django-comments-xtd Documentation

Release 2.0.4

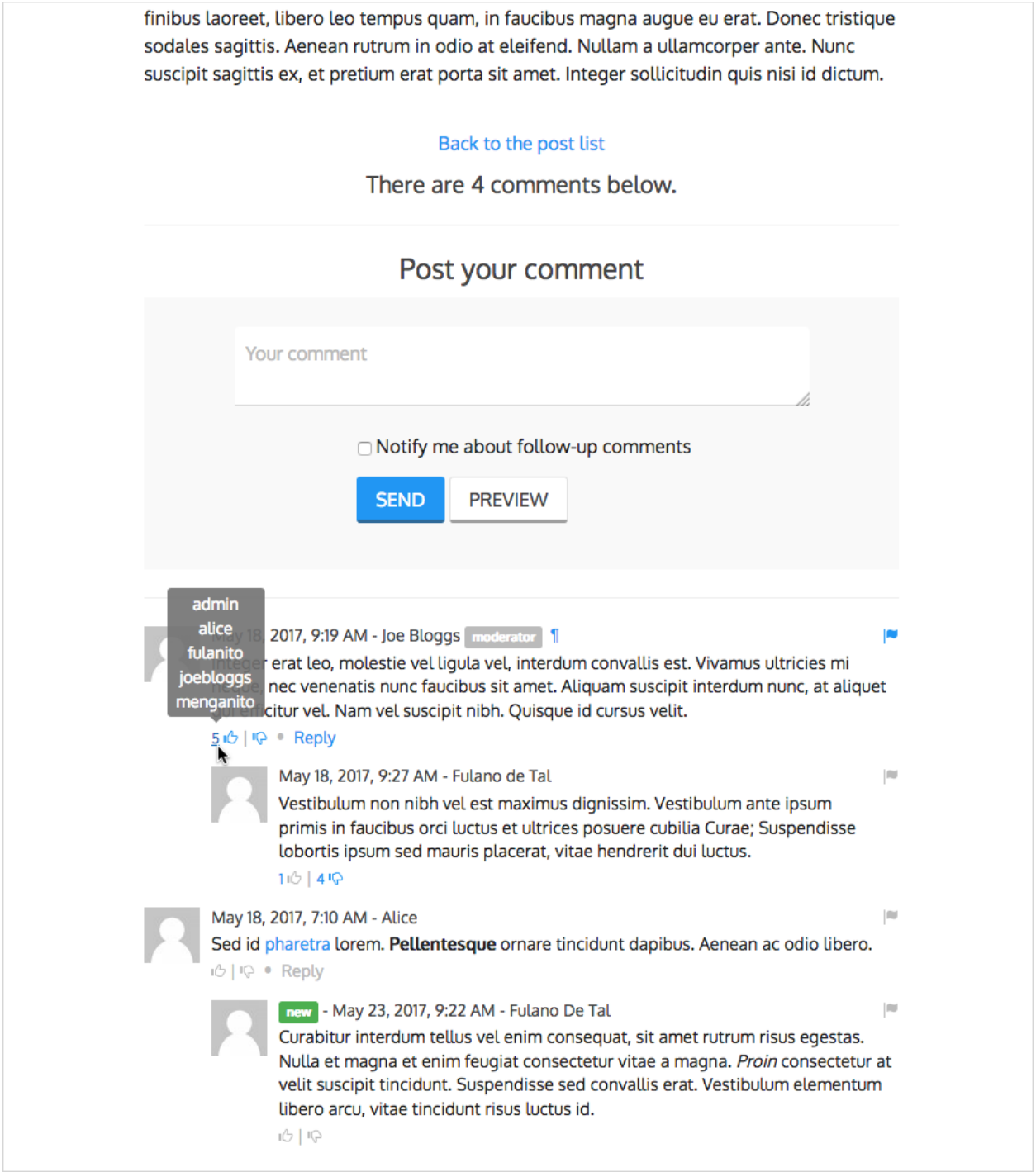
Daniel Rus Morales

Jul 19, 2017

Contents

A Django pluggable application that adds comments to your project. It extends the once official Django Comments Framework with the following features:

