django-import-export Documentation

Release 2.8.0

Bojan Mihelac

Mar 31, 2022

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

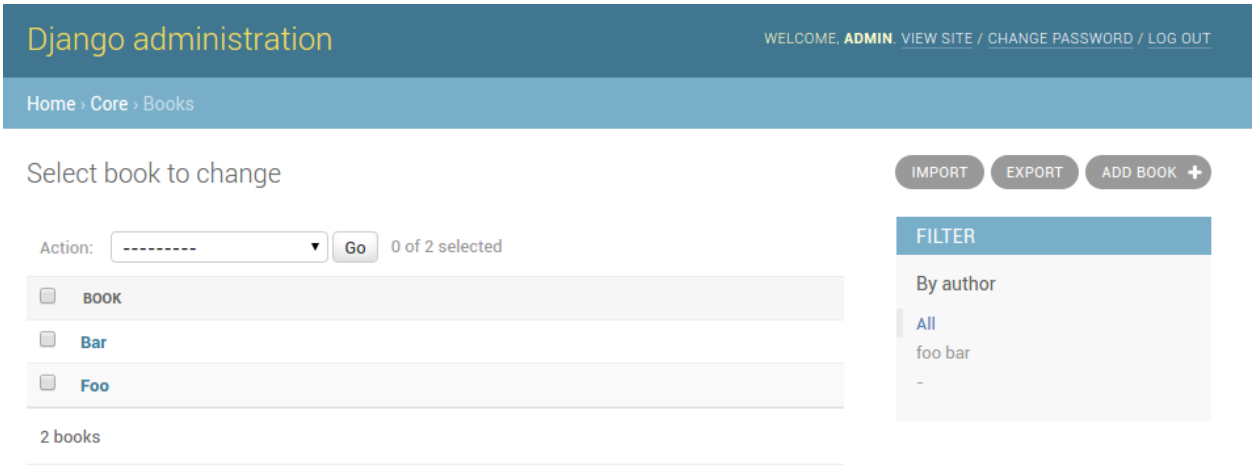

Fig. 1: A screenshot of the change view with Import and Export buttons.

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
```

This will automatically install many formats supported by tablib. If you need additional formats like `cli` or `Pandas DataFrame`, you should install the appropriate tablib dependencies (e.g. `pip install tablib[pandas]`). Read more on the tablib format documentation page.

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

1.1 Settings

You can configure the following in your settings file:

1.1.1 IMPORT_EXPORT_USE_TRANSACTIONS

Controls if resource importing should use database transactions. Defaults to `False`. Using transactions makes imports safer as a failure during import won't import only part of the data set.

Can be overridden on a `Resource` class by setting the `use_transactions` class attribute.

1.1.2 IMPORT_EXPORT_SKIP_ADMIN_LOG

If set to `True`, skips the creation of admin log entries when importing. Defaults to `False`. This can speed up importing large data sets, at the cost of losing an audit trail.

Can be overridden on a `ModelAdmin` class inheriting from `ImportMixin` by setting the `skip_admin_log` class attribute.

1.1.3 IMPORT_EXPORT_TMP_STORAGE_CLASS

Controls which storage class to use for storing the temporary uploaded file during imports. Defaults to `import_export.tmp_storages.TempFolderStorage`.

Can be overridden on a `ModelAdmin` class inheriting from `ImportMixin` by setting the `tmp_storage_class` class attribute.

1.1.4 IMPORT_EXPORT_IMPORT_PERMISSION_CODE

If set, lists the permission code that is required for users to perform the "import" action. Defaults to `None`, which means everybody can perform imports.

Django's built-in permissions have the codes `add`, `change`, `delete`, and `view`. You can also add your own permissions.

1.1.5 IMPORT_EXPORT_EXPORT_PERMISSION_CODE

If set, lists the permission code that is required for users to perform the "export" action. Defaults to `None`, which means everybody can perform exports.

Django's built-in permissions have the codes `add`, `change`, `delete`, and `view`. You can also add your own permissions.

1.1.6 IMPORT_EXPORT_CHUNK_SIZE

An integer that defines the size of chunks when iterating a QuerySet for data exports. Defaults to `100`. You may be able to save memory usage by decreasing it, or speed up exports by increasing it.

Can be overridden on a `Resource` class by setting the `chunk_size` class attribute.

1.2 Example app

There's an example application that showcases what django-import-export can do. It's assumed that you have set up a Python `venv` with all required dependencies (from `test.txt` requirements file) and are able to run Django locally.

You can run the example application as follows:

```
cd tests
./manage.py makemigrations
./manage.py migrate
./manage.py createsuperuser
./manage.py loaddata category.json book.json
./manage.py runserver
```

Go to http://127.0.0.1:8000

`books-sample.csv` contains sample book data which can be imported.

CHAPTER 2

Getting started

2.1 Test data

There are test data files which can be used for importing in the *test/core/exports* directory.

2.2 The test models

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 __str__(self):
        return self.name


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

    def __str__(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)
```

