django-import-export Documentation

Release 0.6.1

Bojan Mihelac

Dec 11, 2017

User Guide

django-import-export is a Django application and library for importing and exporting data with included admin integration.

Features:

- support multiple formats (Excel, CSV, JSON, ... and everything else that tablib supports)
- admin integration for importing
- preview import changes
- admin integration for exporting
- export data respecting admin filters

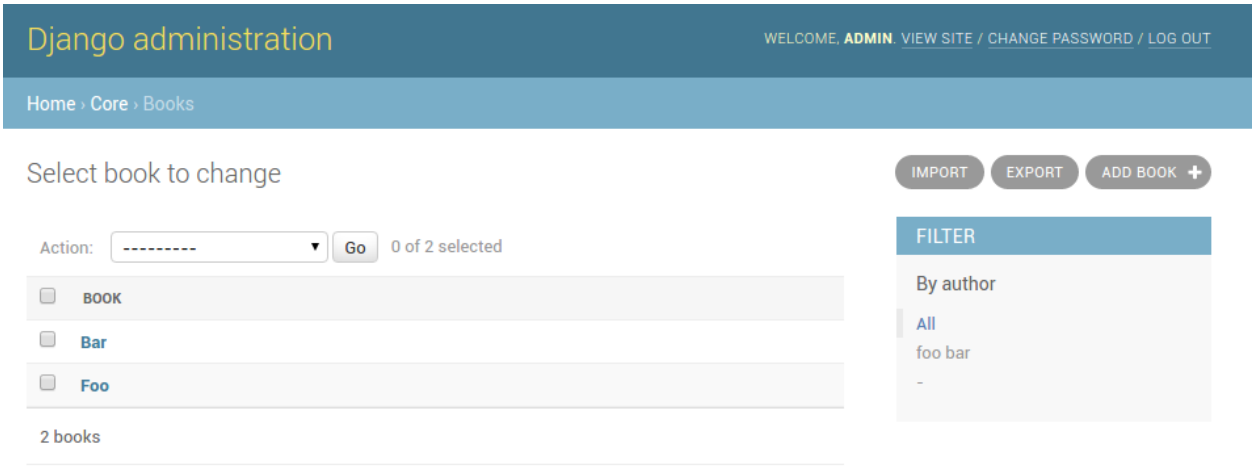

Fig. 1: A screenshot of the change view with Import and Export buttons.

CHAPTER 1

Installation and configuration

django-import-export is available on the Python Package Index (PyPI), so it can be installed with standard Python tools like `pip` or `easy_install`:

```
$ pip install django-import-export
```

Alternatively, you can install the git repository directly to obtain the development version:

```
$ pip install -e git+https://github.com/django-import-export/django-import-export.git
↪#egg=django-import-export
```

Now, you're good to go, unless you want to use django-import-export from the admin as well. In this case, you need to add it to your `INSTALLED_APPS` and let Django collect its static files.

```
# settings.py
INSTALLED_APPS = (
    ...
    'import_export',
)
```

```
$ python manage.py collectstatic
```

All prequisites are set up! See *Getting started* to learn how to use django-import-export in your project.

1.1 Settings

You can use the following directives in your settings file:

IMPORT_EXPORT_USE_TRANSACTIONS Global setting controls if resource importing should use database transactions. Default is `False`.

IMPORT_EXPORT_SKIP_ADMIN_LOG Global setting controls if creating log entries for the admin changelist should be skipped when importing resource. The *skip_admin_log* attribute of *ImportMixin* is checked first,

which defaults to `None`. If not found, this global option is used. This will speed up importing large datasets, but will lose changing logs in the admin changelist view. Default is `False`.

`IMPORT_EXPORT_TMP_STORAGE_CLASS` Global setting for the class to use to handle temporary storage of the uploaded file when importing from the admin using an *ImportMixin*. The *tmp_storage_class* attribute of *ImportMixin* is checked first, which defaults to `None`. If not found, this global option is used. Default is `TempFolderStorage`.

1.2 Example app

There's an example application that showcases what django-import-export can do. You can run it via:

```
cd tests
./manage.py runserver
```

Username and password for admin are `admin` and `password`.

CHAPTER 2

Getting started

For example purposes, we'll use a simplified book app. Here is our `models.py`:

```
# app/models.py

class Author(models.Model):
    name = models.CharField(max_length=100)

    def __unicode__(self):
        return self.name


class Category(models.Model):
    name = models.CharField(max_length=100)

    def __unicode__(self):
        return self.name


class Book(models.Model):
    name = models.CharField('Book name', max_length=100)
    author = models.ForeignKey(Author, blank=True, null=True)
    author_email = models.EmailField('Author email', max_length=75, blank=True)
    imported = models.BooleanField(default=False)
    published = models.DateField('Published', blank=True, null=True)
    price = models.DecimalField(max_digits=10, decimal_places=2, null=True,␣
↪blank=True)
    categories = models.ManyToManyField(Category, blank=True)

    def __unicode__(self):
        return self.name
```

2.1 Creating import-export resource

To integrate *django-import-export* with our `Book` model, we will create a *ModelResource* class in `admin.py` that will describe how this resource can be imported or exported:

```
# app/admin.py

from import_export import resources
from core.models import Book

class BookResource(resources.ModelResource):

    class Meta:
        model = Book
```

2.2 Exporting data

Now that we have defined a *ModelResource* class, we can export books:

```
>>> from app.admin import BookResource
>>> dataset = BookResource().export()
>>> print dataset.csv
id,name,author,author_email,imported,published,price,categories
2,Some book,1,,0,2012-12-05,8.85,1
```

2.3 Customize resource options

By default *ModelResource* introspects model fields and creates *Field*-attributes with an appropriate *Widget* for each field.

