Bricks Documentation

Release 1.0

Germano Guerrini

January 27, 2015

Bricks is a Django application that helps you to sort heterogeneous models applying a set of criteria.

Suppose for example that your site publishes news and videos and you need to show them on a single page mixed together and sorted by one or more criteria.

Depending on how complex your models and your criteria are, it can get pretty tricky.

The *Basic Usage* will guide you through a complete tutorial to explain the concepts behind Bricks and to show how simple can be to achieve that goal.

CHAPTER 1

Requirements

Python 2	>= 2.6
Python 3	>= 3.2
Django	>= 1.5

CHAPTER 2

Contents

2.1 Getting Started

2.1.1 Installation

Use your favorite Python package manager to install the app from PyPI, e.g.

Example:

```
pip install djangobricks
```

2.1.2 Configuration

Add `djangobricks` to the `INSTALLED_APPS` within your settings file (usually `settings.py`).

Example:

```
INSTALLED_APPS = [
    [...]
    'djangobricks',
]
```

2.2 Basic Usage

2.2.1 Use Case Scenario

To demonstrate how Bricks works, let's pretend we run a website that publishes both news and videos on its homepage, and that it has two tabs to let the user choose between two different sorting order: descending publication date (newest first) and descending number of comments (most commented first).

So let's write some *very basic* Django models that we are going to use as a base for out example:

```
from django.db import models

class News(models.Model):
    title = models.CharField()
    text = models.TextField()
    pub_date = models.DateTimeField()
```

```
class Video(models.Model):
    title = models.CharField()
    video = models.FileField()
    pub_date = models.DateTimeField()


class Thread(models.Model):
    """A model that can be generically associated to a news or a video."""
    content_type = models.ForeignKey(ContentType)
    object_id = models.PositiveIntegerField()
    content_object = generic.GenericForeignKey('content_type', 'object_id')
    comments_count = models.PositiveIntegerField(default=0) # Denormalization


class Comment(models.Model):
    thread = models.ForeignKey(Thread, related_name='comments')
    text = models.TextField()
    # Guess other fields
```

Bricks exposes three main concepts:

Brick

A brick is a container for an instance of a Django model or even a (presumably small) list of instances.

Its a brick responsibility to *wrap* a queryset and returns a list of wrapped objects, to retrieve a value from its content to be used as a sorting key and finally to render its content.

Bricks offers two classes: `SingleBrick` and `ListBrick`.

Normally, subclasses will only need to override the `template_name` attribute and, in case of a `ListBrick` subclass, the `get_bricks_for_queryset` class method.

Criterion

A criterion is a proxy for a value of the brick.

If the brick contains a single instance, then it's usually a property of the instance itself, otherwise it's a property of the list.

You can also specify a callable instead of a model property or a default value if, for example, the value has no meaning for a given model.

It's important to note that a criterion has no information about the actual sorting order, so you have to pass that info along using the `SORTING_ASC` and the `SORTING_DESC` constant.

Be sure to check the `Criterion` class reference.

Wall

Not surprisingly, a wall is a list of bricks. Mixed with a set of criteria, it sorts the bricks and can be iterated to get them back.

2.2.2 Setting up a wall

To start, we should create the criteria. They are subclasses of `Criterion`:

```
import datetime

from djangobricks.models import Criterion

CRITERION_PUB_DATE = Criterion('pub_date', default=datetime.datetime.now)
CRITERION_COMMENT_COUNT = Criterion('thread__comment_count', default=0)
```

Next, we are going to subclass `SingleBrick` to create a container for our objects. In this case, we can probably get away with a single subclass, but for the sake of completeness let's create a brick for a each model:

```
from djangobricks.models import SingleBrick

class NewsBrick(SingleBrick):
    template_name = 'bricks/single/news.html'

class VideoBrick(SingleBrick):
    template_name = 'bricks/single/video.html'
```

There is also a `ListBrick` class, but let's stick with a simple case for now.

At this point we can create our wall by hand, but let's use the `BaseWallFactory` class instead.

```
from myapp.models import News, Video

from djangobricks.models import BaseWallFactory

class HomepageWallFactory(BaseWallFactory):
    def get_content(self):
        return (
            (NewsBrick, News.objects.all()),
            (VideoBrick, Video.objects.all())
        )
```

The `BaseWallFactory.get_content` method returns an iterable of tuples, where the first element is a `BaseBrick` subclass and the second the queryset whose elements should be rendered using that class.

We are almost there! All we have to do is to create our wall in the view:

```
from djangobricks.models import SORTING_DESC

def index(request):
    last_content_criteria = (
        (CRITERION_PUB_DATE, SORTING_DESC),
    )
    last_content_wall = HomepageWallFactory(last_content_criteria)

    most_commented_criteria = (
        (CRITERION_COMMENT_COUNT, SORTING_DESC),
    )
    most_commented_content_wall = HomepageWallFactory(most_commented_criteria)

    context = {
        'last_content_wall': last_content_wall,
        'most_commented_content_wall: most_commented_content_wall
    }
    return render_to_response('index.html', context,
                              context_instance=RequestContext(request))
```

2.2.3 Render a Wall

Now that we have not one but two walls, we can render them within a Django template:

```
{% load bricks %}

{% for brick in last_content_wall %}
    {% render_brick brick %}
{% endfor%}

{% for brick in most_commented_content_wall %}
    {% render_brick brick %}
{% endfor%}
```

Done!

