django-nsync Documentation

Release 0.3.9

Andrew Dodd

April 22, 2016

CHAPTER 1

django-nsync

Django NSync provides a simple way to keep your Django Model data ‘n-sync with N external systems.

1.1 Features

Includes:

- Synchronise models with data from external systems
  - Create, update or delete model objects
  - Modify relational fields
  - Allow multiple systems to modify the same model object
- CSV file support out of the box
  - Ships with commands to process a single CSV file or multiple CSV files
- No need for more code
  - Nsync does not require you to inherit from special classes, add ‘model mapping’ objects or really define anything in Python

Not-included:

- Export (to CSV or anything else for that matter)
  - There are other packages that can do this
  - Why do you want the data out? Isn’t that what your application is for? ;-)
- Admin integration
  - There isn’t much to this package, if you want to add the models to your admin pages it is probably better if you do it (that’s what I’ve done in my use case)

Not-yet included:

- Other file formats out of the box
  - Love it or hate it, CSV is ubiquitous and simple (its limitations also force simplicity)
  - The CSV handling part is separated from the true NSync part, so feel free to write your own lyrics-from-wav-file importer.
- Intricate data format handling
  - E.g. parsing date times etc

    – This can be side-stepped by creating `@property` annotated handlers though (see the examples from more info)

1.2 Documentation

The full documentation is at https://django-nsync.readthedocs.org.

1.3 Credits

Tools used in rendering this package:

- Cookiecutter Used to create the initial repot
- cookiecutter-pypackage Used by Cookiecutter to create the initial repo

For helping me make sense of the python pacakging world (and the bad practices codified in some of the tools/blogs out there):

- Hynek Schlawack Whose blog posts on packaging Python apps etc were indispensible
- Ionel Cristian Maries (sorry, too lazy for unicode) Whose blog post on python packaging was also indispensible

1.4 User Guide

1.4.1 Quickstart

Installation

To get started using `django-nsync`, install it with `pip`:

```
$ pip install django-nsync
```

Add `"nsync"` to your project's `INSTALLED_APPS` setting. E.g.:

```
INSTALLED_APPS += (
    'nsync',
)
```

Run `python manage.py migrate` to create the Django-Nsync models.

You will now have in your application:

- An `ExternalSystem` model, used to represent 'where' information is synchronising from
- An `ExternalKeyMapping` model, used to record the mappings from an `ExternalSystem`'s key for a model object, to the actual model object internally
- Two 'built-in' commands for synchronising data with:
    – *syncfile* - Which synchronises a single file, but allows the user to specify the `ExternalSystem`, the `Model` and the `application` explicity
    – *syncfiles* - Which synchronises multiple files, but uses a Regular Expression to find the required information about the `ExternalSystem`, the `Model` and the `application`.

Usage

Create your CSV file(s) with the data you need to synchronise with:

```
first_name,last_name,employee_id,action_flags,match_on
Andrew,Dodd,E1234,cu,employee_id
Some,Other-Guy,E4321,d,employee_id
```

Run one of the built in command (i.e. if you have a “Winner” Django model)s:

```
> python manage.py syncfile 'HRSystem' 'prizes' 'Winner' /tmp/the/file.csv
```

Check your application to see that Andrew Dodd is now a Winner and that other guy was deleted.

NOTE WELL: There is no need to write any Python to make this work!

1.4.2 Overview

Yeah, yeah, but whats the point?

It is quite common to need to load information from other computer systems into your Django application. There are also many ways to do this (manually, via a Restful API, through SQL/database cleverness) and there are a large number of existing tools to do this (see below).

However, often one must obtain and synchronise information about a model object from **multiple information sources** (e.g. the HR system, the ERP, that cool new Web API, Dave’s spreadsheet), and **continue to do it** in order to keep one’s Django app running properly. This project allows you to do just that.

Similar projects

There are a number of projects that are similar in nature but are (I believe) unsuitable for the reasons listed;

- django-synchro - Focussed on the synchonisation between databases (e.g. production & fail-over, production & testing)
  - This is quite a ‘full on’ project, and it is mainly focussed on synchronising two Django applications, not disparate systems
- django-external-data-sync - This is quite close in purpose (I think, I didn’t look at it too closely yet) to `django-nsync`, which is periodically synchronising with external data
  - Focusses on the infrastructure surrounding the ‘synchronisation’
  - Does not provide any synchronisation functions (you must subclass *Synchronizer*)
  - Not packaged on PyPI
- django-mapped-fields - Provides form fields to map structured data to models
  - Seems ok (I didn’t really look too closely)
  - Not designed for automated operation (i.e. it is about a Django Form workflow)
- django-csvimport - A generic importer tool for uploading CSV files to populate data
  - Extends the Admin Interface functionality, not really automated
- django-import-export - Application and library for importing and exporting
  - Looks to be excellent, certainly close in concept to `django-nsync`
  - Requires the creation of *ModelResource* subtypes to marshall the importing (i.e. requires code changes)