To affect which model fields will be included in an import-export resource, use the `fields` option to whitelist fields:

```
class BookResource(resources.ModelResource):

    class Meta:
        model = Book
        fields = ('id', 'name', 'price',)
```

Or the `exclude` option to blacklist fields:

```
class BookResource(resources.ModelResource):

    class Meta:
        model = Book
        exclude = ('imported', )
```

An explicit order for exporting fields can be set using the `export_order` option:

```
class BookResource(resources.ModelResource):

    class Meta:
        model = Book
```

```
        fields = ('id', 'name', 'author', 'price',)
        export_order = ('id', 'price', 'author', 'name')
```

The default field for object identification is `id`, you can optionally set which fields are used as the `id` when importing:

```
class BookResource(resources.ModelResource):

    class Meta:
        model = Book
        import_id_fields = ('isbn',)
        fields = ('isbn', 'name', 'author', 'price',)
```

When defining *ModelResource* fields it is possible to follow model relationships:

```
class BookResource(resources.ModelResource):

    class Meta:
        model = Book
        fields = ('author__name',)
```

Note: Following relationship fields sets `field` as readonly, meaning this field will be skipped when importing data.

By default all records will be imported, even if no changes are detected. This can be changed setting the `skip_unchanged` option. Also, the `report_skipped` option controls whether skipped records appear in the import `Result` object, and if using the admin whether skipped records will show in the import preview page:

```
class BookResource(resources.ModelResource):

    class Meta:
        model = Book
        skip_unchanged = True
        report_skipped = False
        fields = ('id', 'name', 'price',)
```

See also:

Resources

2.4 Declaring fields

It is possible to override a resource field to change some of its options:

```
from import_export import fields

class BookResource(resources.ModelResource):
    published = fields.Field(column_name='published_date')

    class Meta:
        model = Book
```

Other fields that don't exist in the target model may be added:

```
from import_export import fields
```

```
class BookResource(resources.ModelResource):
    myfield = fields.Field(column_name='myfield')

    class Meta:
        model = Book
```

See also:

Fields Available field types and options.

2.5 Advanced data manipulation

Not all data can be easily extracted from an object/model attribute. In order to turn complicated data model into a (generally simpler) processed data structure, `dehydrate_<fieldname>` method should be defined:

```
from import_export import fields

class BookResource(resources.ModelResource):
    full_title = fields.Field()

    class Meta:
        model = Book

    def dehydrate_full_title(self, book):
        return '%s by %s' % (book.name, book.author.name)
```

2.6 Customize widgets

A *ModelResource* creates a field with a default widget for a given field type. If the widget should be initialized with different arguments, set the `widgets` dict.

In this example widget, the `published` field is overriden to use a different date format. This format will be used both for importing and exporting resource.

```
class BookResource(resources.ModelResource):

    class Meta:
        model = Book
        widgets = {
                'published': {'format': '%d.%m.%Y'},
                }
```

See also:

Widgets available widget types and options.

2.7 Importing data

Let's import some data!

```
>>> import tablib
>>> from import_export import resources
>>> from core.models import Book
>>> book_resource = resources.modelresource_factory(model=Book)()
>>> dataset = tablib.Dataset(['', 'New book'], headers=['id', 'name'])
>>> result = book_resource.import_data(dataset, dry_run=True)
>>> print result.has_errors()
False
>>> result = book_resource.import_data(dataset, dry_run=False)
```

In the fourth line we use *modelresource_factory()* to create a default *ModelResource*. The ModelResource class created this way is equal to the one shown in the example in section *Creating import-export resource*.

In fifth line a `Dataset` with columns `id` and `name`, and one book entry, are created. A field for a primary key field (in this case, `id`) always needs to be present.

In the rest of the code we first pretend to import data using *import_data()* and `dry_run` set, then check for any errors and actually import data this time.

See also:

Import data workflow for a detailed description of the import workflow and its customization options.

2.7.1 Deleting data

To delete objects during import, implement the *for_delete()* method on your *Resource* class.

The following is an example resource which expects a `delete` field in the dataset. An import using this resource will delete model instances for rows that have their column `delete` set to `1`:

```
class BookResource(resources.ModelResource):
    delete = fields.Field(widget=widgets.BooleanWidget())

    def for_delete(self, row, instance):
        return self.fields['delete'].clean(row)

    class Meta:
        model = Book
```

2.8 Signals

To hook in the import export workflow, you can connect to `post_import`, `post_export` signals:

```
from django.dispatch import receiver
from import_export.signals import post_import, post_export

@receiver(post_import, dispatch_uid='balabala...')
def _post_import(model, **kwargs):
    # model is the actual model instance which after import
    pass

@receiver(post_export, dispatch_uid='balabala...')
def _post_export(model, **kwargs):
    # model is the actual model instance which after export
    pass
```

2.9 Admin integration

2.9.1 Exporting via list filters

Admin integration is achieved by subclassing *ImportExportModelAdmin* or one of the available mixins (*ImportMixin*, *ExportMixin*, *ImportExportMixin*):

```
# app/admin.py
from import_export.admin import ImportExportModelAdmin

class BookAdmin(ImportExportModelAdmin):
    resource_class = BookResource
```