1. Thread support, so comments can be nested.
2. Customizable maximum thread level, either for all models or on a per app.model basis.
3. Optional notifications on follow-up comments via email.
4. Mute links to allow cancellation of follow-up notifications.
5. Comment confirmation via email when users are not authenticated.
6. Comments hit the database only after they have been confirmed.
7. Registered users can like/dislike comments and can suggest comments removal.
8. Template tags to list/render the last N comments posted to any given list of app.model pairs.
9. Emails sent through threads (can be disable to allow other solutions, like a Celery app).
10. Fully functional JavaScript plugin using ReactJS, jQuery, Bootstrap, Remarkable and MD5.


finibus laoreet, libero leo tempus quam, in faucibus magna augue eu erat. Donec tristique sodales sagittis. Aenean rutrum in odio at eleifend. Nullam a ullamcorper ante. Nunc suscipit sagittis ex, et pretium erat porta sit amet. Integer sollicitudin quis nisi id dictum.
Back to the post list
There are 4 comments below.
Post your comment
Your comment
Notify me about follow-up comments
SEND
PREVIEW
admin
alice
fulanito
joebloggs
menganito
2017, 9:19 AM - Joe Bloggs
moderator
erat leo, molestie vel ligula vel, interdum convallis est. Vivamus ultricies mi
nec venenatis nunc faucibus sit amet. Aliquam suscipit interdum nunc, at aliquet
citur vel. Nam vel suscipit nibh. Quisque id cursus velit.
5
Reply
May 18, 2017, 9:27 AM - Fulano de Tal
Vestibulum non nibh vel est maximus dignissim. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Suspendisse lobortis ipsum sed mauris placerat, vitae hendrerit dui luctus.
1
4
May 18, 2017, 7:10 AM - Alice
Sed id pharetra lorem. Pellentesque ornare tincidunt dapibus. Aenean ac odio libero.
Reply
new
- May 23, 2017, 9:22 AM - Fulano De Tal
Curabitur interdum tellus vel enim consequat, sit amet rutrum risus egestas. Nulla et magna et enim feugiat consectetur vitae a magna. Proin consectetur at velit suscipit tincidunt. Suspendisse sed convallis erat. Vestibulum elementum libero arcu, vitae tincidunt risus luctus id.

CHAPTER 1

Contents

Quick start guide

To get started using django-comments-xtd follow these steps:

1. `pip install django-comments-xtd`
2. Enable the "sites" framework by adding `'django.contrib.sites'` to `INSTALLED_APPS` and defining `SITE_ID`. Visit the admin site and be sure that the domain field of the `Site` instance points to the correct domain (`localhost:8000` when running the default development server), as it will be used to create comment verification URLs, follow-up cancellations, etc.
3. Add `'django_comments_xtd'` and `'django_comments'`, in that order, to `INSTALLED_APPS`.
4. Set the `COMMENTS_APP` setting to `'django_comments_xtd'`.
5. Set the *`COMMENTS_XTD_MAX_THREAD_LEVEL`* to `N`, being `N` the maximum level of threading up to which comments will be nested in your project.

```
# 0: No nested comments:
#  Comment (level 0)
# 1: Nested up to level one:
#  Comment (level 0)
#    |-- Comment (level 1)
# 2: Nested up to level two:
#  Comment (level 0)
#    |-- Comment (level 1)
#          |-- Comment (level 2)
COMMENTS_XTD_MAX_THREAD_LEVEL = 2
```

   The thread level can also be established on a per `<app>.<model>` basis by using the *`COMMENTS_XTD_MAX_THREAD_LEVEL_BY_APP_MODEL`* setting. Use it to establish different maximum threading levels for each model. ie: no nested comments for quotes, up to thread level 2 for blog stories, etc.