  - Focussed more on the 'Admin interface' interactions
  - (NB: Now that I'm writing up these docs and looking at this project it seems that they are quite similar)

1.4.3 Concepts

Model Actions

There are three key model actions (Create, Update, & Delete), and one action modifier (Force).

Create Action

This is used to 'create' a model object with the given information. If a matching object is found is will **NOT** create another one and it will **NOT** modify the existing object.

This action is always considered *'forced'*, so that it will override any non-empty/null defaults.

Update Action

This action will look for a model object and, if one is found, update it with the given information. It will **NOT** create an object.

- If **NOT** forced, the update will only affect fields whose current value is `None` or `''`
- If **forced**, the update will clobber any exiting value for the field

Delete Action

This action will look for a model object and, if one is found, attempt to delete it.

- If **forced**, the action will remove the object
- If **NOT** forced, the action will only delete the object if:
  - The target has external key mapping information; AND
  - The key mapping described exists; AND
  - The key mapping points to the corresponding object; AND
  - There are no other key mappings pointing to the object

NB: The *Delete action* will typically also manage the deletion of any corresponding `ExternalKeyMapping` objects associated with the object and the particular `ExternalSystem` that is performing the sync process (more information will be provided later).

Forced Actions

The option to *force* an action is provided to allow optionality to the synchroniser. It allows systems that 'might' have useful information but not the 'authoritative' answer to provide 'provisional' information in the sync process.

As mentioned above, the `force` option allows the following modifications of behaviour:

- `CREATE` actions - are *always* forced, to ensure they update non-`None` and non-`''` default values

- For `UPDATE` actions - it allows the action to forcibly replace the value in the corresponding field, rather than only replacing it if the value is `None` or `''`
- For `DELETE` actions - it allows the action to forcibly delete the model object, even if other systems have synchronised links to it

CSV Encodings

For use in CSV files (with the built in `syncfile` and `syncfiles` commands), the CSV file should include a column with the header `action_flags`. The values for the field can be:

Value	Meaning
c	Create only
u	Update only
d	Delete only
cu	Create and update
u*	Forced update
d*	Forced delete

The following values are pointless / not allowed:

Value	Meaning	Reason
	No action	Pointless, omit the row
c*	Forced create	Pointless, all creates are already forced
cd	Create and delete	Illegal, cannot request delete action with either create or update action.
ud	Update and delete	
cud	Create, update and delete	

Which object to act on?

The action uses the provided information to attempt to find (or guarantee the absence of) the object it should be acting upon. The *'provided information'* is the set of values used to set the fields in `CREATE` or `UPDATE` actions (NB: in all three cases it must contain the information to find the specific object).

Rules / Choices in design

The current choices for how this 'selection' behaves are:

- Always acts on a single object
- Found by the "`match_on`" value
  - Found by looking for an object that has the same 'values' as those provided for the model fields specified in the '`match_on`' list of fields. (huh? feeling lost?, it'll make sense in the examples)
  - The '`match_on`' column could be a 'unique' field with which to find your object, OR it could be a list of fields to use to find your object.
- Actions that target mulitple obects "could" be possible, but they are hard and probably not worth the trouble
- This would be trying to address some very general cases, which would be too hard to get logic correct for (I feel the risk of doing the wrong thing here accidentally would be too high)

- Computers are good at doing things quickly, get a computer to write the 'same' thing for the multiple targets and to execuse the request against multiple objects

Field Options

Fields are modified by using the `setattr()` built-in Python function. The field to update is based on the behaviour of this function to set the attribute based on the dictionary of information provided. **NB:** If the list of values includes 'fields' that are not part of the model object's definition, they will be ignored (more work to come here)

Referential Fields

One of the most important features is the ability to update 'referred to' fields, such as `Person` object that is assigned a company `Car` object.