(continues on next page)

(continued from previous page)

```
    categories = models.ManyToManyField(Category, blank=True)

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

2.3 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
```

2.4 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
```

2.5 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

2.6 Declaring fields

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

```
from import_export.fields import Field

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

    class Meta:
        model = Book
```

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

```
from import_export.fields import Field

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

    class Meta:
        model = Book
```

See also:

Fields Available field types and options.

2.7 Advanced data manipulation on export

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 on export, `dehydrate_<fieldname>` method should be defined:

```
from import_export.fields import Field

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

    class Meta:
        model = Book

    def dehydrate_full_title(self, book):
        book_name = getattr(book, "name", "unknown")
        author_name = getattr(book.author, "name", "unknown")
        return '%s by %s' % (book_name, author_name)
```

In this case, the export looks like this:

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

2.8 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 overridden 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 = {
```

(continues on next page)

(continued from previous page)

```
                'published': {'format': '%d.%m.%Y'},
                }
```

See also:

Widgets available widget types and options.

2.9 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.

2.9.1 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
```

2.10 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
```

2.11 Admin integration

2.11.1 Exporting

Exporting via list filters

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

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

class BookAdmin(ImportExportModelAdmin):
    resource_class = BookResource

admin.site.register(Book, BookAdmin)
```

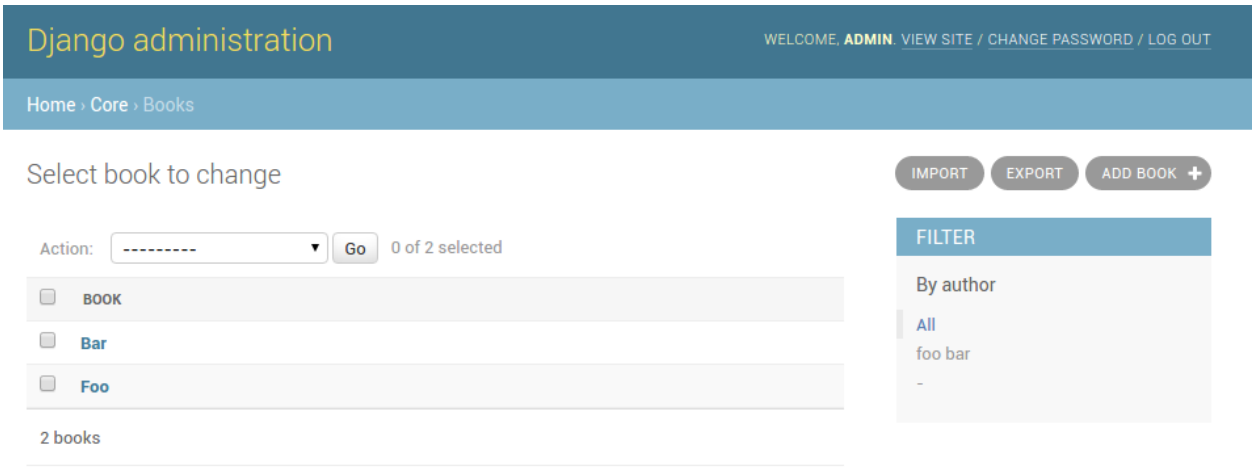

Fig. 1: A screenshot of the change view with Import and Export buttons.

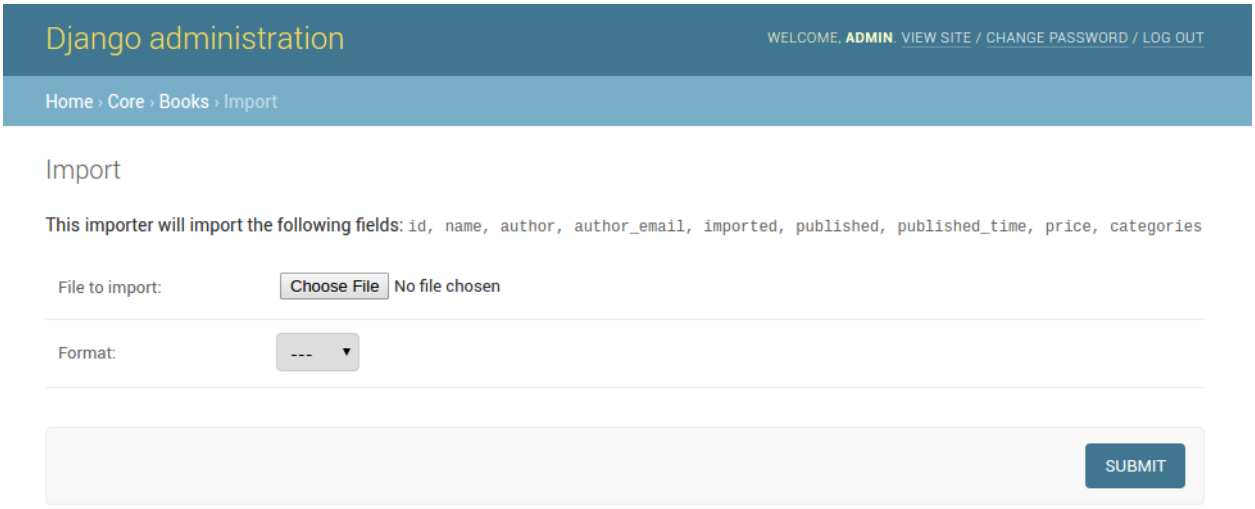

Fig. 2: A screenshot of the import view.

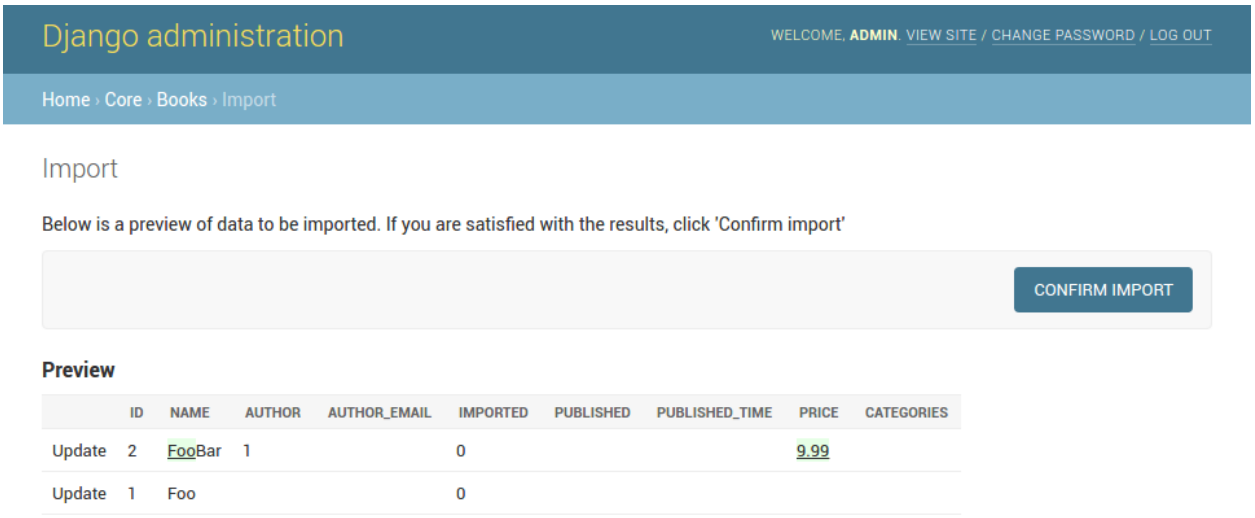

Fig. 3: A screenshot of the confirm import view.

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
```

