Django Generic Ratings Documentation

Release 0.6

Francesco Banconi

August 12, 2016

Contents

This application provides rating functionality to a Django project.

You can handle scores and number of votes for each content type without adding additional fields to your models.

Different vote types can be associated to a single content object, and you can write rules and business logic in a customized rating handler describing how a model instance can be voted.

This app provides *jQuery* based widgets, useful for increasing the voting experience of users (e.g.: slider rating, star rating).

The source code for this app is hosted on https://bitbucket.org/frankban/django-generic-ratings

Contents:

CHAPTER 1

Getting started

1.1 Requirements

Python	>= 2.5
Django	>= 1.0

jQuery >= 1.4 is required if you want to take advantage of *AJAX* voting, or if you want to use customized rating methods like slider rating or star rating. This application, out of the box, provides widgets for these kind of rating user interfaces (see Forms and widgets reference).

1.2 Installation

The Mercurial repository of the application can be cloned with this command:

```
hg clone https://frankban@bitbucket.org/frankban/django-generic-ratings
```

The `ratings` package, included in the distribution, should be placed on the `PYTHONPATH`.

Otherwise you can just `pip install django-generic-ratings`.

1.3 Settings

Add the request context processor in your *settings.py*, e.g.:

```
from django.conf.global_settings import TEMPLATE_CONTEXT_PROCESSORS
TEMPLATE_CONTEXT_PROCESSORS += (
    'django.core.context_processors.request',
)
```

Add `'ratings'` to the `INSTALLED_APPS` in your *settings.py*.

See Customization section in this documentation for other settings options. However, in settings you can define global application options valid for all handled models (i. e. models whose instances can be voted), but it is easy to customize rating options for each handled models (see Using handlers).

Add the ratings urls to your *urls.py*, e.g.:

```
(r'^ratings/', include('ratings.urls')),
```

Time to create the needed database tables using *syncdb* management command:

```
./manage.py syncdb
```

1.4 Quickstart

First, you have to tell to the system that your model can be voted and that its instances have a rating.

For instance, having a *Film* model:

```
from ratings.handlers import ratings
ratings.register(Film)
```

The *Film* model is now handled, and, by default, if you didn't customize things in your settings file, only authenticated users can vote films using 1-5 ranged scores (without decimal places). See Using handlers for an explanation of how to change rating options and how to define a custom rating handler.

Now it's time to let your users vote a film, e.g.:

```
{% load ratings_tags %}

{% get_rating_form for film as rating_form %}

<form action="{% url ratings_vote %}" method="post">
    {% csrf_token %}
    {{ rating_form }}
    <p><input type="submit" value="Vote &rarr;"></p>
</form>
```

And why not to display current score for our film?

```
{% load ratings_tags %}

{% get_rating_score for film as score %}

{% if score %}
    Average score: {{ score.average|floatformat }}
    Number of votes: {{ score.num_votes }}
{% else %}
    How sad: nobody voted {{ film }}
{% endif %}
```

This application provides templatetags to get a vote by a given user, to annotate a queryset with scores and votes, to get the latest votes given to an object or by a user, and so on: see Templatetags reference for a detailed explanation of provided templatetags.

Anyway, you may want to take a look at Using handlers first.

CHAPTER 2

Using handlers

As seen in Getting started, a model instance can be voted and can have an associated score only if its model class is handled. Being handled, for a model, means it is registered with an handler.

We have seen how to do that:

```
from ratings.handlers import ratings
ratings.register(Film)
```

The handler class is an optional argument of the `ratings.register` method, and, if not provided, the default `ratings.handlers.RatingHandler` handler is used

The previous code can be written:

```
from ratings.handlers import ratings, RatingHandler
ratings.register(Film, RatingHandler)
```

For convenience, `ratings.register` can also accept a list of model classes in place of a single model; this allows easier registration of multiple models with the same handler class, e.g.:

```
from ratings.handlers import ratings, RatingHandler
ratings.register([Film, Series], RatingHandler)
```

Where should this code live? You can register handlers anywhere you like. However, you'll need to make sure that the module it's in gets imported early on so that the model gets registered before any voting is performed or rating is requested. This makes your app's *models.py* a good place to put the above code.

The default rating handler provides only one 1-5 ranged (without decimal places) score for each content object, and allows voting only for authenticated users. It also allows user to delete and change their vote.

We can, however, override some options while registering the model. For instance, if we want 1-10 ranged votes with a step of 0.5 (half votes), and we don't want users to delete their votes, we can give these options as *kwargs*:

```
from ratings.handlers import ratings, RatingHandler
ratings.register(Film, RatingHandler,
    score_range=(1, 10), score_step=0.5, can_delete_vote=False)
```

The handler manages the voting form too, and, by default the widget used to render the score is a simple text input. If you want to use the more cool star rating widget, you can do:

```
from ratings.handlers import ratings
from ratings.forms import StarVoteForm
ratings.register(Film, form_class=StarVoteForm)
```

For a list of all available built-in options, see Handlers reference.

However, there are situations where the built-in options are not sufficient.

What if, for instance, you want only active objects to be voted for a given model? As in Django own `contrib.admin.ModelAdmin`, you can write subclasses of `RatingHandler` to override the methods which actually perform the voting process, and apply any logic they desire.

Here is an example meeting the staff users needs:

```
from ratings.handlers import ratings, RatingHandler

class MyHandler(RatingHandler):
   def allow_vote(self, request, instance, key):
       allowed = super(MyHandler, self).allow_vote(request, instance, key)
       return allowed and instance.is_active

ratings.register(Film, MyHandler)
```

In the above example, the `allow_vote` method is called before any voting attempt, and takes the current *request*, the *instance* being voted and a *key*.

The *key* is a string representing the type of rating we are giving to an object. For example, the same film can be associated with multiple types of rating (e.g. a score for the photography, one for the direction, one for the music, and so on): a user can vote the music or the direction, so the *key* can be used to distinguish music from direction. In fact, the *key* can even be the string `'music'` or the string `'direction'`.