This is specified by including the field and matchfields in the 'key' side of the values, concatenated with '=>' (you can see the CSV heritage creeping in here). For example, if you had classes like this:

```python
class Person(models.Model):
    first_name = models.CharField(
        blank=False,
        max_length=50,
    )
    last_name = models.CharField(
        blank=False,
        max_length=50,
    )
    assigned_car = models.ForeignKey(Car, blank=True, null=True)

class Car(models.Model):
    rego_number= models.CharField(max_length=10, unique=True)
    name = models.CharField(max_length=50)
```

You could load the assignment by synchronising with the following file for `Person` model:

Table 1.1: persons.csv

action_flags	match_on	first_name	last_name	assigned_car=>rego_number
cu	employee_id	Andrew	Dodd	BG29JL

However, you can also supply multiple inputs to a Referential assignment, which is especially handy for resolving situations where your models do not have a field that can be used to address them uniquely. For example, if you had classes like this instead (which is far more likely):

```python
class Person(models.Model):
    first_name = models.CharField(
        blank=False,
        max_length=50,
    )
    last_name = models.CharField(
        blank=False,
        max_length=50,
    )

class Car(models.Model):
    rego_number= models.CharField(max_length=10, unique=True)
    name = models.CharField(max_length=50)
    assigned_to = models.ForeignKey(Person, blank=True, null=True)
```

You could load the assignment by synchronising with the following file for `Car` model:

Table 1.2: cars.csv

action_flags	match_on	rego_number	name	assigned_to=>first_name	assigned_to=>last_name
cu	rego_number	BG29JL	Herman the Sherman	Andrew	Dodd

ExternalSystem & ExternalKeyMapping

This library also creates some objects to help keep track of the internal model objects modified by the external systems. With the purpose being to supply a way for users of the library to peform their own 'reverse' on which internal objects are being touched by which external systems. This is not particularly interesting, but it is perhaps worth checking out the `ExternalSystem` and `ExternalKeyMapping` classes.

These classes are also used to decide which 'object' is update if the '`match_on`' fields are changed (i.e. by an SQL UPDATE) but the 'external system key' remains the same.

But how?

It is probaby easiest to look at the examples page or have a look at the integration tests for the two out of the box commands.

1.4.4 Examples

The following are some examples of using the Nsync functionality. The following Django models will be used as the target models:

```python
class Person(models.Model):
    first_name = models.CharField(
        blank=False,
        max_length=50,
        verbose_name='First Name'
    )
    last_name = models.CharField(
        blank=False,
        max_length=50,
        verbose_name='Last Name'
    )
    age = models.IntegerField(blank=True, null=True)

    hair_colour = models.CharField(
        blank=False,
        max_length=50,
        default="Unknown")


class House(models.Model):
    address = models.CharField(max_length=100)
    country = models.CharField(max_length=100, blank=True)
    floors = models.IntegerField(blank=True, null=True)
    owner = models.ForeignKey(TestPerson, blank=True, null=True)
```

Example - Basic

Using this file:

Table 1.3: persons.csv

action_flags	match_on	first_name	last_name	employee_id
cu	employee_id	Andrew	Dodd	EMP1111
d*	employee_id	Some	Other-Guy	EMP2222
cu	employee_id	Captain	Planet	EMP3333
u*	employee_id	3.	Batman	EMP1234

And running this command:

```
> python manage.py syncfile TestSystem myapp Person persons.csv
```

Would:

- Create and/or update `myapp.Person` objects with Employee Ids EMP1111 & EMP3333. However, it would not update the two name fields if the objects already existed with non-blank fields.
- Delete any `myapp.Person` objects with Employee Id EMP2222.
- If a person with Employee Id EMP1234 exists, then it will forcibly update the name fields to 'C.' and 'Batman' respectively.

NB it would also:

- Create an `nsync.ExternalSystem` object with the name TestSystem, as the default is to create missing external systems. However, because there are no `external_key` values, no `ExternalKeyMapping` objects would be created.

Example - Basic with External Ids

Using this file:

Table 1.4: persons.csv

external_key	action_flags	match_on	first_name	last_name	employee_id
12212281	cu	employee_id	Andrew	Dodd	EMP1111
43719289	d*	employee_id	Some	Other-Guy	EMP2222
99999999	cu	employee_id	Captain	Planet	EMP3333
11235813	u*	employee_id	3.	Batman	EMP1234

And running this command:

```
> python manage.py syncfile TestSystem myapp Person persons.csv
```

Would:

- Perform all of steps in the 'Plain file' example
- Delete any `ExternalKeyMapping` objects that are for the 'TestSystem' and have the external key '43719289' (i.e. the record for Some Other-Guy).
- Create or update `ExternalKeyMapping` objects for each of the other three `myapp.Person` objects, which contain the `external_key` value.

