ALeRCE Alert Processing Framework

Release 0.0.1

ALeRCE

Dec 14, 2022

CONTENTS

1 Installing *db_plugins* **3**

2 Documentation **5**

2.1 Database Connection 5

2.1.1 1. Plugins 5

2.1.2 2. Database Initialization 5

2.1.3 3. Migrations 6

2.2 Database plugins 6

2.2.1 SQL 6

Index **13**

db_plugins is an ORM style library created to interact with different databases. The main feature of these plugins is to provide an interface for database querying, reducing the amount of code and helping to decouple components.

CHAPTER

ONE

INSTALLING *DB_PLUGINS*

db_plugins installation can be done with *pip*. You can clone the repository and then

```
pip install .
```

or you can install it from Python Package Index

```
pip install db-plugins
```

CHAPTER

TWO

DOCUMENTATION

2.1 Database Connection

db_plugins is an ORM style library created to interact with different databases. The main feature of these plugins is to provide an interface for database querying, reducing the amount of code and helping to decouple components.

2.1.1 1. Plugins

ALeRCE integrates with databases through plugins. Each plugin is supposed to provide functionality for a specific database engine.

The design concept is that there are multiple database connections but all of them share the same interface so that connecting to any provided engine is done in a similar way.

This provides a way to connect and query different database engines using the same methods and classes, for example a database connection *db* has a *query* method that returns a *BaseQuery* object that has methods for inserting, updating or getting paginated results from a SQL database, but it also works the same way for a Mongodb database.

This also provides the option to change database engines without having to change the application structure too much.

2.1.2 2. Database Initialization

Database plugins will read the configuration you define in a `settings.py` file. This file should have a *DB_CONFIG* dictionary with the database connection parameters.

Here is and example on the params used with the SQL plugin:

```
DB_CONFIG: {
    "SQL": {
      "ENGINE": "postgresql",
      "HOST": "host",
      "USER": "username",
      "PASSWORD": "pwd",
      "PORT": 5432, # postgresql tipically runs on port 5432. Notice that we use an int␣
↪here.
      "DB_NAME": "database",
    }
}
```

After defining *DB_CONFIG* you can now initiaize your database. To do so, run the `initdb` command as follows

```
dbp initdb
```

2.1.3 3. Migrations

When changes to models are made you would want to update the database without creating it all again, or maybe you want to undo some changes and return to a previous state.

The solution is to create migrations. Migrations keep track of your database changes and let you detect differences between your database and models and update the database accordingly.

Migrations will be created by running `dbp make_migrations`. This command will read your database credentials from *DB_CONFIG* inside `settings.py`.

Then, to update your database to latest changes execute `dbp migrate`.

2.2 Database plugins

In this page you will find documentation for all plugins that are available.