6. Set the *COMMENTS_XTD_CONFIRM_EMAIL* to `True` to require comment confirmation by email for no logged-in users.
7. Run `manage.py migrate` to create the tables.
8. Add the URLs of the comments-xtd app to your project's `urls.py`:

```
urlpatterns = [
    ...
    url(r'^comments/', include('django_comments_xtd.urls')),
    ...
]
```

9. Customize your project's email settings:

```
EMAIL_HOST = "smtp.mail.com"
EMAIL_PORT = "587"
EMAIL_HOST_USER = "alias@mail.com"
EMAIL_HOST_PASSWORD = "yourpassword"
DEFAULT_FROM_EMAIL = "Helpdesk <helpdesk@yourdomain>"
```

10. As of version 1.7.1 django-comments-xtd comes with templates styled with twitter-bootstrap v3 to allow a quick start. If you want to build your own templates, use the comments templatetag module, provided by the django-comments app. Create a `comments` directory in your templates directory and copy the templates you want to customise from the Django Comments Framework. The following are the most important:
    - `comments/list.html`, used by the `render_comments_list` templatetag.
    - `comments/form.html`, used by the `render_comment_form` templatetag.
    - `comments/preview.html`, used to preview the comment or when there are errors submitting it.
    - `comments/posted.html`, which gets rendered after the comment is sent.
11. Add extra settings to control comments in your project. Check the available settings in the Django Comments Framework and in the *django-comments-xtd app*.

These are the steps to quickly start using django-comments-xtd. Follow to the next page, the *Tutorial*, to read a detailed guide that takes everything into account. In addition to the tutorial, the *Demo projects* implement several commenting applications.

Tutorial

This tutorial guides you through the steps to use every feature of django-comments-xtd together with the Django Comments Framework. The Django project used throughout the tutorial is available to download. Following the tutorial will take about an hour, and it is highly recommended to get a comprehensive understanding of django-comments-xtd.

Table of Contents

Introduction

Through the following sections the tutorial will cover the creation of a simple blog with stories to which we will add comments, exercising each and every feature provided by both, django-comments and django-comments-xtd, from comment post verification by mail to comment moderation and nested comments.

Preparation

Before we install any package we will set up a virtualenv and install everything we need in it.

```
$ mkdir ~/django-comments-xtd-tutorial
$ cd ~/django-comments-xtd-tutorial
$ virtualenv venv
$ source venv/bin/activate
(venv)$ pip install django-comments-xtd
(venv)$ wget https://github.com/danirus/django-comments-xtd/raw/master/
↪example/tutorial.tar.gz
(venv)$ tar -xvzf tutorial.tar.gz
(venv)$ cd tutorial
```

By installing django-comments-xtd we install all its dependencies, Django and django-contrib-comments among them. So we are ready to work on the project. Take a look at the content of the tutorial directory, it contains:

- A **blog** app with a **Post** model. It uses two generic class-based views to list the posts and show a post in detail.

- The **templates** directory, with a **base.html** and **home.html**, and the templates for the blog app: **blog/post_list.html** and **blog/post_detail.html**.
- The **static** directory with a **css/bootstrap.min.css** file (this file is a static asset available, when the app is installed, under the path **django_comments_xtd/css/bootstrap.min.css**).
- The **tutorial** directory containing the **settings** and **urls** modules.
- And a **fixtures** directory with data files to create the *admin* superuser (with *admin* password), the default site and some blog posts.

Let's finish the initial setup, load the fixtures and run the development server:

```
(venv)$ python manage.py migrate
(venv)$ python manage.py loaddata fixtures/*.json
(venv)$ python manage.py runserver
```

Head to http://localhost:8000 and visit the tutorial site.