The default *key* is `'main'`. Don't worry: we will talk more about rating keys in Usage and examples.

2.1 Handlers API

Handlers are not only used to manage and customize the voting process, but also grant a simplified access to the underneath Django models api.

First, we have to obtain the handler instance associated with our model:

```
from ratings.handlers import ratings
handler = ratings.get_handler(Film)
```

The method `ratings.get_handler` returns None if model is not registered, and can take a model instance too:

```
from ratings.handlers import ratings
film = Film.objects.latest()
handler = ratings.get_handler(film)
```

What we can do with the handler? For instance, we can get the `'main'` score or our *film*:

```
score = handler.get_score(film, 'main')

if score:
    print 'Average score:', score.average
    print 'Number of votes:', score.num_votes
    print 'Total score:', score.total
else:
    print u'Nobody voted %s' % film
```

Or we can check if current user has voted our *film*:

```
voted = handler.has_voted(film, 'main', request.user)
```

See Handlers reference for a detailed explanation of other utility methods of handlers, and of `ratings.handlers.ratings` registry too. And in Models reference you will find the lower level Django model's API.

It could be clear now that the rating handler is a layer of abstraction above Django models and forms, and handlers are used by templatetags and views too. This way, building our own handlers means we can customize the behaviour of the entire application.

Before going to see the Handlers reference, maybe it is better to take a look at some Usage and examples.

CHAPTER 3

Usage and examples

As seen previously in Using handlers, we can customize the voting process creating and registering rating handlers.

In this section we will deal with some real-world examples of usage of Django Generic Ratings.

3.1 Simple rating

We want votes in range 1-10 (including extremes) and we want to use a slider widget to let the user vote.

The model ragistration is straightforward:

```
from ratings.handlers import ratings
from ratings.forms import SliderVoteForm
ratings.register(Film, score_range=(1, 10), form_class=SliderVoteForm)
```

The template where we want users to vote requires very little code:

```
{# javascripts required by SliderVoteForm #}
<script src="path/to/jquery.js" type="text/javascript"></script>
<script src="path/to/jquery-ui.js" type="text/javascript"></script>

{% load ratings_tags %}

{% get_rating_form for film as rating_form %}
<form action="{% url ratings_vote %}" method="post">
    {% csrf_token %}
    {{ rating_form }}
    <p><input type="submit" value="Vote &rarr;"></p>
</form>

{% get_rating_score for film as score %}
{% if score %}
    Average score: {{ score.average|floatformat }}
    Number of votes: {{ score.num_votes }}
{% else %}
    How sad: nobody voted {{ film }}
{% endif %}
```

Done. See Forms and widgets reference for a description of all available forms and widgets.

3.2 Multiple ratings

We want users to vote (in range 1-10) both the film and the trailer of the film (I know: this is odd).

We have to customize the handler in order to make it deal with two different rating keys (that we call `'film'` and `'trailer'`):

```
from ratings.handlers import ratings, RatingHandler

class MyHandler(RatingHandler):
    score_range = (1, 10)

    def allow_key(self, request, instance, key):
        return key in ('film', 'trailer')

ratings.register(Film, MyHandler)
```

This way we are saying to the handler to allow those new keys.

The template is very similar to the one seen in simple rating, but we must specify the rating key when using template-tags:

```
{% load ratings_tags %}

Vote the film:
{# note the 'using' argument below #}
{% get_rating_form for film using 'film' as film_rating_form %}
<form action="{% url ratings_vote %}" method="post">
    {% csrf_token %}
    {{ film_rating_form }}
    <p><input type="submit" value="Vote Film &rarr;"></p>
</form>

Vote the trailer:
{% get_rating_form for film using 'trailer' as trailer_rating_form %}
<form action="{% url ratings_vote %}" method="post">
    {% csrf_token %}
    {{ trailer_rating_form }}
    <p><input type="submit" value="Vote Trailer &rarr;"></p>
</form>

{# note the 'using' argument below #}
{% get_rating_score for film using 'film' as film_score %}
{% if film_score %}
    Average film score: {{ film_score.average|floatformat }}
    Number of votes: {{ film_score.num_votes }}
{% else %}
    How sad: nobody voted {{ film }}
{% endif %}

{% get_rating_score for film using 'trailer' as trailer_score %}
{% if trailer_score %}
    Average trailer score: {{ trailer_score.average|floatformat }}
    Number of votes: {{ trailer_score.num_votes }}
{% else %}
    How sad: nobody voted {{ film }}'s trailer
{% endif %}
```

That's all: of course you can assign more than 2 rating keys to each model.

3.3 Conditional ratings

We want users to star rate our film, using five stars with a step of half star. This time we want two different ratings: the first, we call it `'expectation'`, is used when the film is not yet released, while the second one, we call it `real` is used after the film release. Again, this is odd too, but at least this is something I actually had to implement.

So, we want the rating system to use two different rating keys based on the release status of the voted object:

```
import datetime
from ratings.handlers import ratings, RatingHandler

class MyHandler(RatingHandler):
    score_range = (1, 5)
    score_step = 0.5

    def get_key(self, request, instance):
        today = datetime.date.today()
        return 'expectation' if instance.release_date < today else 'real'
```

The template looks like this (here we assume the film has an `is_released` self explanatory method):

```
{# javascripts and css required by StarVoteForm #}
<script src="path/to/jquery.js" type="text/javascript"></script>
<script src="path/to/jquery-ui.js" type="text/javascript"></script>
<link href="/path/to/jquery.rating.css" rel="stylesheet" type="text/css" />
<script type="text/javascript" src="/path/to/jquery.MetaData.js"></script>
<script type="text/javascript" src="/path/to/jquery.rating.js"></script>

{% load ratings_tags %}

{# do not specify the key -> the key is obtained using our handler #}
{% get_rating_form for film as rating_form %}
<form action="{% url ratings_vote %}" method="post">
    {% csrf_token %}
    {{ rating_form }}
    <p><input type="submit" value="Vote &rarr;"></p>
</form>

{% if film.is_released %}

    {% get_rating_score for film using 'real' as real_score %}
    {% if real_score %}
        Average score: {{ real_score.average|floatformat }}
        Number of votes: {{ real_score.num_votes }}
    {% else %}
        How sad: nobody voted {{ film }}
    {% endif %}

{% else %}

    {% get_rating_score for film using 'expectation' as expected_score %}
    {% if expected_score %}
        Average expectation: {{ expected_score.average|floatformat }}
        Number of votes: {{ expected_score.num_votes }}
    {% else %}
        Good: nobody expected something!
    {% endif %}
```

```
{% endif %}
```

Note that while the `allow_key` method (from previous example) is used to validate the key submitted by the form, the `get_key` one is used only if the key is not specified as a templatetag argument.