2.2.1 SQL

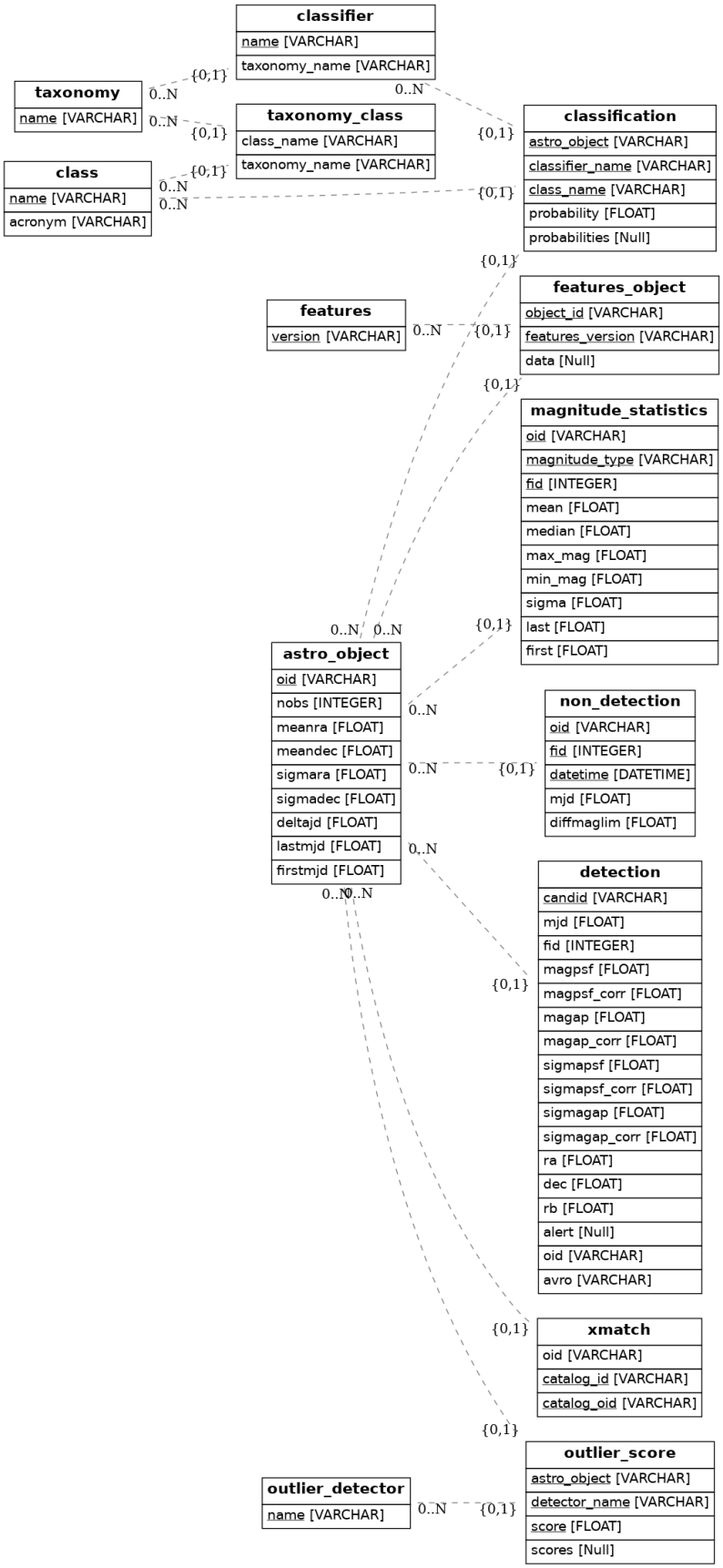
Initialize database

Before you connect to your database, make sure you initialize it first. To do that execute the following command from a directory containing a `settings.py` file.

You can also privide the settings directory with `--settings_path`.

```
dbp initdb
```

When you run this command with an empty database it will create the following schema:


classifier
name [VARCHAR]
taxonomy_name [VARCHAR]
taxonomy
name [VARCHAR]
taxonomy_class
class_name [VARCHAR]
taxonomy_name [VARCHAR]
class
name [VARCHAR]
acronym [VARCHAR]
classification
astro_object [VARCHAR]
classifier_name [VARCHAR]
class_name [VARCHAR]
probability [FLOAT]
probabilities [Null]
features
version [VARCHAR]
features_object
object_id [VARCHAR]
features_version [VARCHAR]
data [Null]
magnitude_statistics
oid [VARCHAR]
magnitude_type [VARCHAR]
fid [INTEGER]
mean [FLOAT]
median [FLOAT]
max_mag [FLOAT]
min_mag [FLOAT]
sigma [FLOAT]
last [FLOAT]
first [FLOAT]
astro_object
oid [VARCHAR]
nobs [INTEGER]
meanra [FLOAT]
meandec [FLOAT]
sigmara [FLOAT]
sigmadec [FLOAT]
deltajd [FLOAT]
lastmjd [FLOAT]
firstmjd [FLOAT]
non_detection
oid [VARCHAR]
fid [INTEGER]
datetime [DATETIME]
mjd [FLOAT]
diffmaglim [FLOAT]
detection
candid [VARCHAR]
mjd [FLOAT]
fid [INTEGER]
magpsf [FLOAT]
magpsf_corr [FLOAT]
magap [FLOAT]
magap_corr [FLOAT]
sigmapsf [FLOAT]
sigmapsf_corr [FLOAT]
sigmagap [FLOAT]
sigmagap_corr [FLOAT]
ra [FLOAT]
dec [FLOAT]
rb [FLOAT]
alert [Null]
oid [VARCHAR]
avro [VARCHAR]
xmatch
oid [VARCHAR]
catalog_id [VARCHAR]
catalog_oid [VARCHAR]
outlier_detector
name [VARCHAR]
outlier_score
astro_object [VARCHAR]
detector_name [VARCHAR]
score [FLOAT]
scores [Null]
0..N
{0,1}

Migrations

Migrations keep track of database changes. To fully init the database with your step configuration run

```
dbp make_migrations
dbp migrate
```

This will set the head state for tracking changes on the database and also execute any migrations that might be present.

The first command `dbp make_migrations` will create migration files according to differences from dbp models and your database.

The seccond command `dbp migrate` will execute the migrations and update your database.

What migrations can and can't detect

Migrations will detect:

- Table additions, removals.
- Column additions, removals.
- Change of nullable status on columns.
- Basic changes in indexes

Migrations can't detect:

- Changes of table name. These will come out as an add/drop of two different tables, and should be hand-edited into a name change instead.
- Changes of column name. Like table name changes, these are detected as a column add/drop pair, which is not at all the same as a name change.

Set database Connection

```
from db_plugins.db.sql import SQLConnection
from db_plugins.db.sql.models import *
```

```
db_config = {
    "SQL": {
      "ENGINE": "postgresql",
      "HOST": "host",
      "USER": "username",
      "PASSWORD": "pwd",
      "PORT": 5432, # postgresql tipically runs on port 5432. Notice that we use an int␣
↪here.
      "DB_NAME": "database",
    }
}
```

```
db = SQLConnection()
db.connect(config=db_config["SQL"])
```

The above code will create a connection to the database wich we will later use to store objects.