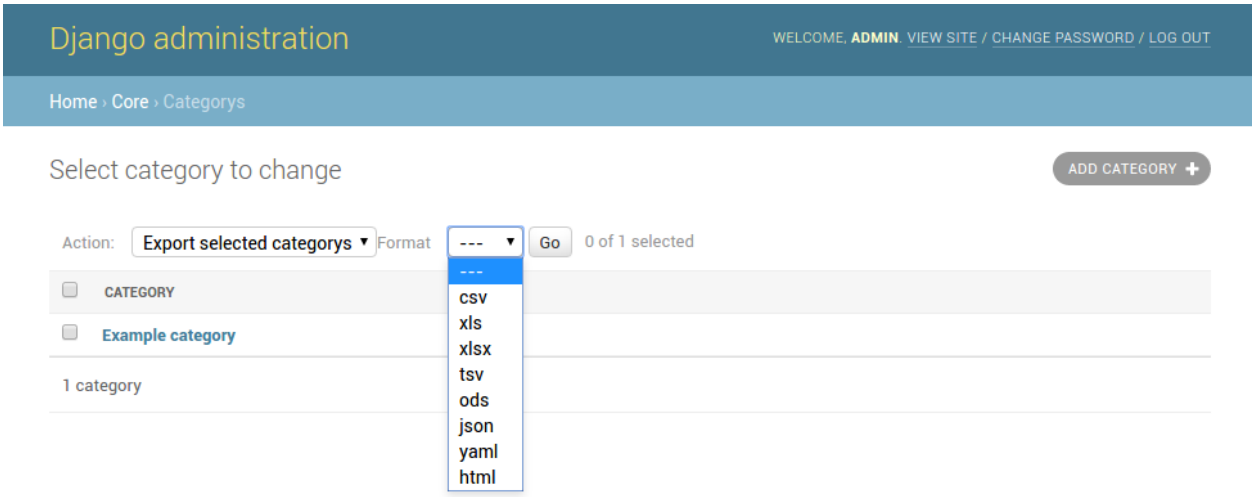

Fig. 4: A screenshot of the change view with Import and Export as an admin action.