Actually, the default implementation of `allow_key` only checks if the given key matches the key returned by `get_key`.

3.4 Like/Dislike rating

We want users to rate *+1* or *-1* our film. Actually this application does not provide a widget for like/dislike rating, and it's up to you creating one. But the business logic is straightforward:

```
from somewhere import LikeForm
from ratings.handlers import ratings

ratings.register(Film, score_range=(-1, 1), form_class=LikeForm)
```

In the template we can show the current film rating using the total sum of all votes, e.g.:

```
{% load ratings_tags %}

{% get_rating_score for film as score %}
{% if score %}
    Film score: {% if score.total > 0 %}+{% endif %}{{ score.total }}
    Number of votes: {{ score.num_votes }}
{% else %}
    How sad: nobody voted {{ film }}
{% endif %}
```

3.5 Working with querysets

Consider the following code, printing all votes given by current user:

```
from ratings.models import Vote
for vote in Vote.objects.filter(user=request.user):
    print "%s -> %s" % (vote.content_object, vote.score)
```

There is nothing wrong in the above code snippet, except that it does, for each vote, a query to retrieve the voted object. You can avoid this using the `filter_with_contents` method of the *Vote* and *Score* models, e.g.:

```
from ratings.models import Vote
for vote in Vote.objects.filter_with_contents(user=request.user):
    print "%s -> %s" % (vote.content_object, vote.score)
```

This way only a query for each different content type is performed. We have shortcuts for votes retreival: for example the previous code can be rewritten like this:

```
from ratings.handlers import ratings
for vote in ratings.get_votes_by(request.user):
    print "%s -> %s" % (vote.content_object, vote.score)
```

The application also provides handler's shortcuts to get votes associated to a particular content type:

```
from ratings.handlers import ratings
handler = ratings.get_handler(MyModel)

# get all votes by user (regarding MyModel instances)
user_votes = handler.get_votes_by(request.user)

# get all votes given to myinstance
instance_votes = handler.get_votes_for(myinstance)
```

What if instead you have a queryset and you want to print the *main* score of each object in it? Of course you can write something like this:

```
from ratings.handlers import ratings

queryset = Film.objects.all()
handler = ratings.get_handler(queryset.model)
key = 'main'

for instance in queryset:
    score = handler.get_score(instance, key)
    print 'film:', instance
    print 'average score:', score.average
    print 'votes:', score.num_votes
```

Again, this is correct but you are doing a query for each object in the queryset. The ratings handler lets you annotate the *queryset* with scores using a given *key*, e.g.:

```
from ratings.handlers import ratings

queryset = Film.objects.all()
handler = ratings.get_handler(queryset.model)
key = 'main'

queryset_with_scores = handler.annotate_scores(queryset, key,
    myaverage='average', num_votes='num_votes')

for instance in queryset_with_scores:
    print 'film:', instance
    print 'average score:', instance.myaverage
    print 'votes:', instance.num_votes
```

As seen, each film in queryset has two new attached fields: *myaverage* and *num_votes*. The same kind of annotation can be done with user's votes, see Handlers reference.

3.6 Using AJAX

This application comes with out-of-the-box *AJAX* voting support.

All is needed is the inclusion of the provided `ratings.js` javascript in the template where the vote form is displayed. The javascript file is present in the `static/ratings/js/` directory of the distribution.

The script will handle the AJAX vote submit for all forms having *ratings* class.

Here is a working example of an *AJAX* voting form that uses the slider widget:

```
{# javascripts and css required by SliderVoteForm #}
<script src="path/to/jquery.js" type="text/javascript"></script>
<script src="path/to/jquery-ui.js" type="text/javascript"></script>
```

```
script type="text/javascript" src="/path/to/ratings.js"></script>

{% load ratings_tags %}

{% get_rating_form for object as rating_form %}

<form action="{% url ratings_vote %}" class="ratings" method="post">
    {% csrf_token %}
    {{ rating_form }}
    <p>
        {# only authenticated users can vote #}
        {% if user.is_authenticated %}
            <input type="submit" value="Vote"></p>
        {% else %}
            <a href="{% url login %}?next={{ request.path }}">Vote</a>
        {% endif %}
    </p>
    <span class="success" style="display: none;">Vote registered!</span>
    <span class="error" style="display: none;">Errors...</span>
</form>
```

By default, if you did not customize the handler, the *AJAX* request (on form submit) returns a *JSON* response containing:

```
{
    'key': 'the_rating_key',
    'vote_id': vote.id,
    'vote_score': vote.score,
    'score_average': score.average,
    'score_num_votes': score.num_votes,
    'score_total': score.total,
}
```

In the previous example, we put two hidden elements inside the form, the former having class *success* and the latter having class *error*. Each one, if present, is showed whenever an AJAX vote is successfully completed or not.

Further more, various javascript events are triggered during *AJAX* votes: see Forms and widgets reference for details.

3.7 Performance and database denormalization

One goal of *Django Generic Ratings* is to provide a generic solution to rate model instances without the need to edit your (or third party) models.

Sometimes, however, you may want to denormalize ratings data, for example because you need to speed up *order by* queries for tables with a lot of records, or for backward compatibility with legacy code.

Assume you want to store the average score and the number of votes in your film instances, and you want these values to change each time a user votes a film.