We covered the basic of Bricks, but it can handle much more complex scenarios. Be sure to check the *Advanced Usage*.

2.3 Advanced Usage

2.3.1 Using a ListBrick

Now that you have run through the *basics*, let's tackle some more advanced topic.

Building up from our previous example, let's say that our designers team decided that the list of videos and news must change. Specifically, the news have a new layout and they have to be grouped in list of five elements each.

In this case, we are going to use the `ListBrick` class.

So let's replace our declaration of `NewsBricks`:

```
from djangobricks.models import ListBrick

class NewsBrick(ListBrick):
    template_name = 'bricks/list/news.html
```

The `VideoBrick` doesn't need to be changed.

By default `get_bricks_for_queryset` returns a list of bricks containing 5 elements each. Unless you need some more complicated behaviour, you can simply change the number of elements by setting the `chunk_size` attribute accordingly.

Now, as the brick does not contain a single element, is not clear what a `Criterion` should return when applied to it. The value of the first element? The average? That is really up to you.

In this case, let's say that we want to change `CRITERION_PUB_DATE` to return the max value of the list (that is, the brick will be sorted based on the newest news it contains) and `CRITERION_COMMENT_COUNT` to return the average number of comments.

To do that, we simply change the criteria declaration by adding a callback function that accepts a list and return a value:

```
CRITERION_PUB_DATE = Criterion('pub_date', max, default=datetime.datetime.now)
CRITERION_COMMENT_COUNT = Criterion('thread__comment_count',
                                    lambda x:sum(x)/len(x), default=0)
```

We don't need to change anything else.

2.3.2 Filtering a wall

Our designers are relentless. They want the user to be able to filter some content from the wall. In our simple example, we can say that they want to hide the videos from the wall.

In this case, we can probably just build a second wall and returns it in our view depending on user choice, but we are going to do it the Bricks way.

The `BaseWall` class provides a simple `filter` method that accepts a list of callables that accepts a brick instance and returns a boolean, and a boolean operator. Each brick is then filtered against the list of callables: all of them if the operator is `AND` (the default value) or any of them if the operator is `OR`.

In our case, we need a single callback that should return `True` if the brick contains some news (remember: we want to hide the videos!) and `False` otherwise.

Something like this would to the trick:

```
def is_news_brick(brick):
    return brick.__class__.__name__ == 'NewsBrick'
```

And in our view:

```
...
filtered_last_content_wall = last_content_wall.filter(is_news_brick)
...
```

The advantage of this approach is speed. The creation of a wall can be an expensive operation. Caching a wall and filtering the cached result can be faster then building a new wall from scratch, especially if you have a more complicated setup with a lot of filters.

2.3.3 Handling heterogeneous models

Now let's say that we need to add another model to our wall, defined below:

```
from django.db import models

class PhotoGallery(models.Model):
    title = models.CharField()
    images = models.ManyToManyField(Photo)
    public_from_date = models.DateTimeField()
```

As you can see, this model doesn't have a `pub_date` field like `News` and `Video`. How can we use the `CRITERION_PUB_DATE` over this model?

Remember that is up to the brick to return a value for a given criterion of its content. So let's write a brick class for our new model:

```
from djangobricks.models import SingleBrick

class PhotoGalleryBrick(SingleBrick):
    template_name = 'bricks/single/photo_gallery.html'
    def get_value_for_criterion(self, criterion):
        if criterion.attrname == 'pub_date':
            return self.item.public_from_date
        return super(PhotoGalleryBrick, self).get_value_for_criterion(criterion)
```

And that's it! Unless you are sure to cover each possible criterion, it's a good practice to return the value from super at least, as shown above.

2.3.4 Adding context to the template

By default, `SingleBrick` will pass the object to the context with an `object` key.

The `ListBrick` context contains an `object_list` key instead.

If you want to add extra context to render the template, you can either override the `get_context` method as show below:

```
class NewsBrick(SingleBrick):

    def get_context(self, **kwargs):
        context = super(NewsBrick, self).get_context(**kwargs)
        context['color'] = 'red'
        return context
```

or you can add them using directly the templatetag

```
{% render_brick brick color='red' %}
```

2.4 API Reference

2.4.1 Classes

2.4.2 Utilities

2.5 Changelog

2.5.1 Version 1.1

- Added support for Python >= 3.2
- Minor documentation fixes

2.5.2 Version 1.0

- First version

CHAPTER 3

Indices and tables

- *genindex*
- *search*