Example - Basic with multiple match fields

Sometimes you might not have a ‘unique’ field to find your objects with (like ‘Employee Id’). In this instance, you can specify multiple fields for finding your object (separated with a space, ‘ ‘).

For example, using this file:

Table 1.5: persons.csv

action_flags	match_on	first_name	last_name	age
cu*	first_name last_name	Michael	Martin	30
cu*	first_name last_name	Martin	Martin	40
cu*	first_name last_name	Michael	Michael	50
cu*	first_name last_name	Martin	Michael	60

And running this command:

```
> python manage.py syncfile TestSystem myapp Person persons.csv
```

Would:

- Create and/or update four persons of various “Michael” and “Martin” name combinations
- Ensure they are updated/created with the correct age!

Example - Two or more systems

This is probably the main purpose of this library: the ability to synchronise from multiple systems.

Perhaps we need to synchronise from two data sources on housing information, one is the ‘when built’ information and the other is the ‘renovations’ information.

As-built data:

Table 1.6: AsBuiltDB_myapp_House.csv

external_key	action_flags	match_on	address	country	floors
111	cu	address	221B Baker Street	England	1
222	cu	address	Wayne Manor	Gotham City	2

Renovated data:

Table 1.7: RenovationsDB_myapp_House.csv

external_key	action_flags	match_on	address	floors
ABC123	u*	address	221B Baker Street	2
ABC456	u*	address	Wayne Manor	4
FOX123	u*	address	742 Evergreen Terrace	2

And running this command:

```
> python manage.py syncfiles AsBuiltDB_myapp_House.csv RenovationsDB_myapp_House.csv
```

Would:

- Use the **mutliple file command**, `syncfiles`, to perform multiple updates in one command
- Create the two houses from the ‘AsBuilt’ file
- Only update the `country` values of the two houses from the ‘AsBuilt’ file IFF the objects already existed but they did not have a value for `country`

- Forcibly set the `floors` attribute for the first two houses in the 'Renovations' file.
- Create 4 `ExternalKeyMapping` objects:

External System	Ext. Key	House Object
AsBuiltDB	111	212B Baker Street
RenovationsDB	ABC123	
AsBuiltDB	222	Wayne Manor
RenovationsDB	ABC456	

- Only update the `floors` attribute for "742 Evergreen Terrace" if the house already exists (and would then also create an `ExternalKeyMapping`)

Example - Referential fields

You can also manage referential fields with Nsync. For example, if you had the following people:

Table 1.8: Examples_myapp_Person.csv

external_key	action_flags	match_on	first_name	last_name	employee_id
1111	cu*	employee_id	Homer	Simpson	EMP1
2222	cu*	employee_id	Bruce	Wayne	EMP2
3333	cu*	employee_id	John	Wayne	EMP3

You could set their houses with a file like this:

Table 1.9: Examples_myapp_House.csv

external_key	action_flags	match_on	address	owner=>first_name
ABC456	cu*	address	Wayne Manor	Bruce
FOX123	cu*	address	742 Evergreen Terrace	Homer

The **"=>"** is used by Nsync to follow the the related field on the provided object.

Example - Referential field gotchas

The referential field update will ONLY be performed if the referred-to-fields target a single object. For example, if you had the following list of people:

Table 1.10: Examples_myapp_Person.csv

external_key	action_flags	match_on	first_name	last_name	employee_id
1111	cu*	employee_id	Homer	Simpson	EMP1
2222	cu*	employee_id	Homer	The Greek	EMP2
3333	cu*	employee_id	Bruce	Wayne	EMP3
4444	cu*	employee_id	Bruce	Lee	EMP4
5555	cu*	employee_id	John	Wayne	EMP5
6666	cu*	employee_id	Marge	Simpson	EMP6

The `owner=>first_name` from the previous example is insufficient to pick out a single person to link a house to (there are 2 Homers and 2 Bruces). Using just the `employee_id` field would work, but that piece of information may not be available in the system for houses.

