Django-data-exports Documentation

Release 0.8dev

Mathieu Agopian

Jul 06, 2017

Contents

- Author: Mathieu Agopian and contributors
- Licence: BSD
- Compatibility: Python 2.6, Python 2.7, Python 3.3, Python 3.4, Django 1.3+ (class-based-views required)
- Requirements: django-inspect-model
- Project URL: https://github.com/magopian/django-data-exports/
- Documentation: http://django-data-exports.readthedocs.org/en/latest/

Django-data-exports is a model data exports app for Django. It allows you to easily create exports for your models.

Adding this app to your project will let you create exports for your models, and customize the data that will be exported by specifying which columns to include, and which format to use.

Typical use case: display a few columns from one of your models as a HTML table to be easily copy/pasted to a spreadsheet.

CHAPTER 1

Installation

```
pip install django-data-exports
```

Then add to your project's `INSTALLED_APPS`. In `settings.py`:

```
INSTALLED_APPS = (
    '...',
    # whatever you already have
    '...',
    'data_exports',
)
```

Install the models:

```
./manage.py syncdb  # or ./manage.py migrate if you're using south
```

And finally, plug the urls to your `ROOT_URLCONF`:

```
urlpatterns = patterns(
    '',
    # ... all the other urls you already have

    # exports
    url(r'^exports/', include('data_exports.urls', namespace='data_exports')),
)
```

CHAPTER 2

Usage

Either add exports through the admin, or use the included example views. If there's no export format attached to an export, the `data_exports/export_detail.html` template will be rendered with the following context:

- `export`: the export itself
- `data`: a queryset of all the `export.model`'s instance

Using the admin

There's nothing specific to do here: connect to the admin, and add new exports. A few things to note:

- when you create an export, it's not possible to add columns at first. The reason being that the model is needed to be able to populate the column names
- when you add an export, clicking on the "save" button will have the same effect as clicking on "save and continue editing"
- once an export is created, and is being edited, the columns can be added (and are displayed as inlines)

Using the included example views

There's three included example views:

- `/exports/add`: create a new export
- `/exports/<export slug>/columns`: add columns to your export
- `/exports/<export slug>`: visualize your export

There is, at the moment, no example view for the export formats.

Export columns

Column choices make use of django-inspect-model to build the list of accessible "items". Please check this app's documentation to know more about "items".

Choices are built by `data_exports.forms.get_choices`, and will consist of all the accessible items on the exported model, and on all its related models. The only related fields accessible are those on models that are directly related, using forward or reverse OneToOne fields and forward ForeignKey fields.

Example:

```
class Foo(models.Model):
    name = CharField(max_length=50)
    bar = ForeignKey(Bar)

class Bar(models.Model):
    name = CharField(max_length=50)
```

An export of `Foo` will have the following column choices:

- `name`: Foo.name
- `bar`: Foo.bar, which is unicode(Foo.bar)
- `bar.name`: Bar.name

To display the value of those columns, the included templates use `data_exports.templatetags.getter_tags`:

Getattribute filter

```
{% load getter_tags %}
{{ obj|getattribute:column }}
```

This is roughly equivalent to the `getattr` python builtin, but can cope with column choices:

- if `column` doesn't have a dot, return `getattr(obj, column)`, or `getattr(obj, column)()` if it's a callable
- if `column` does have a dots (eg: `bar.name`), recursively call `getattribute()` to get to the final attribute:

```
attr = getattribute(obj, 'bar.name')
# equivalent to:
temp = getattr(obj, 'bar')
attr = getattr(temp, 'name')
```

Nice_display filter

```
{% load getter_tags %}
{{ obj|getattribute:column|nice_display }}
```

For now, all this does is return a comma-separated list of related instances for a many-to-many field.