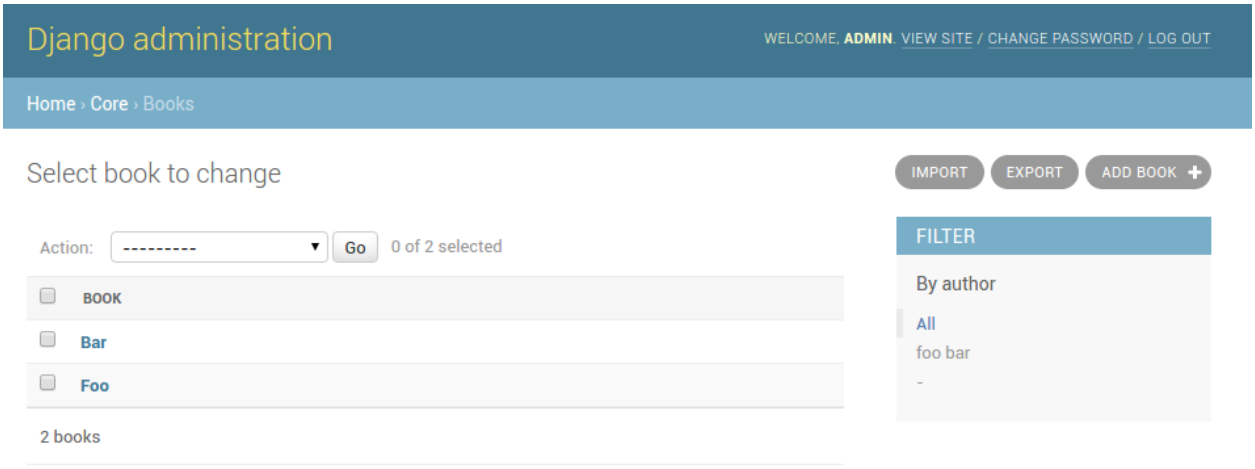

Fig. 2.1: A screenshot of the change view with Import and Export buttons.

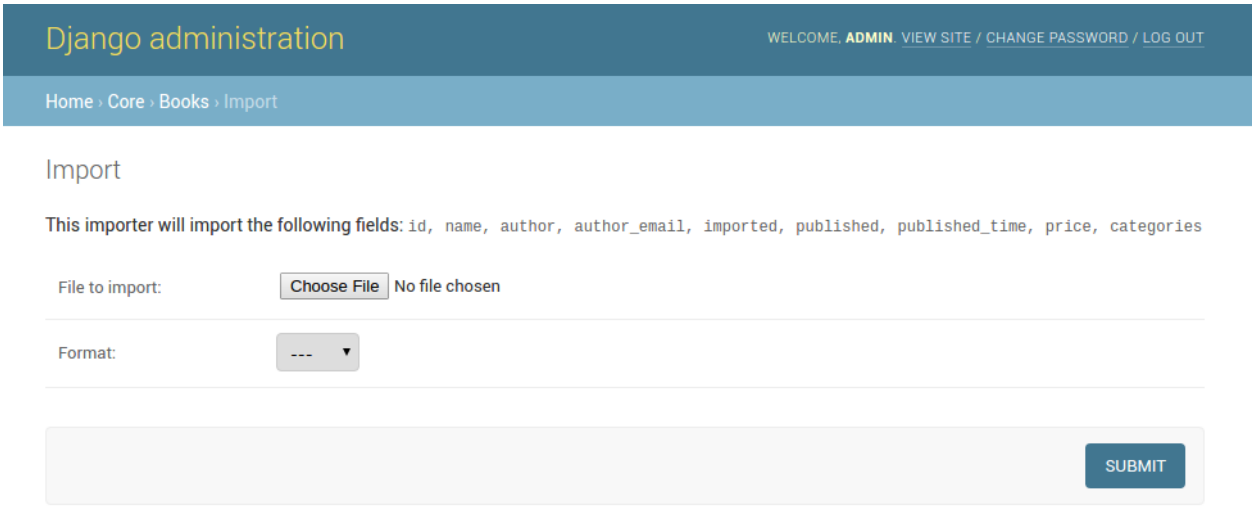

Fig. 2.2: A screenshot of the import view.