Note that to use the *ExportMixin* or *ExportActionMixin*, you must declare this mixin **before** `admin.ModelAdmin`:

```
# app/admin.py
from django.contrib import admin
from import_export.admin import ExportActionMixin

class BookAdmin(ExportActionMixin, admin.ModelAdmin):
    pass
```

Note that *ExportActionMixin* is declared first in the example above!

2.11.2 Importing

It is also possible to enable data import via standard Django admin interface. To do this subclass *ImportExportModelAdmin* or use one of the available mixins, i.e. *ImportMixin*, or *ImportExportMixin*. Customizations are, of course, possible.

Customize admin import forms

It is possible to modify default import forms used in the model admin. For example, to add an additional field in the import form, subclass and extend the *ImportForm* (note that you may want to also consider *ConfirmImportForm* as importing is a two-step process).

To use the customized form(s), overload *ImportMixin* respective methods, i.e. *get_import_form()*, and also *get_confirm_import_form()* if need be.

For example, imagine you want to import books for a specific author. You can extend the import forms to include `author` field to select the author from.

Customize forms:

```
from django import forms

class CustomImportForm(ImportForm):
    author = forms.ModelChoiceField(
        queryset=Author.objects.all(),
        required=True)

class CustomConfirmImportForm(ConfirmImportForm):
    author = forms.ModelChoiceField(
        queryset=Author.objects.all(),
        required=True)
```

Customize `ModelAdmin`:

```
class CustomBookAdmin(ImportMixin, admin.ModelAdmin):
    resource_class = BookResource

    def get_import_form(self):
        return CustomImportForm

    def get_confirm_import_form(self):
        return CustomConfirmImportForm

    def get_form_kwargs(self, form, *args, **kwargs):
        # pass on `author` to the kwargs for the custom confirm form
        if isinstance(form, CustomImportForm):
            if form.is_valid():
                author = form.cleaned_data['author']
                kwargs.update({'author': author.id})
        return kwargs


admin.site.register(Book, CustomBookAdmin)
```

To further customize admin imports, consider modifying the following *ImportMixin* methods: *get_form_kwargs()*, `get_import_resource_kwargs()`, *get_import_data_kwargs()*.

Using the above methods it is possible to customize import form initialization as well as importing customizations.

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 initialized. 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:

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

      1. *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 customize how the new object is created (i.e. set default values).

2. *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.
3. 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.
4. 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`.
5. 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.
6. *save_m2m()* is called to save many to many fields.
7. `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*
8. 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

Bulk imports

django-import-export provides a 'bulk mode' to improve the performance of importing large datasets.

In normal operation, django-import-export will call `instance.save()` as each row in a dataset is processed. Bulk mode means that `instance.save()` is not called, and instances are instead added to temporary lists. Once the number of rows processed matches the `batch_size` value, then either `bulk_create()` or `bulk_update()` is called.

If `batch_size` is set to `None`, then `bulk_create()` / `bulk_update()` is only called once all rows have been processed.

Bulk deletes are also supported, by applying a `filter()` to the temporary object list, and calling `delete()` on the resulting query set.

4.1 Caveats

- The model's `save()` method will not be called, and `pre_save` and `post_save` signals will not be sent.
- `bulk_update()` is only supported in Django 2.2 upwards.
- Bulk operations do not work with many-to-many relationships.
- Take care to ensure that instances are validated before bulk operations are called. This means ensuring that resource fields are declared appropriately with the correct widgets. If an exception is raised by a bulk operation, then that batch will fail. It's also possible that transactions can be left in a corrupted state. Other batches may be successfully persisted, meaning that you may have a partially successful import.
- In bulk mode, exceptions are not linked to a row. Any exceptions raised by bulk operations are logged (and re-raised if `raise_errors` is true).
- If you use *`ForeignKeyWidget`* then this can affect performance, because it reads from the database for each row. If this is an issue then create a subclass which caches `get_queryset()` results rather than reading for each invocation.

For more information, please read the Django documentation on bulk_create() and bulk_update().

4.2 Performance tuning

Consider the following if you need to improve the performance of imports.

- Enable `use_bulk` for bulk create, update and delete operations (read *Caveats* first).
- If your import is creating instances only (i.e. you are sure there are no updates), then set `force_init_instance = True`.
- If your import is updating or creating instances, and you have a set of existing instances which can be stored in memory, use *`CachedInstanceLoader`*
- By default, import rows are compared with the persisted representation, and the difference is stored against each row result. If you don't need this diff, then disable it with `skip_diff = True`.
- Setting `batch_size` to a different value is possible, but tests showed that setting this to `None` always resulted in worse performance in both duration and peak memory.