Create model instances

Use get_or_create function to get an instance of a model. The instance will be an object from the database if it already exists or it will create a new instance.

```
oid = "ZTF_OID"
model_args = {}
model_args["ndethist"] = 0
model_args["ncovhist"] = 0
model_args["mjdstarthist"] = 0.0
model_args["mjdendhist"] = 0.0
model_args["firstmjd"] = 0.0
model_args["lastmjd"] = 0.0
model_args["ndet"] = 0
model_args["deltajd"] = 0.0
model_args["meanra"] = 0.0
model_args["meandec"] = 0.0
model_args["step_id_corr"] = "v1.0.0"
model_args["corrected"] = False
model_args["stellar"] = False
```

```
obj, created = db.query(Object).get_or_create(filter_by={"oid": oid}, **model_args)
print(obj, "created: " + str(created))
```

```
<AstroObject(oid='ZTFid')> created:  False
```

In the above example we use the object id as a filter since it is the primary key of the Object model. Notice that `get_or_create` can receive the model as a parameter or it can inherit it from the `query` parameter.

The **important** part is that `model_args` should contain all attributes of the table.

Add multiple objects to the database

If you need to insert multiple objects at once, there is a faster way than using `get_or_create` multiple times. You can use the `bulk_insert` method.

It will take a list of dictionaries where each dictionary has all the attributes for a table as keys.

```
db.query(Detection).bulk_insert(prv_detections)
```

Where prv_detections is a list of dict where each dict contains information to populate Detection table.

Update instances

There is a particularity when you make updates to instances. Let's say that we have an object instance and we want to change its lastmjd.

```
obj = db.query(Object).get_or_create(filter_by={"oid": "ZTF123"})

obj = db.query().update(obj, {"lastmjd": 12345})
```

After updating the instance you have to commit the changes. This is done in the following way:

```
db.session.commit()
```

DatabaseConnection documentation

class db_plugins.db.sql.**SQLConnection**(*config=None*, *engine=None*, *Base=None*, *Session=None*, *session=None*)

Methods

connect(config[, base, session_options, ...])	Establishes connection to a database and initializes a session.
create_db()	
create_session(use_scoped[, scope_func])	Creates a SQLAlchemy Session object to interact with the database.
drop_db()	
query(*args)	Creates a BaseQuery object that allows you to query the database using the SQLAlchemy API, or using the BaseQuery methods like `get_or_create`

end_session	

connect(*config*, *base=None*, *session_options=None*, *create_session=True*, *use_scoped=False*, *scope_func=None*)

Establishes connection to a database and initializes a session.

Parameters

config
[dict] Database configuration. For example:

```
"SQL": {
    "ENGINE": "postgresql",
    "HOST": "host",
    "USER": "username",
    "PASSWORD": "pwd",
    "PORT": 5432, # postgresql tipically runs on port 5432. Notice␣
↪that we use an int here.
    "DB_NAME": "database",
}
```

base
[sqlalchemy.ext.declarative.declarative_base()] Base class used by sqlalchemy to create tables

session_options
[dict] Options passed to sessionmaker

create_session
[Boolean] Whether to instantiate a session or not. The default value is True since this is

the previous behavior. Proper usage should be to pass False and create / close the session on demand inside the application. You should call this method first and then call SQLConnection.create_session(use_scoped)

```
# scoped session example
conn = SQLConnection()
conn.connect(config, create_session=False)
conn.create_session(use_scoped=True)
# use session
conn.query()
conn.session.remove()
```

use_scoped
[Boolean] Whether to use scoped session or not. Use a scoped session if you are using it in a web service like an API. Read more about scoped sessions at: https://docs.sqlalchemy.org/en/13/orm/contextual.html?highlight=scoped

scope_func
[function] A function which serves as the scope for the session. The session will live only in the scope of that function.

`create_session`(*use_scoped*, *scope_func=None*)

Creates a SQLAlchemy Session object to interact with the database.

Parameters

use_scoped
[Boolean] Whether to use scoped session or not. Use a scoped session if you are using it in a web service like an API. Read more about scoped sessions at: https://docs.sqlalchemy.org/en/13/orm/contextual.html?highlight=scoped

scope_func
[function] A function which serves as the scope for the session. The session will live only in the scope of that function.

`query`(**args*)

Creates a BaseQuery object that allows you to query the database using the SQLAlchemy API, or using the BaseQuery methods like `get_or_create`

Parameters

args
[tuple] Args you can pass to SQLALchemy Query class, for example a model.

Examples

```
# Using SQLAlchemy API
db_conn.query(Probability).all()
# Using db-plugins
db_conn.query(Probability).find(filter_by=**filters)
db_conn.query().get_or_create(model=Object, filter_by=**filters)
```

INDEX

C

`connect()` (*db_plugins.db.sql.SQLConnection method*), 10
`create_session()` (*db_plugins.db.sql.SQLConnection method*), 11

Q

`query()` (*db_plugins.db.sql.SQLConnection method*), 11

S

`SQLConnection` (*class in db_plugins.db.sql*), 10