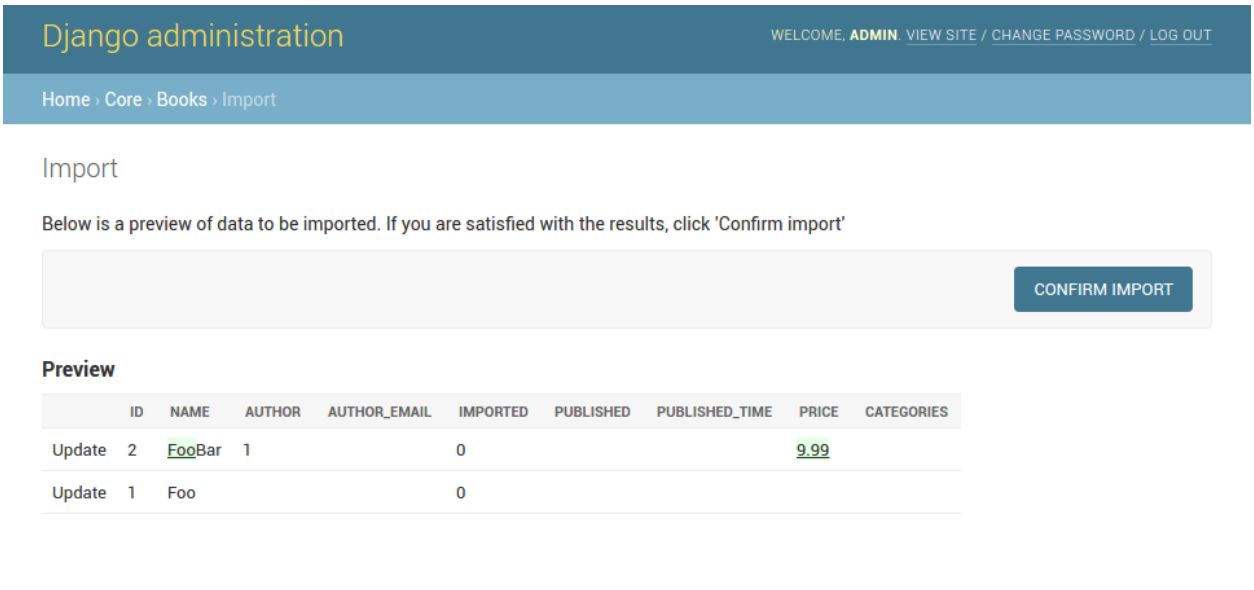

Fig. 2.3: A screenshot of the confirm import view.

2.9.2 Exporting via admin action

Another approach to exporting data is by subclassing *ImportExportActionModelAdmin* which implements export as an admin action. As a result it's possible to export a list of objects selected on the change list page:

```
# app/admin.py
from import_export.admin import ImportExportActionModelAdmin

class BookAdmin(ImportExportActionModelAdmin):
    pass
```

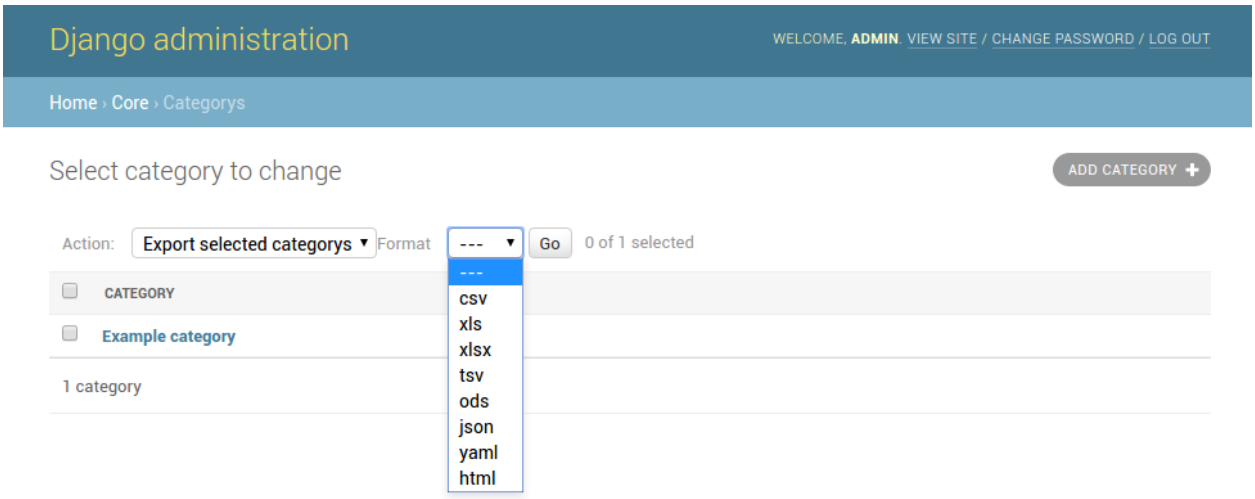

Fig. 2.4: A screenshot of the change view with Import and Export as an admin action.

See also:

Admin available mixins and options.

CHAPTER 3

Import data workflow

This document describes the import data workflow in detail, with hooks that enable customization of the import process. The central aspect of the import process is a resource's *import_data()* method which is explained below.

import_data(*dataset*, *dry_run=False*, *raise_errors=False*)

The *import_data()* method of *Resource* is responsible for importing data from a given dataset.

`dataset` is required and expected to be a `tablib.Dataset` with a header row.

`dry_run` is a Boolean which determines if changes to the database are made or if the import is only simulated. It defaults to `False`.

`raise_errors` is a Boolean. If `True`, import should raise errors. The default is `False`, which means that eventual errors and traceback will be saved in `Result` instance.

This is what happens when the method is invoked:

1. First, a new *Result* instance, which holds errors and other information gathered during the import, is initialized.

   Then, an `InstanceLoader` responsible for loading existing instances is intitalized. A different *BaseInstanceLoader* can be specified via *ResourceOptions*'s `instance_loader_class` attribute. A *CachedInstanceLoader* can be used to reduce number of database queries. See the source for available implementations.

2. The *before_import()* hook is called. By implementing this method in your resource, you can customize the import process.

3. Each row of the to-be-imported dataset is processed according to the following steps:

   (a) The *before_import_row()* hook is called to allow for row data to be modified before it is imported

   (b) *get_or_init_instance()* is called with current *BaseInstanceLoader* and current row of the dataset, returning an object and a Boolean declaring if the object is newly created or not.

   If no object can be found for the current row, `init_instance()` is invoked to initialize an object.

   As always, you can override the implementation of `init_instance()` to customized how the new object is created (i.e. set default values).

(c) *for_delete()* is called to determine if the passed `instance` should be deleted. In this case, the import process for the current row is stopped at this point.

(d) If the instance was not deleted in the previous step, *import_obj()* is called with the `instance` as current object, `row` as current row and `dry run`.

import_field() is called for each field in *Resource* skipping many- to-many fields. Many-to-many fields are skipped because they require instances to have a primary key and therefore assignment is postponed to when the object has already been saved.

import_field() in turn calls *save()*, if `Field.attribute` is set and `Field.column_name` exists in the given row.