CHAPTER 5

Using celery to perform imports

You can use the 3rd party django-import-export-celery application to process long imports in celery.

CHAPTER 6

Changelog

6.1 2.8.0 (2022-03-31)

- Updated import.css to support dark mode (#1318)
- Fix crash when import_data() called with empty Dataset and *collect_failed_rows=True* (#1381)
- Improve Korean translation (#1402)
- Update example subclass widget code (#1407)
- Drop support for python3.6, django 2.2, 3.0, 3.1 (#1408)
- Add get_export_form() to ExportMixin (#1409)

6.2 2.7.1 (2021-12-23)

- Removed *django_extensions* from example app settings (#1356)
- Added support for Django 4.0 (#1357)

6.3 2.7.0 (2021-12-07)

- Big integer support for Integer widget (#788)
- Run compilemessages command to keep .mo files in sync (#1299)
- Added *skip_html_diff* meta attribute (#1329)
- Added python3.10 to tox and CI environment list (#1336)
- Add ability to rollback the import on validation error (#1339)
- Fix missing migration on example app (#1346)

- Fix crash when deleting via admin site (#1347)
- Use Github secret in CI script instead of hard-coded password (#1348)
- Documentation: correct error in example application which leads to crash (#1353)

6.4 2.6.1 (2021-09-30)

- Revert 'dark mode' css: causes issues in django2.2 (#1330)

6.5 2.6.0 (2021-09-15)

- Added guard for null 'options' to fix crash (#1325)
- Updated import.css to support dark mode (#1323)
- Fixed regression where overridden mixin methods are not called (#1315)
- Fix xls/xlsx import of Time fields (#1314)
- Added support for 'to_encoding' attribute (#1311)
- Removed travis and replaced with github actions for CI (#1307)
- Increased test coverage (#1286)
- Fix minor date formatting issue for date with years < 1000 (#1285)
- Translate the zh_Hans missing part (#1279)
- Remove code duplication from mixins.py and admin.py (#1277)
- Fix example in BooleanWidget docs (#1276)
- Better support for Django main (#1272)
- don't test Django main branch with python36,37 (#1269)
- Support Django 3.2 (#1265)
- Correct typo in Readme (#1258)
- Rephrase logical clauses in docstrings (#1255)
- Support multiple databases (#1254)
- Update django master to django main (#1251)
- Add Farsi translated messages in the locale (#1249)
- Update Russian translations (#1244)
- Append export admin action using ModelAdmin.get_actions (#1241)
- Fix minor mistake in makemigrations command (#1233)
- Remove EOL Python 3.5 from CI (#1228)
- CachedInstanceLoader defaults to empty when import_id is missing (#1225)
- Add kwargs to import_row, import_object and import_field (#1190)
- Call load_workbook() with data_only flag (#1095)

6.6 2.5.0 (2020-12-30)

- Changed the default value for `IMPORT_EXPORT_CHUNK_SIZE` to 100. (#1196)
- Add translation for Korean (#1218)
- Update linting, CI, and docs.

6.7 2.4.0 (2020-10-05)

- Fix deprecated Django 3.1 `Signal(providing_args=...)` usage.
- Fix deprecated Django 3.1 `django.conf.urls.url()` usage.

6.8 2.3.0 (2020-07-12)

- Add missing translation keys for all languages (#1144)
- Added missing Portuguese translations (#1145)
- Add kazakh translations (#1161)
- Add bulk operations (#1149)

6.9 2.2.0 (2020-06-01)

- Deal with importing a BooleanField that actually has *True*, *False*, and *None* values. (#1071)
- Add row_number parameter to before_import_row, after_import_row and after_import_instance (#1040)
- Paginate queryset if Queryset.prefetch_related is used (#1050)

6.10 2.1.0 (2020-05-02)

- Fix DurationWidget handling of zero value (#1117)
- Make import diff view only show headers for user visible fields (#1109)
- Make confirm_form accessible in get_import_resource_kwargs and get_import_data_kwargs (#994, #1108)
- Initialize Decimal with text value, fix #1035 (#1039)
- Adds meta flag 'skip_diff' to enable skipping of diff operations (#1045)
- Update docs (#1097, #1114, #1122, #969, #1083, #1093)

6.11 2.0.2 (2020-02-16)

- Add support for tablib >= 1.0 (#1061)
- Add ability to install a subset of tablib supported formats and save some automatic dependency installations (needs tablib >= 1.0)

- Use column_name when checking row for fields (#1056)

6.12 2.0.1 (2020-01-15)

- Fix deprecated Django 3.0 function usage (#1054)
- Pin tablib version to not use new major version (#1063)
- Format field is always shown on Django 2.2 (#1007)

6.13 2.0 (2019-12-03)

