django-data-migration Documentation

Release 0.2.1

Philipp Böhm

Sep 27, 2017

Contents

`django-data-migration` is a reusable Django app that migrates your legacy data into your new django app. The only thing you have to supply is an appropriate SQL query that transforms your data fromthe old schema into your model structure. Dependencies between these migrations will be resolved automatically. Give it a try!

Warning: This documentation is a work in progress. Please open an issue on Github if any information are missing.

CHAPTER 1

Contents:

Installing

Installation of `django-data-migration` is straight forward, as it only requires the following steps (assuming you have already set up `virtualenv` and `pip`).

1. Install using pip:

   ```
   pip install django-data-migration
   ```

2. Add `django-data-migration` to your `requirements.txt`
3. Add to `INSTALLED_APPS`:

   ```
   'data_migration',
   ```

4. Run `./manage.py migrate` or `./manage.py syncdb` to create the included models

Writing Migrations

What is a Migration?

A migration is a Python class that should be placed in a file called `data_migration_spec.py` in one of your app-directories. `django-data-migrations` searches in each app, included in `INSTALLED_APPS`, for this file and imports all from it automatically.

Your migration normally specifies the following things:

- A database connection to your legacy data (whereever this is)
- The model class, the migration should create instances for
- A corresponding SQL-Query, which maps the old DB-schema to the new Django-model-schema

   - You can specify what should be done with special columns, returned by the query (Many2Many-, ForeignKey-, One2One-Relations). With minimal configuration, these things can be migrated automatically.

- Dependencies to other models can be specified. This is used, to determine the order each migration can be applied. e.g. If a migration specifies a model as dependency, his migration will be executed before our migration will be processed.
- You can implement different hooks, where you normally manipulate the data returned by the query or do some things which are not possible by SQL itself.
- You can specify, if your migration should look for new instances on a second run. This is not the default case.

A complete Migration example

To give you an overview, how a common migration looks, the following listing shows a migration for a `Post` model. This is an excerpt from a `data_migration_spec.py` which can be found in a testing app, which is used by `django-data-migration` itself.

The complete app can be found here ...

```
class PostMigration(BaseMigration):
    query = """
    SELECT id,
        Title as title,
        Body as body,
        Posted as posted,
        Author as author,
        (
            SELECT
                GROUP_CONCAT(id)
            FROM comments c
            WHERE c.Post = p.id
        ) as comments
    FROM posts p;
    """
    model = Post
    depends_on = [ Author, Comment ]
    column_description = {
        'author': is_a(Author, search_attr="id", fk=True, prefetch=False),
        'comments': is_a(Comment, search_attr="id", m2m=True,
                         delimiter=",", prefetch=False)
    }

    @classmethod
    def hook_after_save(self, instance, row):
        # because of the auto_now_add flag, we have to set it hard to this value
        instance.posted = row['posted']
        instance.save()
```

As you can see, `PostMigration` inherits from a class called `BaseMigration`. This is one of the classes which is listed here *Setup Database Connection*.

Migration details

Setup Database Connection

`django-data-migration` should support as many databases as possible, so the connection part is not implemented directly for each database. You have to override the `open_db_connection` classmethod in your migration.

Tip: The connection handling should be implemented once in a `BaseMigration` where all other Migrations inherit from.

Important: `django-data-migration` requires that the database returns a `DictCursor`, where each row is a dict with column names as keys and the row as corresponding values.

SQLite

The following code implements an example database connection for SQLite:

```
import sqlite3

class BaseMigration(Migration):

    @classmethod
    def open_db_connection(self):
        conn = sqlite3.connect(......)

        def dict_factory(cursor, row):
            d = {}
            for idx, col in enumerate(cursor.description):
                d[col[0]] = row[idx]
            return d

        conn.row_factory = dict_factory
        return conn
```

MySQL

You have to install the corresponding MySQL-Python-driver by executing:

```
pip install MySQL-python
```

The following code implements an example database connection for MySQL.

```
import MySQLdb

class BaseMigration(Migration):

    @classmethod
    def open_db_connection(self):
        return MySQLdb.connect(......,
            cursorclass=MySQLdb.cursors.DictCursor
        )
```

PostgreSQL

You have to install the corresponding PostgreSQL-Python-driver by executing:

```
pip install psycopg2
```

Important: a version of psycopg >= 2.5 is required as it allows the `cursor_factory` to be specified through `connect()` instead of `get_cursor()`.