(e) It then is determined whether the newly imported object is different from the already present object and if therefore the given row should be skipped or not. This is handled by calling *skip_row()* with `original` as the original object and `instance` as the current object from the dataset.

If the current row is to be skipped, `row_result.import_type` is set to `IMPORT_TYPE_SKIP`.

(f) If the current row is not to be skipped, *save_instance()* is called and actually saves the instance when `dry_run` is not set.

There are two hook methods (that by default do nothing) giving you the option to customize the import process:

- *before_save_instance()*
- *after_save_instance()*

Both methods receive `instance` and `dry_run` arguments.

(g) *save_m2m()* is called to save many to many fields.

(h) `RowResult` is assigned with a diff between the original and the imported object fields, as well as and `import_type` attribute which states whether the row is new, updated, skipped or deleted.

If an exception is raised during row processing and *import_data()* was invoked with `raise_errors=False` (which is the default) the particular traceback is appended to `RowResult` as well.

If either the row was not skipped or the *Resource* is configured to report skipped rows, the `RowResult` is appended to the *Result*

(i) The *after_import_row()* hook is called

4. The *Result* is returned.

3.1 Transaction support

If transaction support is enabled, whole import process is wrapped inside transaction and rollbacked or committed respectively. All methods called from inside of `import_data` (create / delete / update) receive `False` for `dry_run` argument.

CHAPTER 4

Changelog

4.1 0.6.1 (2017-12-04)

- Refactors and optimizations (#686, #632, #684, #636, #631, #629, #635, #683)
- Travis tests for Django 2.0.x (#691)

4.2 0.6.0 (2017-11-23)

- Refactor import_row call by using keyword arguments (#585)
- Added {{ block.super }} call in block bodyclass in admin/base_site.html (#582)
- Add support for the Django DurationField with DurationWidget (#575)
- GitHub bmihelac -> django-import-export Account Update (#574)
- Add intersphinx links to documentation (#572)
- Add Resource.get_import_fields() (#569)
- Fixed readme mistake (#568)
- Bugfix/fix m2m widget clean (#515)
- Allow injection of context data for template rendered by import_action() and export_action() (#544)
- Bugfix/fix exception in generate_log_entries() (#543)
- Process import dataset and result in separate methods (#542)
- Bugfix/fix error in converting exceptions to strings (#526)
- Fix admin integration tests for the new “Import finished...” message, update Czech translations to 100% coverage. (#596)
- Make import form type easier to override (#604)

- Add saves_null_values attribute to Field to control whether null values are saved on the object (#611)
- Add Bulgarian translations (#656)
- Add django 1.11 to TravisCI (#621)
- Make Signals code example format correctly in documentation (#553)
- Add Django as requirement to setup.py (#634)
- Update import of reverse for django 2.x (#620)
- Add Django-version classifiers to setup.py's CLASSIFIERS (#616)
- Some fixes for Django 2.0 (#672)
- Strip whitespace when looking up ManyToMany fields (#668)
- Fix all ResourceWarnings during tests in Python 3.x (#637)
- Remove downloads count badge from README since shields.io no longer supports it for PyPi (#677)
- Add coveralls support and README badge (#678)

4.3 0.5.1 (2016-09-29)

- French locale not in pypi (#524)
- Bugfix/fix undefined template variables (#519)

4.4 0.5.0 (2016-09-01)

- Hide default value in diff when importing a new instance (#458)
- Append rows to Result object via function call to allow overriding (#462)
- Add get_resource_kwargs to allow passing request to resource (#457)
- Expose Django user to get_export_data() and export() (#447)
- Add before_export and after_export hooks (#449)
- fire events post_import, post_export events (#440)
- add **kwargs to export_data / create_dataset
- Add before_import_row() and after_import_row() (#452)
- Add get_export_fields() to Resource to control what fields are exported (#461)
- Control user-visible fields (#466)
- Fix diff for models using ManyRelatedManager
- Handle already cleaned objects (#484)
- Add after_import_instance hook (#489)
- Use optimized xlsx reader (#482)
- Adds resource_class of BookResource (re-adds) in admin docs (#481)
- Require POST method for process_import() (#478)
- Add SimpleArrayWidget to support use of django.contrib.postgres.fields.ArrayField (#472)

- Add new Diff class (#477)
- Fix #375: add row to widget.clean(), obj to widget.render() (#479)
- Restore transactions for data import (#480)
- Refactor the import-export templates (#496)
- Update doc links to the stable version, update rtfd to .io (#507)
- Fixed typo in the Czech translation (#495)

4.5 0.4.5 (2016-04-06)

- Add FloatWidget, use with model fields models.FloatField (#433)
- Fix default values in fields (#431, #364)

  Field constructor *default* argument is NOT_PROVIDED instead of None Field clean method checks value against *Field.empty_values* [None, ‘’]

4.6 0.4.4 (2016-03-22)

- FIX: No static/ when installed via pip #427
- Add total # of imports and total # of updates to import success msg

4.7 0.4.3 (2016-03-08)

- fix MediaStorage does not respect the read_mode parameter (#416)
- Reset SQL sequences when new objects are imported (#59)
- Let Resource rollback if import throws exception (#377)
- Fixes error when a single value is stored in m2m relation field (#177)
- Add support for django.db.models.TimeField (#381)

4.8 0.4.2 (2015-12-18)