- Removed support for Django < 2.0
- Removed support for Python < 3.5
- feat: Support for Postgres JSONb Field (#904)

6.14 1.2.0 (2019-01-10)

- feat: Better surfacing of validation errors in UI / optional model instance validation (#852)
- chore: Use modern setuptools in setup.py (#862)
- chore: Update URLs to use https:// (#863)
- chore: remove outdated workarounds
- chore: Run SQLite tests with in-memory database
- fix: Change logging level (#832)
- fix: Changed *get_instance()* return val (#842)

6.15 1.1.0 (2018-10-02)

- fix: Django2.1 ImportExportModelAdmin export (#797) (#819)
- setup: add django2.1 to test matrix
- JSONWidget for jsonb fields (#803)
- Add ExportActionMixin (#809)
- Add Import Export Permissioning #608 (#804)
- write_to_tmp_storage() for import_action() (#781)
- follow relationships on ForeignKeyWidget #798
- Update all pypi.python.org URLs to pypi.org
- added test for tsv import
- added unicode support for TSV for python 2
- Added ExportViewMixin (#692)

6.16 1.0.1 (2018-05-17)

- Make deep copy of fileds from class attr to instance attr (#550)
- Fix #612: NumberWidget.is_empty() should strip the value if string type (#613)
- Fix #713: last day isn't included in results qs (#779)
- use Python3 compatible MySql driver in development (#706)
- fix: warning U mode is deprecated in python 3 (#776)
- refactor: easier overridding widgets and default field (#769)
- Updated documentation regardign declaring fields (#735)
- custom js for action form also handles grappelli (#719)
- Use 'verbose_name' in breadcrumbs to match Django default (#732)
- Add Resource.get_diff_class() (#745)
- Fix and add polish translation (#747)
- Restore raise_errors to before_import (#749)

6.17 1.0.0 (2018-02-13)

- Switch to semver versioning (#687)
- Require Django>=1.8 (#685)
- upgrade tox configuration (#737)

6.18 0.7.0 (2018-01-17)

- skip_row override example (#702)
- Testing against Django 2.0 should not fail (#709)
- Refactor transaction handling (#690)
- Resolves #703 fields shadowed (#703)
- discourage installation as a zipped egg (#548)
- Fixed middleware settings in test app for Django 2.x (#696)

6.19 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)

6.20 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)

6.21 0.5.1 (2016-09-29)

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

6.22 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)

6.23 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, ‘’]

6.24 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

6.25 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)

6.26 0.4.2 (2015-12-18)

- add xlsx import support

6.27 0.4.1 (2015-12-11)

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

6.28 0.4.0 (2015-12-02)

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

6.29 0.3.1 (2015-11-20)

- FIX: importing csv in python 3

6.30 0.3 (2015-11-20)

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

6.31 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)

6.32 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)

6.33 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

6.34 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

6.35 0.2.5 (2014-10-04)

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

6.36 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

6.37 0.2.3 (2014-07-01)

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

6.38 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

6.39 0.2.1 (2014-02-20)

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

6.40 0.2.0 (2014-01-30)

- Python 3 support

6.41 0.1.6 (2014-01-21)

- Additional hooks for customizing the workflow (#61)

6.42 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)

6.43 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

6.44 0.1.3

- Fix packaging
- DB transactions support for importing data

6.45 0.1.2

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

6.46 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 available formats in Admin integration
- bugfixes

6.47 0.1.0

- Refactor api

CHAPTER 7

Admin

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

class import_export.admin.**ExportActionMixin**(**args*, ***kwargs*)
Mixin with export functionality implemented as an admin action.

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

get_actions(*request*)
Adds the export action to the list of available actions.

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

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

This is intended to be mixed with django.contrib.admin.ModelAdmin https://docs.djangoproject.com/en/dev/ref/contrib/admin/

change_list_template = 'admin/import_export/change_list_export.html'
template for change_list view

export_template_name = 'admin/import_export/export.html'
template for export view

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

get_export_form()
Get the form type used to read the export format.

get_export_queryset(*request*)
Returns export queryset.

Default implementation respects applied search and filters.

has_export_permission(*request*)
Returns whether a request has export permission.

to_encoding = None
export data encoding

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.

change_list_template = 'admin/import_export/change_list_import_export.html'
template for change_list view

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

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

This is intended to be mixed with django.contrib.admin.ModelAdmin https://docs.djangoproject.com/en/dev/ref/contrib/admin/

change_list_template = 'admin/import_export/change_list_import.html'
template for change_list view

from_encoding = 'utf-8'
import data encoding

get_confirm_import_form()
Get the form type (class) used to confirm the import.

get_form_kwargs(*form*, **args*, ***kwargs*)
Prepare/returns kwargs for the import form.

