):

```
git clone https://github.com/boblefrag/drunken_boat
cd drunken_boat
```

Usual actions

The *setup.py* is the reference card for usual actions in development environment:

- Install development toolkit with *python setup.py develop*.
- Run tests with *python setup.py test*.
- Build documentation: *python setup.py build_sphinx*
- Release *drunken_boat* project with zest.releaser [3]: `fullrelease`.

Notes & references

[1] https://github.com/boblefrag/drunken-boat/issues
[2] http://git-scm.com/book/en/v2/Git-Branching-Rebasing
[3] https://pypi.python.org/pypi/zest.releaser/

Indices and tables

- genindex
- modindex
- search