Nsync allows you to specify multiple fields to use in order to 'filter' the correct object to create the link with. In this instance, this file would perform correctly:

Table 1.11: Examples_myapp_House.csv

external_key	action_flags	match_on	address	owner=>first_name	owner=>last_name
ABC456	cu*	address	Wayne Manor	Bruce	Wayne
FOX123	cu*	address	742 Evergreen Terrace	Homer	Simpson

Example - Complex Fields

If you want a more complex update you can:

- Write an extension to Nsync and submit a Pull Request! OR
- Extend your Django model with a custom setter

If your Person model has a photo ImageField, then you could add a custom handler to update the photo based on a provided file path:

```
class Person(models.Model):
    ...
    photo = models.ImageField(
        blank = True,
        null = True,
        max_length = 200,
        upload_to = 'person_photos',
    )
    ...

    @photo_filename.setter
    def photo_filename(self, file_path):
        ...
        Do the processing of the file to update the model
```

And then supply the photos with a file sync file like:

Table 1.12: persons.csv

action_flags	match_on	first_name	last_name	employee_id	photo_filename
cu*	employee_id	Andrew	Dodd	EMP1111	/tmp/photos/ugly_headshot.jpg

Example - Update uses external key mapping over matched object

This is an example that is to do with the changes for Issue 1

If Nsync is ‘updating’ objects but their ‘match fields’ change, Nsync will still update the ‘correct’ object.

A common occurrence of this is if the sync data is being produced from a database and an in-row update occurs which changes the match fields but leaves the ‘external key’ (i.e. an SQL ‘UPDATE ... WHERE ...’ statement).

E.g. A person table might look like this:

ID (a DB sequence)	Employee Number	Name
10123	EMP001	Andrew Dodd

This could be used to produce an Nsync input CSV like this:

external_key	action_flags	match_on	employee_number	name
10123	cu*	employee_number	EMP001	Andrew Dodd

This would result in an "Andrew Dodd, EMP001" Person object being created and/or updated with an *ExternalKeyMapping* object holding the '10123' id and a link to Andrew.

If Andrew became a contractor instead of an employee, perhaps the table could be updated to look like this:

ID (a DB sequence)	Employee Number	Name
10123	CONT999	Andrew Dodd

This would then produce an Nsync input CSV like this:

external_key	action_flags	match_on	employee_number	name
10123	cu*	employee_number	CONT999	Andrew Dodd

Nsync will use the *ExternalKeyMapping* object if it is available instead of relying on the 'match fields'. In this case, the resulting action will cause the Andrew Dodd object to change its 'employee_number'. This is instead of Nsync using the 'employee_number' for finding Andrew.

NB: In this instance, Nsync will also delete any objects that have the 'new' match field but are not pointed to by the external key.

Example - Delete tricks

This is a list of tricky / gotchas to be aware of when deleting objects.

When syncing from external systems that have external key mappings, it is probably best to use the 'unforced delete'. This ensures that an object is not removed until all of the external systems think it should be removed.

If using 'forced delete', beware that (depending on which sync policy you use) you may end up with different systems fighting over the existence of an object (i.e. one system creating the object, then another deleting it in the same sync).

A system without external key mappings cannot delete objects if it uses an 'unforced delete'. The reason for this is that the 'unforced delete' only removes the model object IF AND ONLY IF it is the last remaining external key mapping. Thus, if a system without external key mappings is the source-of-truth for the removal of an object, you must use the 'forced delete' for it to be able to remove the objects.

Alternative Sync Policies

The out-of-the-box sync policies are pretty straightforward and are probably worth a read (see the `policies.py` file). The system is made so that it is pretty easy for you to define your own custom policy and write a command (similar to the ones in Nsync) to use it.

Some examples of alternative policies might be:

- Run deletes before creates and updates
- Search and execute certain actions before all others

1.5 Project Infomation

1.5.1 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/andrewdodd/django-nsync/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

django-nsync could always use more documentation, whether as part of the official django-nsync docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/andrewdodd/django-nsync/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here's how to set up *django-nsync* for local development.

1. Fork the *django-nsync* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/django-nsync.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
    $ mkvirtualenv django-nsync
    $ cd django-nsync/
    $ python setup.py develop
```

4. Create a branch for local development:

```
    $ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 src tests
$ python setup.py test
$ tox
```

To get flake8 and tox, just pip install them into your virtualenv. NB: Don't worry about flake8 issues in the migrations files or if URLs are too long.

6. Commit your changes and push your branch to GitHub:

```
    $ git add .
    $ git commit -m "Your detailed description of your changes."
    $ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.3+, and for PyPy. Check https://travis-ci.org/andrewdodd/django-nsync/pull_requests and make sure that the tests pass for all supported Python versions.

Tips

To run a subset of tests:

```
$ python -m unittest tests.test_nsync
```

1.5.2 Credits

Development Lead

- Andrew Dodd <andrew.john.dodd@gmail.com>

Contributors

- Chris Snell

1.5.3 History

0.1.0 (2015-10-02)

- First release on PyPI.