To distinguish between import and confirm import forms, the following approach may be used:

if isinstance(form, ImportForm): # your code here for the import form kwargs # e.g. update.kwargs({...})

elif isinstance(form, ConfirmImportForm): # your code here for the confirm import form kwargs # e.g. update.kwargs({...})

...

get_import_data_kwargs(*request*, **args*, ***kwargs*)
Prepare kwargs for import_data.

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

has_import_permission(*request*)
Returns whether a request has import permission.

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.

import_template_name = 'admin/import_export/import.html'
template for import view

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

CHAPTER 8

Resources

8.1 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*, *row_number=None*, ***kwargs*)
Override to add additional logic. Does nothing by default.

after_import_row(*row*, *row_result*, *row_number=None*, ***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*, *row_number=None*, ***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.

bulk_create(*using_transactions*, *dry_run*, *raise_errors*, *batch_size=None*)
Creates objects by calling `bulk_create`.

bulk_delete(*using_transactions*, *dry_run*, *raise_errors*)
Deletes objects by filtering on a list of instances to be deleted, then calling `delete()` on the entire queryset.

bulk_update(*using_transactions*, *dry_run*, *raise_errors*, *batch_size=None*)
Updates objects by calling `bulk_update`.

delete_instance(*instance*, *using_transactions=True*, *dry_run=False*)
Calls `instance.delete()` as long as `dry_run` is not set. If `use_bulk` then instances are appended to a list for bulk import.

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_bulk_update_fields()
Returns the fields to be included in calls to bulk_update(). `import_id_fields` are removed because *id* fields cannot be supplied to bulk_update().

classmethod get_diff_class()
Returns the class used to display the diff for an imported instance.

get_diff_headers()
Diff representation headers.

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

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

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

get_import_id_fields()

get_instance(*instance_loader*, *row*)
If all 'import_id_fields' are present in the dataset, calls the *InstanceLoader*. Otherwise, returns *None*.

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*, *rollback_on_validation_errors=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.
- **rollback_on_validation_errors** – If both `use_transactions` and `rollback_on_validation_errors` are set to `True`, the import process will be rolled back in case of ValidationError.
- **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*, *is_m2m=False*, ***kwargs*)
Calls *import_export.fields.Field.save()* if `Field.attribute` is specified, and `Field.column_name` is found in `data`.

import_obj(*obj*, *data*, *dry_run*, ***kwargs*)
Traverses every field in this Resource and calls *import_field()*. If `import_field()` results in a `ValueError` being raised for one of more fields, those errors are captured and reraised as a single, multi-field ValidationError.

import_row(*row*, *instance_loader*, *using_transactions=True*, *dry_run=False*, *raise_errors=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.

init_instance(*row=None*)
Initializes an object. Implemented in *import_export.resources.ModelResource.init_instance()*.

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

Objects can be created in bulk if `use_bulk` is enabled.

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 and skip_diff == False.

If skip_diff is True, then no comparisons can be made because `original` will be None.

When left unspecified, skip_diff and skip_unchanged both default to `False`, and rows are never skipped.

Override this method to handle skipping rows meeting certain conditions.

Use `super` if you want to preserve default handling while overriding

```
class YourResource(ModelResource):
    def skip_row(self, instance, original):
        # Add code here
        return super(YourResource, self).skip_row(instance, original)
```

validate_instance(*instance*, *import_validation_errors=None*, *validate_unique=True*)
Takes any validation errors that were raised by *import_obj()*, and combines them with validation errors raised by the instance's `full_clean()` method. The combined errors are then re-raised as single, multi-field ValidationError.

If the `clean_model_instances` option is False, the instances's `full_clean()` method is not called, and only the errors raised by `import_obj()` are re-raised.

8.2 ModelResource

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

DEFAULT_RESOURCE_FIELD
alias of *import_export.fields.Field*

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.

classmethod get_fk_widget(*field*)
Prepare widget for fk and o2o fields

classmethod get_m2m_widget(*field*)
Prepare widget for m2m field

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.

Includes mapping of Postgres Array and JSON fields. In the case that psycopg2 is not installed, we consume the error and process the field regardless.

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

8.3 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:

batch_size = 1000
The batch_size parameter controls how many objects are created in a single query. The default is to create objects in batches of 1000. See bulk_create(). This parameter is only used if `use_bulk` is True.

chunk_size = None
Controls the chunk_size argument of Queryset.iterator or, if prefetch_related is used, the per_page attribute of Paginator.

clean_model_instances = False
Controls whether `instance.full_clean()` is called during the import process to identify potential validation errors for each (non skipped) row. The default value is False.

exclude = None
Controls what introspected fields the Resource should NOT include. A blacklist of fields.

export_order = None
Controls export order for columns.

fields = None
Controls what introspected fields the Resource should include. A whitelist of fields.

force_init_instance = False
If True, this parameter will prevent imports from checking the database for existing instances. Enabling this parameter is a performance enhancement if your import dataset is guaranteed to contain new instances.