This is easily achievable, again, customizing the handler, e.g.:

```
from ratings.handlers import RatingHandler, ratings

class FilmRatingHandler(RatingHandler):

    def post_vote(self, request, vote, created):
        instance = vote.content_object
        score = vote.get_score()
```

```
        instance.average_vote = score.average
        instance.num_votes = score.num_votes
        instance.save()

ratings.register(Film, FilmRatingHandler)
```

3.8 Deleting model instances

When you delete a model instance all related votes and scores are contextually deleted too.

CHAPTER 4

Customization

When you register an handler you can customize all the ratings options, as seen in Using handlers.

But it is also possible to register an handler without overriding options or methods, and that handler will work using pre-defined global settings.

This section describes the settings used to globally customize ratings handlers, together with their default values.

`GENERIC_RATINGS_ALLOW_ANONYMOUS = False`

Set to False to allow votes only by authenticated users.

`GENERIC_RATINGS_SCORE_RANGE = (1, 5)`

A sequence of minimum and maximum values allowed in scores.

`GENERIC_RATINGS_SCORE_STEP = 1`

Step allowed in scores.

`GENERIC_RATINGS_WEIGHT = 0`

The weight used to calculate average score.

`GENERIC_RATINGS_DEFAULT_KEY = 'main'`

Default key to use for votes when there is only one vote-per-content.

`GENERIC_RATINGS_NEXT_QUERYSTRING_KEY = 'next'`

Querystring key that can contain the url of the redirection performed after voting.

`GENERIC_RATINGS_VOTES_PER_IP_ADDRESS = 0`

In case of anonymous users it is possible to limit votes per ip address (0 = no limits).

```
GENERIC_RATINGS_COOKIE_NAME_PATTERN = 'grvote_%(model)s_%(object_id)s_%(key)s'
```

The pattern used to create a cookie name.

```
GENERIC_RATINGS_COOKIE_MAX_AGE = 60 * 60 * 24 * 365 # one year
```

The cookie max age (number of seconds) for anonymous votes.

CHAPTER 5

Django signals

ratings.signals.**vote_will_be_saved**
Providing args: *vote*, *request*

Fired before a vote is saved.

Receivers can stop the vote process returning *False*.

One receiver is always called: *handler.pre_vote*

---

ratings.signals.**vote_was_saved**
Providing args: *vote*, *request*, *created*

Fired after a vote is saved.

One receiver is always called: *handler.post_vote*.

---

ratings.signals.**vote_will_be_deleted**
Providing args: *vote*, *request*

Fired before a vote is deleted.

Receivers can stop the vote deletion process returning *False*.

One receiver is always called: *handler.pre_delete*

---

ratings.signals.**vote_was_deleted**
Providing args: *vote*, *request*

Fired after a vote is deleted.

One receiver is always called: *handler.post_delete*

CHAPTER 6

Templatetags reference

In order to use the following templatetags you must `{% load ratings_tags %}` in your template.

6.1 get_rating_form

Return (as a template variable in the context) a form object that can be used in the template to add, change or delete a vote for the specified target object. Usage:

```
{% get_rating_form for *target object* [using *key*] as *var name* %}
```

Example:

```
```

{% get_rating_form for object as rating_form %} # key here is 'main' {% get_rating_form for target_object using 'mykey' as rating_form %}

The key can also be passed as a template variable (without quotes).

If you do not specify the key, then the key is taken using the registered handler for the model of given *object*.

Having the form object, it is quite easy to display the form, e.g.:

```
<form action="{% url ratings_vote %}" method="post">
    {% csrf_token %}
    {{ rating_form }}
    <p><input type="submit" value="Vote &rarr;"></p>
</form>
```

If the target object's model is not handled, then the template variable will not be present in the context.

6.2 get_rating_score

Return (as a template variable in the context) a score object representing the score given to the specified target object. Usage:

```
{% get_rating_score for *target object* [using *key*] as *var name* %}
```

Example:

```
{% get_rating_score for object as score %}
{% get_rating_score for target_object using 'mykey' as score %}
```

The key can also be passed as a template variable (without quotes).

If you do not specify the key, then the key is taken using the registered handler for the model of given *object*.

Having the score model instance you can display score info, as follows:

```
Average score: {{ score.average }}
Number of votes: {{ score.num_votes }}
```

If the target object's model is not handled, then the template variable will not be present in the context.

6.3 scores_annotate

Use this templatetag when you need to update a queryset in bulk adding score values, e.g:

```
{% scores_annotate queryset with myaverage='average' using 'main' %}
```

After this call each queryset instance has a *myaverage* attribute containing his average score for the key 'main'. The score field name and the key can also be passed as template variables, without quotes, e.g.:

```
{% scores_annotate queryset with myaverage=average_var using key_var %}
```

You can also specify a new context variable for the modified queryset, e.g.:

```
{% scores_annotate queryset with myaverage='average' using 'main' as new_queryset %}
{% for instance in new_queryset %}
    Average score: {{ instance.myaverage }}
{% endfor %}
```

You can annotate a queryset with different score values at the same time, remembering that accepted values are 'average', 'total' and 'num_votes':

```
{% scores_annotate queryset with myaverage='average',num_votes='num_votes' using 'main' %}
```

Finally, you can also sort the queryset, e.g.:

```
{% scores_annotate queryset with myaverage='average' using 'main' ordering by '-myaverage' %}
```

The order of arguments is important: the following example shows how to use this tempaltetag with all arguments:

```
{% scores_annotate queryset with myaverage='average',num_votes='num_votes' using 'main' ordering by
```

The following example shows how to display in the template the ten most rated films (and how is possible to order the queryset using multiple fields):

```
{% scores_annotate films with avg='average',num='num_votes' using 'user_votes' ordering by '-avg,-nur
{% for film in top_rated_films|slice:":10" %}
    Film: {{ film }}
    Average score: {{ film.avg }}
    ({{ film.num }} vote{{ film.num|pluralize }})
{% endfor %}
```

If the queryset's model is not handled, then this templatetag returns the original queryset.