- add xlsx import support

4.9 0.4.1 (2015-12-11)

- fix for fields with a dyanmic default callable (#360)

4.10 0.4.0 (2015-12-02)

- Add Django 1.9 support
- Django 1.4 is not supported (#348)

4.11 0.3.1 (2015-11-20)

- FIX: importing csv in python 3

4.12 0.3 (2015-11-20)

- FIX: importing csv UnicodeEncodeError introduced in 0.2.9 (#347)

4.13 0.2.9 (2015-11-12)

- Allow Field.save() relation following (#344)
- Support default values on fields (and models) (#345)
- m2m widget: allow trailing comma (#343)
- Open csv files as text and not binary (#127)

4.14 0.2.8 (2015-07-29)

- use the IntegerWidget for database-fields of type BigIntegerField (#302)
- make datetime timezone aware if USE_TZ is True (#283).
- Fix 0 is interpreted as None in number widgets (#274)
- add possibility to override tmp storage class (#133, #251)
- better error reporting (#259)

4.15 0.2.7 (2015-05-04)

- Django 1.8 compatibility
- add attribute inheritance to Resource (#140)
- make the filename and user available to import_data (#237)
- Add to_encoding functionality (#244)
- Call before_import before creating the instance_loader - fixes #193

4.16 0.2.6 (2014-10-09)

- added use of get_diff_headers method into import.html template (#158)
- Try to use OrderedDict instead of SortedDict, which is deprecated in Django 1.7 (#157)
- fixed #105 unicode import
- remove invalid form action "form_url" #154

4.17 0.2.5 (2014-10-04)

- Do not convert numeric types to string (#149)
- implement export as an admin action (#124)

4.18 0.2.4 (2014-09-18)

- fix: get_value raised attribute error on model method call
- Fixed XLS import on python 3. Optimized loop
- Fixed properly skipping row marked as skipped when importing data from the admin interface.
- Allow Resource.export to accept iterables as well as querysets
- Improve error messages
- FIX: Properly handle NullBoleanField (#115) - Backward Incompatible Change previously None values were handled as false

4.19 0.2.3 (2014-07-01)

- Add separator and field keyword arguments to ManyToManyWidget
- FIX: No support for dates before 1900 (#93)

4.20 0.2.2 (2014-04-18)

- RowResult now stores exception object rather than it's repr
- Admin integration - add EntryLog object for each added/updated/deleted instance

4.21 0.2.1 (2014-02-20)

- FIX import_file_name form field can be use to access the filesystem (#65)

4.22 0.2.0 (2014-01-30)

- Python 3 support

4.23 0.1.6 (2014-01-21)

- Additional hooks for customizing the workflow (#61)

4.24 0.1.5 (2013-11-29)

- Prevent queryset caching when exporting (#44)
- Allow unchanged rows to be skipped when importing (#30)
- Update tests for Django 1.6 (#57)
- Allow different `ResourceClass` to be used in `ImportExportModelAdmin` (#49)

4.25 0.1.4

- Use *field_name* instead of *column_name* for field dehydration, FIX #36
- Handle OneToOneField, FIX #17 - Exception when attempting access something on the related_name.
- FIX #23 - export filter not working

4.26 0.1.3

- Fix packaging
- DB transactions support for importing data

4.27 0.1.2

- support for deleting objects during import
- bug fixes
- Allowing a field to be 'dehydrated' with a custom method
- added documentation

4.28 0.1.1

- added ExportForm to admin integration for choosing export file format
- refactor admin integration to allow better handling of specific formats supported features and better handling of reading text files

- include all avialable formats in Admin integration
- bugfixes

4.29 0.1.0

- Refactor api

CHAPTER 5

Admin

For instructions on how to use the models and mixins in this module, please refer to *Admin integration*.

class import_export.admin.**ExportActionModelAdmin**(**args*, ***kwargs*)
Subclass of ModelAdmin with export functionality implemented as an admin action.

export_admin_action(*request*, *queryset*)
Exports the selected rows using file_format.

class import_export.admin.**ExportMixin**
Export mixin.

get_export_data(*file_format*, *queryset*, **args*, ***kwargs*)
Returns file_format representation for given queryset.

get_export_formats()
Returns available export formats.

get_export_queryset(*request*)
Returns export queryset.

Default implementation respects applied search and filters.

get_export_resource_class()
Returns ResourceClass to use for export.

class import_export.admin.**ImportExportActionModelAdmin**(**args*, ***kwargs*)
Subclass of ExportActionModelAdmin with import/export functionality. Export functionality is implemented as an admin action.

class import_export.admin.**ImportExportMixin**
Import and export mixin.

class import_export.admin.**ImportExportModelAdmin**(*model*, *admin_site*)
Subclass of ModelAdmin with import/export functionality.

class import_export.admin.**ImportMixin**
Import mixin.

get_import_form()
Get the form type used to read the import format and file.

get_import_formats()
Returns available import formats.

get_import_resource_class()
Returns ResourceClass to use for import.

import_action(*request*, **args*, ***kwargs*)
Perform a dry_run of the import to make sure the import will not result in errors. If there where no error, save the user uploaded file to a local temp file that will be used by 'process_import' for the actual import.

process_import(*request*, **args*, ***kwargs*)
Perform the actual import action (after the user has confirmed the import)

CHAPTER 6

Resources

6.1 Resource

class import_export.resources.**Resource**

Resource defines how objects are mapped to their import and export representations and handle importing and exporting data.

after_delete_instance(*instance*, *dry_run*)
Override to add additional logic. Does nothing by default.