import_id_fields = ['id']
Controls which object fields will be used to identify existing instances.

instance_loader_class = None
Controls which class instance will take care of loading existing objects.

model = None
Django Model class. It is used to introspect available fields.

report_skipped = True
Controls if the result reports skipped rows. Default value is True

skip_diff = False
Controls whether or not an instance should be diffed following import. By default, an instance is copied prior to insert, update or delete. After each row is processed, the instance's copy is diffed against the original, and the value stored in each `RowResult`. If diffing is not required, then disabling the diff operation by setting this value to `True` improves performance, because the copy and comparison operations are skipped for each row. If enabled, then `skip_row()` checks do not execute, because 'skip' logic requires comparison between the stored and imported versions of a row. If enabled, then HTML row reports are also not generated (see `skip_html_diff`). The default value is False.

skip_html_diff = False
Controls whether or not a HTML report is generated after each row. By default, the difference between a stored copy and an imported instance is generated in HTML form and stored in each `RowResult`. The HTML report is used to present changes on the confirmation screen in the admin site, hence when this value is `True`, then changes will not be presented on the confirmation screen. If the HTML report is not required, then setting this value to `True` improves performance, because the HTML generation is skipped for each row. This is a useful optimization when importing large datasets. The default value is False.

skip_unchanged = False
Controls if the import should skip unchanged records. Default value is False

use_bulk = False
Controls whether import operations should be performed in bulk. By default, an object's save() method is called for each row in a data set. When bulk is enabled, objects are saved using bulk operations.

use_transactions = None
Controls if import should use database transactions. Default value is `None` meaning `settings.IMPORT_EXPORT_USE_TRANSACTIONS` will be evaluated.

using_db = None
DB Connection name to use for db transactions. If not provided, `router.db_for_write(model)` will be evaluated and if it's missing, DEFAULT_DB_ALIAS constant ("default") is used.

widgets = None
This dictionary defines widget kwargs for fields.

8.4 modelresource_factory

`resources.`**`modelresource_factory`**(*resource_class=<class 'import_export.resources.ModelResource'>*)
Factory for creating `ModelResource` class for given Django model.

CHAPTER 9

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.

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.

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

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.

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

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.**BooleanWidget**
Widget for converting boolean fields.

The widget assumes that `True`, `False`, and `None` are all valid values, as to match Django's BooleanField. That said, whether the database/Django will actually accept NULL values will depend on if you have set `null=True` on that Django field.

While the BooleanWidget is set up to accept as input common variations of "True" and "False" (and "None"), you may need to munge less common values to `True`/`False`/`None`. Probably the easiest way to do this is to override the *before_import_row()* function of your Resource class. A short example:

```
from import_export import fields, resources, widgets

class BooleanExample(resources.ModelResource):
    warn = fields.Field(widget=widgets.BooleanWidget())

    def before_import_row(self, row, row_number=None, **kwargs):
        if "warn" in row.keys():
            # munge "warn" to "True"
            if row["warn"] in ["warn", "WARN"]:
                row["warn"] = True

        return super().before_import_row(row, row_number, **kwargs)
```

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*)
On export, `True` is represented as `1`, `False` as `0`, and `None`/NULL as a empty string.

Note that these values are also used on the import confirmation view.

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

Takes optional `format` parameter.

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

Takes optional `format` parameter.

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.**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.

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.**DurationWidget**
Widget for converting time duration fields.

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.**JSONWidget**
Widget for a JSON object (especially required for jsonb fields in PostgreSQL database.)

Parameters **value** – Defaults to JSON format.

The widget covers two cases: Proper JSON string with double quotes, else it tries to use single quotes and then convert it to proper JSON.

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

```
from import_export import fields, resources
from import_export.widgets import ForeignKeyWidget

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.

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.

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, *args, **kwargs):
        return self.model.objects.filter(
            first_name__iexact=row["first_name"],
            last_name__iexact=row["last_name"]
        )
```

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.**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`.

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.

CHAPTER 10

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*, ***kwargs*)

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*, *is_m2m=False*, ***kwargs*)

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

CHAPTER 11

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 12

Temporary storages

12.1 TempFolderStorage

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

12.2 CacheStorage

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

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

12.3 MediaStorage

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

CHAPTER 13

Results

13.1 Result

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

has_errors()
Returns a boolean indicating whether the import process resulted in any critical (non-validation) errors for this result.

has_validation_errors()
Returns a boolean indicating whether the import process resulted in any validation errors for this result.

CHAPTER 14

Forms

class import_export.forms.**ImportForm**(*import_formats*, **args*, ***kwargs*)

class import_export.forms.**ConfirmImportForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*, *renderer=None*)

Python Module Index

i

Index

D

E

F

G

H

I

J

M

P

R

S

T

U

V

W