6.4 get_rating_vote

Return (as a template variable in the context) a vote object representing the vote given to the specified target object by the specified user. Usage:

```
{% get_rating_vote for *target object* [by *user*] [using *key*] as *var name* %}
```

Example:

```
{% get_rating_vote for object as vote %}
{% get_rating_vote for target_object using 'mykey' as vote %}
{% get_rating_vote for target_object by myuser using 'mykey' as vote %}
```

The key can also be passed as a template variable (without quotes).

If you do not specify the key, then the key is taken using the registered handler for the model of given *object*.

If you do not specify the user, then the vote given by the user of current request will be returned. In this case, if user is anonymous and the rating handler allows anonymous votes, current cookies are used.

Having the vote model instance you can display vote info, as follows:

```
Vote: {{ vote.score }}
Ip Address: {{ vote.ip_address }}
```

If the target object's model is not handled, or the given user did not vote for that object, then the template variable will not be present in the context.

6.5 get_latest_votes_for

Return (as a template variable in the context) the latest vote objects given to a target object.

Usage:

```
{% get_latest_votes_for *target object* [using *key*] as *var name* %}
```

Usage example:

```
{% get_latest_votes_for object as latest_votes %}
{% get_latest_votes_for content.instance using 'main' as latest_votes %}
```

In the following example we display latest 10 votes given to an *object* using the 'by_staff' key:

```
{% get_latest_votes_for object uning 'mystaff' as latest_votes %}
{% for vote in latest_votes|slice:":10" %}
    Vote by {{ vote.user }}: {{ vote.score }}
{% endfor %}
```

The key can also be passed as a template variable (without quotes).

If you do not specify the key, then all the votes are taken regardless what key they have.

6.6 get_latest_votes_by

Return (as a template variable in the context) the latest vote objects given by a user.

Usage:

```
{% get_latest_votes_by *user* [using *key*] as *var name* %}
```

Usage example:

```
{% get_latest_votes_by user as latest_votes %}
{% get_latest_votes_for object.created_by using 'main' as latest_votes %}
```

In the following example we display latest 10 votes given by *user* using the 'by_staff' key:

```
{% get_latest_votes_by user uning 'mystaff' as latest_votes %}
{% for vote in latest_votes|slice:":10" %}
    Vote for {{ vote.content_object }}: {{ vote.score }}
{% endfor %}
```

The key can also be passed as a template variable (without quotes).

If you do not specify the key, then all the votes are taken regardless what key they have.

6.7 votes_annotate

Use this templatetag when you need to update a queryset in bulk adding vote values given by a particular user, e.g:

```
{% votes_annotate queryset with 'user_score' for myuser using 'main' %}
```

After this call each queryset instance has a *user_score* attribute containing the score given by *myuser* for the key 'main'. The score field name and the key can also be passed as template variables, without quotes, e.g.:

```
{% votes_annotate queryset with score_var for user using key_var %}
```

You can also specify a new context variable for the modified queryset, e.g.:

```
{% votes_annotate queryset with 'user_score' for user using 'main' as new_queryset %}
{% for instance in new_queryset %}
    User's score: {{ instance.user_score }}
{% endfor %}
```

Finally, you can also sort the queryset, e.g.:

```
{% votes_annotate queryset with 'myscore' for user using 'main' ordering by '-myscore' %}
```

The order of arguments is important: the following example shows how to use this tempaltetag with all arguments:

```
{% votes_annotate queryset with 'score' for user using 'main' ordering by 'score' as new_queryset %}
```

Note: it is not possible to annotate querysets with anonymous votes.

6.8 show_starrating

Show the starrating widget in read-only mode for the given *score_or_vote*. If *score_or_vote* is a score instance, then the average score is displayed.

Usage:

```
{# show star rating for the given vote #}
{% show_starrating vote %}

{# show star rating for the given score #}
{% show_starrating score %}

{# show star rating for the given score, using 10 stars with half votes #}
{% show_starrating score 10 2 %}
```

Normally the handler is used to get the number of stars and the how each one must be splitted, but you can override using *stars* and *split* arguments.

CHAPTER 7

Handlers reference

class ratings.handlers.**RatingHandler**

Encapsulates content rating options for a given model.

This class can be subclassed to specify different behaviour and options for ratings of a given model, but can also be used directly, just to handle default rating for any model.

The default rating provide only one 1-5 ranged (without decimal places) score for each content object, and allows voting only for authenticated users.

The default rating handler uses the project's settings as options: this way you can register not customized rating handlers and then modify their options just editing the settings file.

Most common rating needs can be handled by subclassing *RatingHandler* and changing the values of pre-defined attributes. The full range of built-in options is as follows.

allow_anonymous
set to False to allow votes only by authenticated users (default: *False*)

score_range
a sequence *(min_score, max_score) representing the allowed score range (including extremes) note that the score *0* is reserved for vote deletion* (default: *(1, 5)*)

score_step
the step allowed in scores (default: *1*)

weight
this is used while calculating the average score and represents the difficulty for a target object to obtain a higher rating (default: *0*)

default_key
default key to use for votes when there is only one vote-per-content (default: *'main'*)

can_delete_vote
set to False if you do not want to allow users to delete a previously saved vote (default: *True*)

can_change_vote
set to False if you do not want to allow users to change the score of a previously saved vote (default: *True*)

next_querystring_key
querystring key that can contain the url of the redirection performed after voting (default: *'next'*)

votes_per_ip_address
the number of allowed votes per ip address, only used if anonymous users can vote (default: *0*, means no limit)

form_class
form class that will be used to handle voting (default: *ratings.forms.VoteForm*) this app, out of the box, provides also *SliderVoteForm* and a *StarVoteForm*

cookie_max_age
if anonymous rating is allowed, you can define here the cookie max age as a number of seconds (default: one year)

success_messages
this should be a sequence of (vote created, vote changed, vote deleted) messages sent (using *django.contrib.messages*) to the user after a successful vote creation, change, deletion (scored without using AJAX) if this is None, then no message is sent (default: *None*)

For situations where the built-in options listed above are not sufficient, subclasses of *RatingHandler* can also override the methods which actually perform the voting process, and apply any logic they desire.