after_export(*queryset*, *data*, **args*, ***kwargs*)
Override to add additional logic. Does nothing by default.

after_import(*dataset*, *result*, *using_transactions*, *dry_run*, ***kwargs*)
Override to add additional logic. Does nothing by default.

after_import_instance(*instance*, *new*, ***kwargs*)
Override to add additional logic. Does nothing by default.

after_import_row(*row*, *row_result*, ***kwargs*)
Override to add additional logic. Does nothing by default.

after_save_instance(*instance*, *using_transactions*, *dry_run*)
Override to add additional logic. Does nothing by default.

before_delete_instance(*instance*, *dry_run*)
Override to add additional logic. Does nothing by default.

before_export(*queryset*, **args*, ***kwargs*)
Override to add additional logic. Does nothing by default.

before_import(*dataset*, *using_transactions*, *dry_run*, ***kwargs*)
Override to add additional logic. Does nothing by default.

before_import_row(*row*, ***kwargs*)
Override to add additional logic. Does nothing by default.

before_save_instance(*instance*, *using_transactions*, *dry_run*)
Override to add additional logic. Does nothing by default.

delete_instance(*instance*, *using_transactions=True*, *dry_run=False*)
Calls `instance.delete()` as long as `dry_run` is not set.

export(*queryset=None*, **args*, ***kwargs*)
Exports a resource.

for_delete(*row*, *instance*)
Returns `True` if `row` importing should delete instance.

Default implementation returns `False`. Override this method to handle deletion.

get_diff_headers()
Diff representation headers.

classmethod get_error_result_class()
Returns the class used to store an error resulting from an import.

classmethod get_field_name(*field*)
Returns the field name for a given field.

get_fields(***kwargs*)
Returns fields sorted according to `export_order`.

get_instance(*instance_loader*, *row*)
Calls the *InstanceLoader*.

get_or_init_instance(*instance_loader*, *row*)
Either fetches an already existing instance or initializes a new one.

classmethod get_result_class()
Returns the class used to store the result of an import.

classmethod get_row_result_class()
Returns the class used to store the result of a row import.

import_data(*dataset*, *dry_run=False*, *raise_errors=False*, *use_transactions=None*, *collect_failed_rows=False*, ***kwargs*)
Imports data from `tablib.Dataset`. Refer to *Import data workflow* for a more complete description of the whole import process.

Parameters

- **dataset** – A `tablib.Dataset`
- **raise_errors** – Whether errors should be printed to the end user or raised regularly.
- **use_transactions** – If `True` the import process will be processed inside a transaction.
- **collect_failed_rows** – If `True` the import process will collect failed rows.
- **dry_run** – If `dry_run` is set, or an error occurs, if a transaction is being used, it will be rolled back.

import_field(*field*, *obj*, *data*)
Calls *import_export.fields.Field.save()* if `Field.attribute` and `Field.column_name` are found in `data`.

import_obj(*obj*, *data*, *dry_run*)
Traverses every field in this Resource and calls *import_field()*.

import_row(*row*, *instance_loader*, *using_transactions=True*, *dry_run=False*, ***kwargs*)
Imports data from `tablib.Dataset`. Refer to *Import data workflow* for a more complete description of the whole import process.

Parameters

- **row** – A `dict` of the row to import
- **instance_loader** – The instance loader to be used to load the row
- **using_transactions** – If `using_transactions` is set, a transaction is being used to wrap the import
- **dry_run** – If `dry_run` is set, or error occurs, transaction will be rolled back.

save_instance(*instance*, *using_transactions=True*, *dry_run=False*)
Takes care of saving the object to the database.

Keep in mind that this is done by calling `instance.save()`, so objects are not created in bulk!

save_m2m(*obj*, *data*, *using_transactions*, *dry_run*)
Saves m2m fields.

Model instance need to have a primary key value before a many-to-many relationship can be used.

skip_row(*instance*, *original*)
Returns `True` if `row` importing should be skipped.

Default implementation returns `False` unless skip_unchanged == True. Override this method to handle skipping rows meeting certain conditions.

6.2 ModelResource

class `import_export.resources.`**ModelResource**
ModelResource is Resource subclass for handling Django models.

after_import(*dataset*, *result*, *using_transactions*, *dry_run*, ***kwargs*)
Reset the SQL sequences after new objects are imported

classmethod field_from_django_field(*field_name*, *django_field*, *readonly*)
Returns a Resource Field instance for the given Django model field.

get_import_id_fields()

get_queryset()
Returns a queryset of all objects for this model. Override this if you want to limit the returned queryset.

init_instance(*row=None*)
Initializes a new Django model.

classmethod widget_from_django_field(*f*, *default=<class 'import_export.widgets.Widget'>*)
Returns the widget that would likely be associated with each Django type.

classmethod widget_kwargs_for_field(*field_name*)
Returns widget kwargs for given field_name.

6.3 ResourceOptions (Meta)

class `import_export.resources.`**ResourceOptions**
The inner Meta class allows for class-level configuration of how the Resource should behave. The following

options are available:

6.4 modelresource_factory

resources.**modelresource_factory**(*model*, *resource_class=<class 'import_export.resources.ModelResource'>*)

Factory for creating `ModelResource` class for given Django model.

CHAPTER 7

Widgets

class import_export.widgets.**Widget**
A Widget takes care of converting between import and export representations.

This is achieved by the two methods, *clean()* and *render()*.

clean(*value*, *row=None*, **args*, ***kwargs*)
Returns an appropriate Python object for an imported value.