The following code implements an example database connection for PostgreSQL.

```
import psycopg2
import psycopg2.extras

class BaseMigration(Migration):

    @classmethod
    def open_db_connection(self):
        return psycopg2.connect(......,
            cursor_factory=psycopg2.extras.RealDictCursor
        )
```

MS-SQL

@aidanlister contributed a sample DB connection for MS-SQL using `pyodbc`, which has to be installed first:

```
pip install pyodbc
```

The following code implements an example database connection for MS-SQL.

```
import pyodbc

class ConnectionWrapper(object):
    def __init__(self, cnxn):
        self.cnxn = cnxn

    def __getattr__(self, attr):
        return getattr(self.cnxn, attr)

    def cursor(self):
        return CursorWrapper(self.cnxn.cursor())


class CursorWrapper(object):
    def __init__(self, cursor):
        self.cursor = cursor

    def __getattr__(self, attr):
        return getattr(self.cursor, attr)

    def fetchone(self):
        row = self.cursor.fetchone()
        if not row:
            return None
        return dict((t[0], value) for t, value in zip(self.cursor.description, row))

    def fetchall(self):
        rows = self.cursor.fetchall()
        if not rows:
            return None

        dictrows = []
        for row in rows:
            row = dict((t[0], value) for t, value in zip(self.cursor.description,␣
↪row))
            dictrows.append(row)
        return dictrows


class BaseMigration(Migration):
    @classmethod
    def open_db_connection(self):
        dsn = "DRIVER={SQL Server Native Client 11.0};SERVER=X;DATABASE=X;UID=X;PWD=X"

        cnxn = pyodbc.connect(dsn)
        wrapped_connection = ConnectionWrapper(cnxn)
        return wrapped_connection
```

What can be configured in every migration

In your migration classes you have several configuration options, which are listed below with a short description. For an in-depth explanation you can consult the paragraphs below.

Writing effective Migration-queries

Important: TODO

Define dependencies

Important: TODO

Describe special columns

Your `query` can include special columns, that are represented as special Django-relations (ForeignKey-, Many2Many- or One2One-Relations). Or you can exclude specific columns from automatic processing. You will normally define these settings with an invocation of the `is_a`-function, which does some tests and returns the required settings. This will then be used by `django-data-migration` in different places.

Some examples for `is_a` can be found here: *A complete Migration example*.

Using Migration Hooks

`data_migration.migration.Migration` defines a number of different hook-functions which will be called at different places allowing you to customize the migration work at different levels.

Error-Handling

In case of an exception when creating the instances, a default error handler will be called, to print the current row to stderr and than reraise the exception.

You can override this hook in your migration if it requires special handling of errors. When this method returns without an exception, the next row from the query will be processed.

Hook-Flowchart

The following graphic shows each Hook-method and when it is called in contrast to the model handling which is done by `django-data-migration`.

```
+------------------+
|hook_before_all() |
+--------------+---+
               |
    +-----+    |
    |     |    |
    |  +--v----v--------------------+
    |  |hook_before_transformation()|
    |  +-------+--------------------+
    |          |
    |      +---v--------------------+
    |      |instance = model(**data)|
    |      +---+--------------------+
    |          |
    |  +-------v----------+
    |  |hook_before_save()|
    |  +-------+----------+
    |          |
    |      +---v-----------+
    |      |instance.save()|
    |      +---+-----------+
    |          |
    |  +-------v---------+
    |  |hook_after_save()|
    |  +-------+---------+
    |          |
    +-------+--+
            |
            |
+-----------v-----+
|hook_after_all() |
+-----------------+
```

Implement updateable Migrations

Important: TODO

Migrating your data

After you wrote all of your Migrations, you can put them in action by executing the `migrate_legacy_data` management command:

```
./manage.py migrate_legacy_data [--commit]
```

If you omit the `--commit`-flag, the data is not saved to the DB. This is useful when you develop your migrations and have some failing migrations, but the db is not cluttered with any incomplete data. When your migrations are succesful you can add `--commit` and your data is saved when no errors occur.

Note: In older versions of this library, the management command is called `migrate_this_shit`. This has been deprecated, but it is still there. `migrate_legacy_data` should be more appropriate.

Troubleshooting

GROUP_CONCAT column size limit in Mysql is too low

Mysql has fairly low limit for rows that can be merged by the `GROUP_CONCAT` function. For large result sets, this has to be increased. This can be done with the following SQL statement, which can be executed in `open_db_connection`.:

```
@classmethod
def open_db_connection(self):
    conn = MySQLdb.connect(....)
    cursor = conn.cursor()

    cursor.execute('SET SESSION group_concat_max_len = 60000000;')
    return conn
```

Changelog

Version 0.2.1

- `atomic()` is now used instead of `commit_on_success()` when it is available. This prevents deprecation warnings that are displayed with Django >= 1.7.

Version 0.2.0

- Introduced some performance improvements by implementing prefetching of related objects, that reduces the number of issued SQL-Queries dramatically. There is now the opportunity to assign related objects by their id instead of a full instance, which can reduce the memory usage.

- There are two new arguments in `is_a`: `prefetch=True` and `assign_by_id=False`. Because prefetching is enabled by default, it should bring a massive performance boost only by upgrading to this version
- Switching to a new minor release because of the changed behaviour in `get_object`

Version < 0.2.0

There is no explicit Changelog until 0.2.0. Use `git log` to get the information from git.

Python Module Index

Index