See the method's docstrings for a description of how each method is used during the voting process.

Methods you may want to override, but not to call directly

get_key (*self*, *request*, *instance*)
Return the ratings key to be used to save the vote if the key is not provided by the user (for example with the optional argument *using* in templatetags).

Subclasses can return different keys based on the *request* and the given target object *instance*.

For example, if you want a different key to be used if the user is staff, you can override this method in this way:

```
def get_key(self, request, instance):
    return 'staff' if request.user.is_superuser else 'normal'
```

This method is called only if the user does not provide a rating key.

allow_key (*self*, *request*, *instance*, *key*)
This method is called when the user tries to vote using the given rating *key* (e.g. when the voting view is called with POST data).

The voting process continues only if this method returns True (i.e. a valid key is passed).

For example, if you want to different rating for each target object, you can use two forms (each providing a different key, say 'main' and 'other') and then allow those keys:

```
def allow_key(self, request, instance, key):
    return key in ('main', 'other')
```

allow_vote (*self*, *request*, *instance*, *key*)
This method can block the voting process if the current user actually is not allowed to vote for the given *instance*

By default the only check made here is for anonymous users, but this method can be subclassed to implement more advanced validations by *key* or target object *instance*.

If you want users to vote only active objects, for instance, you can write inyour subclas:

```
def allow_vote(self, request, instance, key):
    allowed = super(MyClass, self).allow_vote(request, instance, key)
    return allowed and instance.is_active
```

If anonymous votes are allowed, this method checks for ip adresses too.

get_vote_form_class(*self*, *request*)
Return the vote form class that will be used to handle voting. This method can be overridden by view-level passed form class.

get_vote_form_kwargs(*self*, *request*, *instance*, *key*)
Return the optional kwargs used to instantiate the voting form.

pre_vote(*self*, *request*, *vote*)
Called just before the vote is saved to the db, this method takes the *request* and the unsaved *vote* instance.

The unsaved vote can be a brand new vote instance (without *id*) or an existing vote object the user want to change.

Subclasses can use this method to check if the vote can be saved and, if necessary, block the voting process returning False.

This method is called by a *signals.vote_will_be_saved* listener always attached to the handler. It's up to the developer if override this method or just connect another listener to the signal: the voting process is killed if just one receiver returns False.

vote(*self*, *request*, *vote*)
Save the vote to the database. Must return True if the *vote* was created, False otherwise.

By default this method just does *vote.save()* and recalculates the related score (average, total, number of votes).

post_vote(*self*, *request*, *vote*, *created*)
Called just after the vote is saved to the db.

This method is called by a *signals.vote_was_saved* listener always attached to the handler.

pre_delete(*self*, *request*, *vote*)
Called just before the vote is deleted from the db, this method takes the *request* and the *vote* instance.

Subclasses can use this method to check if the vote can be deleted and, if necessary, block the vote deletion process returning False.

This method is called by a *signals.vote_will_be_deleted* listener always attached to the handler. It's up to the developer if override this method or just connect another listener to the signal: the voting deletion process is killed if just one receiver returns False.

delete(*self*, *request*, *vote*)
Delete the vote from the database.

By default this method just do *vote.delete()* and recalculates the related score (average, total, number of votes).

post_delete(*self*, *request*, *vote*)
Called just after the vote is deleted to from db.

This method is called by a *signals.vote_was_deleted* listener always attached to the handler.

success_response(*self*, *request*, *vote*)
Callback used by the voting views, called when the user successfully voted. Must return a Django http response (usually a redirect, or some json if the request is ajax).

failure_response(*self*, *request*, *errors*)
Callback used by the voting views, called when vote form did not validate. Must return a Django http response.

Utility methods you may want to use in your python code

has_voted(*self*, *instance*, *key*, *user_or_cookies*)
Return True if the user related to given *user_or_cookies* has voted the given target object *instance* using the given *key*.

The argument *user_or_cookies* can be a Django User instance or a cookie dict (for anonymous votes).

A *ValueError* is raised if you give cookies but anonymous votes are not allowed by the handler.

get_vote(*self*, *instance*, *key*, *user_or_cookies*)
Return the vote instance created by the user related to given *user_or_cookies* for the target object *instance* using the given *key*.

The argument *user_or_cookies* can be a Django User instance or a cookie dict (for anonymous votes).

Return None if the vote does not exists.

A *ValueError* is raised if you give cookies but anonymous votes are not allowed by the handler.

get_votes_for(*self*, *instance*, ***kwargs*)
Return all votes given to *instance* and filtered by any given *kwargs*. All the content objects related to returned votes are evaluated together with votes.

get_votes_by(*self*, *user*, ***kwargs*)
Return all votes assigned by *user* to model instances handled by this handler, and filtered by any given *kwargs*. All the content objects related to returned votes are evaluated together with votes.

get_score(*self*, *instance*, *key*)
Return the score for the target object *instance* and the given *key*. Return None if the target object does not have a score.

annotate_scores(*self*, *queryset*, *key*, ***kwargs*)
Annotate the *queryset* with scores using the given *key* and *kwargs*.

In *kwargs* it is possible to specify the values to retreive mapped to field names (it is up to you to avoid name clashes). You can annotate the queryset with the number of votes (*num_votes*), the average score (*average*) and the total sum of all votes (*total*).

For example, the following call:

```
annotate_scores(Article.objects.all(), 'main',
    average='average', num_votes='num_votes')
```

Will return a queryset of article and each article will have two new attached fields *average* and *num_votes*.