For example, if you import a value from a spreadsheet, *clean()* handles conversion of this value into the corresponding Python object.

Numbers or dates can be *cleaned* to their respective data types and don't have to be imported as Strings.

render(*value*, *obj=None*)
Returns an export representation of a Python value.

For example, if you have an object you want to export, *render()* takes care of converting the object's field to a value that can be written to a spreadsheet.

class import_export.widgets.**IntegerWidget**
Widget for converting integer fields.

class import_export.widgets.**DecimalWidget**
Widget for converting decimal fields.

class import_export.widgets.**CharWidget**
Widget for converting text fields.

class import_export.widgets.**BooleanWidget**
Widget for converting boolean fields.

class import_export.widgets.**DateWidget**(*format=None*)
Widget for converting date fields.

Takes optional `format` parameter.

class import_export.widgets.**TimeWidget**(*format=None*)
Widget for converting time fields.

Takes optional `format` parameter.

class `import_export.widgets.`**`DateTimeWidget`**(*format=None*)
Widget for converting date fields.

Takes optional `format` parameter. If none is set, either `settings.DATETIME_INPUT_FORMATS` or `"%Y-%m-%d %H:%M:%S"` is used.

class `import_export.widgets.`**`DurationWidget`**
Widget for converting time duration fields.

class `import_export.widgets.`**`ForeignKeyWidget`**(*model*, *field='pk'*, **args*, ***kwargs*)
Widget for a `ForeignKey` field which looks up a related model using "natural keys" in both export an import.

The lookup field defaults to using the primary key (`pk`) as lookup criterion but can be customised to use any field on the related model.

Unlike specifying a related field in your resource like so...

```
class Meta:
    fields = ('author__name',)
```

...using a *`ForeignKeyWidget`* has the advantage that it can not only be used for exporting, but also importing data with foreign key relationships.

Here's an example on how to use *`ForeignKeyWidget`* to lookup related objects using `Author.name` instead of `Author.pk`:

```
class BookResource(resources.ModelResource):
    author = fields.Field(
        column_name='author',
        attribute='author',
        widget=ForeignKeyWidget(Author, 'name'))

    class Meta:
        fields = ('author',)
```

Parameters

- **`model`** – The Model the ForeignKey refers to (required).
- **`field`** – A field on the related model used for looking up a particular object.

`get_queryset`(*value*, *row*, **args*, ***kwargs*)
Returns a queryset of all objects for this Model.

Overwrite this method if you want to limit the pool of objects from which the related object is retrieved.

Parameters

- **`value`** – The field's value in the datasource.
- **`row`** – The datasource's current row.

As an example; if you'd like to have ForeignKeyWidget look up a Person by their pre- **and** lastname column, you could subclass the widget like so:

```
class FullNameForeignKeyWidget(ForeignKeyWidget):
    def get_queryset(self, value, row):
        return self.model.objects.filter(
            first_name__iexact=row["first_name"],
```

```
            last_name__iexact=row["last_name"]
        )
```

class import_export.widgets.**ManyToManyWidget**(*model*, *separator=None*, *field=None*, **args*, ***kwargs*)

Widget that converts between representations of a ManyToMany relationships as a list and an actual ManyToMany field.

Parameters

- **model** – The model the ManyToMany field refers to (required).
- **separator** – Defaults to `','`.
- **field** – A field on the related model. Default is `pk`.

CHAPTER 8

Fields

class import_export.fields.**Field**(*attribute=None*, *column_name=None*, *widget=None*, *default=<class 'django.db.models.fields.NOT_PROVIDED'>*, *readonly=False*, *saves_null_values=True*)

Field represent mapping between *object* field and representation of this field.

Parameters

- **attribute** – A string of either an instance attribute or callable off the object.
- **column_name** – Lets you provide a name for the column that represents this field in the export.
- **widget** – Defines a widget that will be used to represent this field's data in the export.
- **readonly** – A Boolean which defines if this field will be ignored during import.
- **default** – This value will be returned by *clean()* if this field's widget did not return an adequate value.
- **saves_null_values** – Controls whether null values are saved on the object

clean(*data*)

Translates the value stored in the imported datasource to an appropriate Python object and returns it.

export(*obj*)

Returns value from the provided object converted to export representation.

get_value(*obj*)

Returns the value of the object's attribute.

save(*obj*, *data*)

If this field is not declared readonly, the object's attribute will be set to the value returned by *clean()*.

CHAPTER 9

Instance loaders

class import_export.instance_loaders.**BaseInstanceLoader**(*resource*, *dataset=None*)
Base abstract implementation of instance loader.

class import_export.instance_loaders.**ModelInstanceLoader**(*resource*, *dataset=None*)
Instance loader for Django model.

Lookup for model instance by import_id_fields.

class import_export.instance_loaders.**CachedInstanceLoader**(**args*, ***kwargs*)
Loads all possible model instances in dataset avoid hitting database for every get_instance call.

This instance loader work only when there is one import_id_fields field.

CHAPTER 10

Temporary storages

10.1 TempFolderStorage

class import_export.tmp_storages.**TempFolderStorage**(*name=None*)

10.2 CacheStorage

class import_export.tmp_storages.**CacheStorage**(*name=None*)

By default memcache maximum size per key is 1MB, be careful with large files.

10.3 MediaStorage

class import_export.tmp_storages.**MediaStorage**(*name=None*)

CHAPTER 11

Results

11.1 Result

class import_export.results.**Result**(**args*, ***kwargs*)

Python Module Index

i

Index

I

M

P

R

S

T

W