If the `item` field has an `all` method:

```
return ', '.join(map(unicode, item.all()))
```

CHAPTER 3

Advanced usage

Export formats

Exports can export to a given format:

```
class Format(models.Model):
    name = models.CharField(max_length=50)
    file_ext = models.CharField( max_length=10, blank=True)
    mime = models.CharField(max_length=50)
    template = models.CharField(max_length=255)
```

The `mime` field is the `Content-Type` needed for the response. `file_ext` will be used to compute the export's filename, provided via `Content-Disposition` header.

Example: let's take a naive export to csv:

- mime: text/csv
- file_ext: csv
- name: Naive CSV format
- template: `data_exports/export_detail_csv.html` (included as an example)

If an export uses this format, visiting the export's view page `/exports/<export slug>` will offer a file download, named `<export slug>.csv`.

Filtering exports

To restrict entries access, you can use a class method or a static method `export_queryset` which will get the request object and returns the queryset of items to display.

```
from django.contrib.auth.models import User
from django.db import models
```

```
class Client(models.Model):
    name = models.CharField(max_length=63)
    users = models.ManyToManyField(User)


class ClientData(models.Model):
    client = models.ForeignKey('Client')
    address = models.CharField(max_length=255)
    money_hidden_in_the_garden = models.IntegerField()

    @classmethod
    def export_queryset(cls, request):
        qs = cls.objects.all()
        if not request.user.is_superuser:
            qs = qs.filter(client__in=request.user.client_set.all())
        return qs
```

Using your own views

To use your own views, you need to use the same url names as in `data_exports/urls.py`, and make sure they use the `data_exports` namespace, as `django.core.urlresolvers.reverse` is used internally to compute the needed urls.

You can check the included example views in `data_exports/views.py`, and of course reuse the forms provided in `data_exports/forms.py`.

Decorating the included views

Say you need to decorate the export view with the `staff_member_required` decorator:

```
url(r'^export/(?P<slug>[^/]+)/?$',
    staff_member_required(export_view),
    name='export_view'),
```

You still need to include this new url using a namespace, or the calls to `reverse` in the views won't work. This is a way to do it (taken from the Django documentation:

```
from django.conf.urls import include, patterns, url

data_exports_patterns = patterns('',
    url(r'^export/(?P<slug>[^/]+)/?$',
        staff_member_required(export_view),
        name='export_view'),
)

url(r'^exports', include(data_exports_patterns, namespace='data_exports')),
```

Using your own templates

`Django-data-exports` makes use of Django's template overloading mechanism. This means that if you provide

a `data_exports/export_detail.html` template which has precedence over the one bundled with the app, it'll be used.

Example: say you have a `templates/` folder in your project, and the appropriate `TEMPLATE_DIRS` setting. Place your own template in `project/templates/data_exports/export_detail.html` to have it used instead of the template bundled with the app in `data_exports/templates/data_exports/export_detail.html`.

There's three included templates:

- `data_exports/base.html`: extended by the two other templates
- `data_exports/export_detail.html`: used by default for exports that don't specify a format
- `data_exports/export_detail_csv.html`: used by the "naive csv format" detailed in *Export formats*.

CHAPTER 4

Hacking

Setup your environment:

```
git clone https://github.com/magopian/django-data-exports.git
cd django-data-export
```

Hack and run the tests using Tox to test on all the supported python and Django versions:

```
make test
```

To build the docs:

```
make docs
```

CHAPTER 5

Changelog

0.8 (unreleased)

0.7 (2014-09-15)

- increased the length of *Column.label* to 255 chars (fixes #9)
- now compatible with Django1.7
- perf improvements, queryset filtering (fixes #7), thanks @Christophe31

0.6 (2013-11-12)

- fixes #5: compatibility with django 1.6

0.5 (2013-10-14)

- fixes #3: Export's slug is now unique
- fixes #4: fix exports failing for models pointed to with a OneToOneField

0.4 (2013-07-28)

- compatible python 3.3

Older versions

- 0.3: compatible with django 1.4, download link in the admin
- 0.2: fields on related models also available to exports
- 0.1: initial version

CHAPTER 6

Indices and tables

- genindex
- modindex
- search