Of course it is possible to sort the queryset by a score value, e.g.:

```
for article in annotate_scores(Article, 'by_staff',
    staff_avg='average', staff_num_votes='num_votes'
    ).order_by('-staff_avg', '-staff_num_votes'):
    print 'staff num votes:', article.staff_num_votes
    print 'staff average:', article.staff_avg
```

This is basically a wrapper around *ratings.model.annotate_scores*.

annotate_votes(*self*, *queryset*, *key*, *user*, *score='score'*)
Annotate the *queryset* with votes given by the passed *user* using the given *key*.

The score itself will be present in the attribute named *score* of each instance of the returned queryset.

Usage example:

```
for article in annotate_votes(Article.objects.all(), 'main', myuser,
    score='myscore'):
    print 'your vote:', article.myscore
```

This is basically a wrapper around *ratings.model.annotate_votes*. For anonymous voters this functionality is unavailable.

class ratings.handlers.**Ratings**

Registry that stores the handlers for each content type rating system.

An instance of this class will maintain a list of one or more models registered for being rated, and their associated handler classes.

To register a model, obtain an instance of *Ratings* (this module exports one as *ratings*), and call its *register* method, passing the model class and a handler class (which should be a subclass of *RatingHandler*). Note that both of these should be the actual classes, not instances of the classes.

To cease ratings handling for a model, call the *unregister* method, passing the model class.

For convenience, both *register* and *unregister* can also accept a list of model classes in place of a single model; this allows easier registration of multiple models with the same *RatingHandler* class.

register(*self*, *model_or_iterable*, *handler_class=None*, ***kwargs*)

Register a model or a list of models for ratings handling, using a particular *handler_class*, e.g.:

```
from ratings.handlers import ratings, RatingHandler
# register one model for rating
ratings.register(Article, RatingHandler)
# register other two models
ratings.register([Film, Series], RatingHandler)
```

If the handler class is not given, the default *ratings.handlers.RatingHandler* class will be used.

If *kwargs* are present, they are used to override the handler class attributes (using instance attributes), e.g.:

```
ratings.register(Article, RatingHandler,
    score_range=(1, 10), score_step=0.5)
```

Raise *AlreadyHandled* if any of the models are already registered.

unregister(*self*, *model_or_iterable*)

Remove a model or a list of models from the list of models that will be handled.

Raise *NotHandled* if any of the models are not currently registered.

get_handler(*self*, *model_or_instance*)

Return the handler for given model or model instance. Return None if model is not registered.

get_votes_by(*self*, *user*, ***kwargs*)

Return all votes assigned by *user* and filtered by any given *kwargs*. All the content objects related to returned votes are evaluated together with votes.

CHAPTER 8

Forms and widgets reference

The application provides some forms and widgets, useful for managing the vote action using different methods (e.g. star rating, slider rating).

Of course you can subclass the provided forms and widget in order to create your custom vote methods.

The base vote forms, if the handler allows it, are used for vote deletion too.

You can take a look at Usage and examples for an explanation of how to use AJAX with forms.

8.1 Forms

class `ratings.forms.`**`VoteForm`**(*forms.Form*)
Form class to handle voting of content objects.

You can customize the app giving a custom form class, following some rules:

- the form must define the *content_type* and *object_pk* fields
- the form's *__init__* method must take as first and second positional arguments the target object getting voted and the ratings key
- the form must define the *get_vote* method, getting the request and a boolean *allow_anonymous* and returning an unsaved instance of the vote model
- the form must define the *delete* method, getting the request and returning True if the form requests the deletion of the vote

`get_score_field`(*self*, *score_range*, *score_step*, *can_delete_vote*)
Return the score field. Subclasses may ovveride this method in order to change the field used to store score value.

`get_score_widget`(*self*, *score_range*, *score_step*, *can_delete_vote*)
Return the score widget. Subclasses may ovveride this method in order to change the widget used to display score input.

`get_vote`(*self*, *request*, *allow_anonymous*)
Return an unsaved vote object based on the information in this form. Assumes that the form is already validated and will throw a ValueError if not.

The vote can be a brand new vote or a changed vote. If the vote is just created then the instance's id will be None.

`get_vote_model`(*self*)
Return the vote model used to rate an object.

get_vote_data(*self*, *request*, *allow_anonymous*)
Return two dicts of data to be used to look for a vote and to create a vote.

Subclasses in custom ratings apps that override *get_vote_model* can override this method too to add extra fields into a custom vote model.

If the first dict is None, then the lookup is not performed.

delete(*self*, *request*)
Return True if the form requests to delete the vote.

class ratings.forms.**SliderVoteForm**(*VoteForm*)
Handle voting using a slider widget.

In order to use this form you must load the jQuery.ui slider javascript.

This form triggers the following javascript events:

- *slider_change* with the vote value as argument (fired when the user changes his vote)
- *slider_delete* without arguments (fired when the user deletes his vote)

It's easy to bind these events using jQuery, e.g.:

```
$(document).bind('slider_change', function(event, value) {
    alert('New vote: ' + value);
});
```

class ratings.forms.**StarVoteForm**(*VoteForm*)
Handle voting using a star widget.

In order to use this form you must download the jQuery Star Rating Plugin available at http://www.fyneworks.com/jquery/star-rating/#tab-Download and then load the required javascripts and css, e.g.:

```
<link href="/path/to/jquery.rating.css" rel="stylesheet" type="text/css" />
<script type="text/javascript" src="/path/to/jquery.MetaData.js"></script>
<script type="text/javascript" src="/path/to/jquery.rating.js"></script>
```

This form triggers the following javascript events:

- *star_change* with the vote value as argument (fired when the user changes his vote)
- *star_delete* without arguments (fired when the user deletes his vote)

It's easy to bind these events using jQuery, e.g.:

```
$(document).bind('star_change', function(event, value) {
    alert('New vote: ' + value);
});
```

8.2 Widgets

class ratings.forms.widgets.**SliderWidget**(*BaseWidget*)
Slider widget.

In order to use this widget you must load the jQuery.ui slider javascript.

This widget triggers the following javascript events:

- *slider_change* with the vote value as argument (fired when the user changes his vote)
- *slider_delete* without arguments (fired when the user deletes his vote)

It's easy to bind these events using jQuery, e.g.:

```
$(document).bind('slider_change', function(event, value) {
    alert('New vote: ' + value);
});
```

class ratings.forms.widgets.**StarWidget**(*BaseWidget*)

Starrating widget.

In order to use this widget you must download the jQuery Star Rating Plugin available at http://www.fyneworks.com/jquery/star-rating/#tab-Download and then load the required javascripts and css, e.g.:

```
<link href="/path/to/jquery.rating.css" rel="stylesheet" type="text/css" />
<script type="text/javascript" src="/path/to/jquery.MetaData.js"></script>
<script type="text/javascript" src="/path/to/jquery.rating.js"></script>
```

This widget triggers the following javascript events:

- *star_change* with the vote value as argument (fired when the user changes his vote)
- *star_delete* without arguments (fired when the user deletes his vote)

It's easy to bind these events using jQuery, e.g.:

```
$(document).bind('star_change', function(event, value) {
    alert('New vote: ' + value);
});
```

CHAPTER 9

Models reference

9.1 Base models

class ratings.models.**Score**(*models.Model*)
A score for a content object.

Fields: *content_type*, *object_id*, *content_object*, *key*, *average*, *total*, *num_votes*.

Manager: ratings.managers.RatingsManager

get_votes(*self*)
Return all the related votes (same *content_object* and *key*).

recalculate(*self*, *weight=0*, *commit=True*)
Recalculate the score using all the related votes, and updating average score, total score and number of votes.

The optional argument *weight* is used to calculate the average score: an higher value means a lot of votes are needed to increase the average score of the target object.

If the optional argument *commit* is False then the object is not saved.

class ratings.models.**Vote**(*models.Model*)
A single vote relating a content object.

Fields: *content_type*, *object_id*, *content_object*, *key*, *score*, *user*, *ip_address*, *cookie*, *created_at*, *modified_at*.

Manager: ratings.managers.RatingsManager

get_score(*self*)
Return the score related to current *content_object* and *key*. Return None if score does not exist.

by_anonymous(*self*)
Return True if this vote is given by an anonymous user.

9.2 Adding or changing scores and votes

ratings.models.**upsert_score**(*instance_or_content*, *key*, *weight=0*)
Update or create current score values (average score, total score and number of votes) for target object *instance_or_content* and the given *key*.

The argument *instance_or_content* can be a model instance or a sequence *(content_type, object_id)*.

You can use the optional argument *weight* to make more difficult for a target object to obtain a higher rating.

Return a sequence *score, created*.

9.3 Deleting scores and votes

ratings.models.**delete_scores_for**(*instance_or_content*)
Delete all score objects related to *instance_or_content*, that can be a model instance or a sequence *(content_type, object_id)*.

ratings.models.**delete_votes_for**(*instance_or_content*)
Delete all vote objects related to *instance_or_content*, that can be a model instance or a sequence *(content_type, object_id)*.

9.4 In bulk selections

ratings.models.**annotate_scores**(*queryset_or_model*, *key*, ***kwargs*)
Annotate *queryset_or_model* with scores, in order to retreive from the database all score values in bulk.

The first argument *queryset_or_model* must be, of course, a queryset or a Django model object. The argument *key* is the score key.

In *kwargs* it is possible to specify the values to retreive mapped to field names (it is up to you to avoid name clashes). You can annotate the queryset with the number of votes (*num_votes*), the average score (*average*) and the total sum of all votes (*total*).

For example, the following call:

```
annotate_scores(Article.objects.all(), 'main',
    average='average', num_votes='num_votes')
```

Will return a queryset of article and each article will have two new attached fields *average* and *num_votes*.

Of course it is possible to sort the queryset by a score value, e.g.:

```
for article in annotate_scores(Article, 'by_staff',
    staff_avg='average', staff_num_votes='num_votes'
    ).order_by('-staff_avg', '-staff_num_votes'):
    print 'staff num votes:', article.staff_num_votes
    print 'staff average:', article.staff_avg
```

ratings.models.**annotate_votes**(*queryset_or_model*, *key*, *user*, *score='score'*)
Annotate *queryset_or_model* with votes, in order to retreive from the database all vote values in bulk.

The first argument *queryset_or_model* must be, of course, a queryset or a Django model object. The argument *key* is the score key.

The votes are filtered using given *user*. For anonymous voters this functionality is unavailable.

The score itself will be present in the attribute named *score* of each instance of the returned queryset.

Usage example:

```
for article in annotate_votes(Article.objects.all(), 'main', myuser,
    score='myscore'):
    print 'your vote:', article.myscore
```

9.5 Abstract models

class ratings.models.**RatedModel**(*models.Model*)
Mixin for votable models.

get_score(*self*, *key*)
Return the score for the current model instance and *key*. Useful attrs:

- self.get_score(mykey).average
- self.get_score(mykey).total
- self.get_score(mykey).num_votes

If score does not exist, return None.

9.6 Managers

class ratings.managers.**RatingsManager**(*models.Manager*)
Manager used by *Score* and *Vote* models.

get_for(*self*, *content_object*, *key*, ***kwargs*)
Return the instance related to *content_object* and matching *kwargs*. Return None if a vote is not found.

filter_for(*self*, *content_object_or_model*, ***kwargs*)
Return all the instances related to *content_object_or_model* and matching *kwargs*. The argument *content_object_or_model* can be both a model instance or a model class.

filter_with_contents(*self*, ***kwargs*)
Return all instances retreiving content objects in bulk in order to minimize db queries, e.g. to get all objects voted by a user:

```
for vote in Vote.objects.filter_with_contents(user=myuser):
    vote.content_object # this does not hit the db
```

CHAPTER 10

Management commands reference

class ratings.management.commands.upsert_scores.**Command**

Create or update all scores, based on existing votes. This is useful if you have to migrate your votes from a legacy table, or you want to change the weight of current votes, e.g.:

```
./manage.y upsert_scores -w 5
```

r

A

B

C

D

F

G

H

N

P

R

S

U

V

W