Configuration

Now that the project is running we are ready to add comments. Edit the settings module, `tutorial/settings.py`, and make the following changes:

```
INSTALLED_APPS = [
    ...
    'django_comments_xtd',
    'django_comments',
    'blog',
]
...
COMMENTS_APP = 'django_comments_xtd'

# Either enable sending mail messages to the console:
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'

# Or set up the EMAIL_* settings so that Django can send emails:
EMAIL_HOST = "smtp.mail.com"
EMAIL_PORT = "587"
EMAIL_HOST_USER = "alias@mail.com"
EMAIL_HOST_PASSWORD = "yourpassword"
EMAIL_USE_TLS = True
DEFAULT_FROM_EMAIL = "Helpdesk <helpdesk@yourdomain>"
```

Edit the urls module of the project, `tutorial/tutorial/urls.py` and mount the URL patterns of django_comments_xtd in the path `/comments/`. The urls installed with django_comments_xtd include django_comments' urls too:

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

urlpatterns = [
    ...
    url(r'^comments/', include('django_comments_xtd.urls')),
    ...
]
```

Now let Django create the tables for the two new applications:

```
$ python manage.py migrate
```

Be sure that the domain field of the `Site` instance points to the correct domain, which for the development server is expected to be `localhost:8000`. The value is used to create comment verifications, follow-up cancellations, etc. Edit the site instance in the admin interface in case you were using a different value.

Comment confirmation

Before we go any further we need to set up the *COMMENTS_XTD_SALT* setting. This setting plays an important role during the comment confirmation by mail. It helps obfuscating the comment before the user approves its publication.

It is so because django-comments-xtd does not store comments in the server before they have been confirmed. This way there is little to none possible comment spam flooding in the database. Comments are encoded in URLs and sent for confirmation by mail. Only when the user clicks the confirmation URL the comment lands in the database.

This behaviour is disabled for authenticated users, and can be disabled for anonymous users too by simply setting `COMMENTS_XTD_CONFIRM_MAIL` to `False`.

Now let's append the following entries to the tutorial settings module:

```
#  To help obfuscating comments before they are sent for confirmation.
COMMENTS_XTD_SALT = (b"Timendi causa est nescire. "
                     b"Aequam memento rebus in arduis servare mentem.")

# Source mail address used for notifications.
COMMENTS_XTD_FROM_EMAIL = "noreply@example.com"

# Contact mail address to show in messages.
COMMENTS_XTD_CONTACT_EMAIL = "helpdesk@example.com"
```

Comments tags

Next step consist of editing `blog/post_detail.html` and loading the `comments` templatetag module after the `extends` tag:

```
{% extends "base.html" %}
{% load comments %}
```

Now we will change the blog post detail template to:

1. Show the number of comments posted to the blog story,
2. List the comments already posted, and
3. Show the comment form, so that comments can be sent.

By using the `get_comment_count` tag we will show the number of comments posted. Change the code around the link element to make it look as follows:

```
{% get_comment_count for object as comment_count %}
<div class="text-center" style="padding-top:20px">
  <a href="{% url 'blog:post-list' %}">Back to the post list</a>
   &sdot; 
  {{ comment_count }} comments have been posted.
</div>
```

Now let's add the code to list the comments posted to the story. We can make use of two template tags, `render_comment_list` and `get_comment_list`. The former renders a template with the comments while the latter put the comment list in a variable in the context of the template.

When using the first, `render_comment_list`, with a `blog.post` object, Django will look for the template `list.html` in the following directories:

```
comments/blog/post/list.html
comments/blog/list.html
comments/list.html
```

Both, django-contrib-comments and django-comments-xtd, provide the last template of the list, `comments/list.html`. The one provided within django-comments-xtd comes with styling based on twitter-bootstrap.

Django will use the first template found depending on the order in which applications are listed in `INSTALLED_APPS`. In this tutorial django-comments-xtd is listed first and therefore its `comment/list.html` template will be found first.

Let's modify the `blog/blog_detail.html` template to make use of the `render_comment_list`. Add the following code at the end of the page, before the `endblock` tag:

```
{% if comment_count %}
<hr/>
<div class="comments">
  {% render_comment_list for object %}
</div>
{% endif %}
```

Below the list of comments we want to display the comment form. There are two template tags available for that purpose, the `render_comment_form` and the `get_comment_form`. The former renders a template with the comment form while the latter puts the form in the context of the template giving more control over the fields.

We will use the first tag, `render_comment_form`. Again, add the following code before the `endblock` tag:

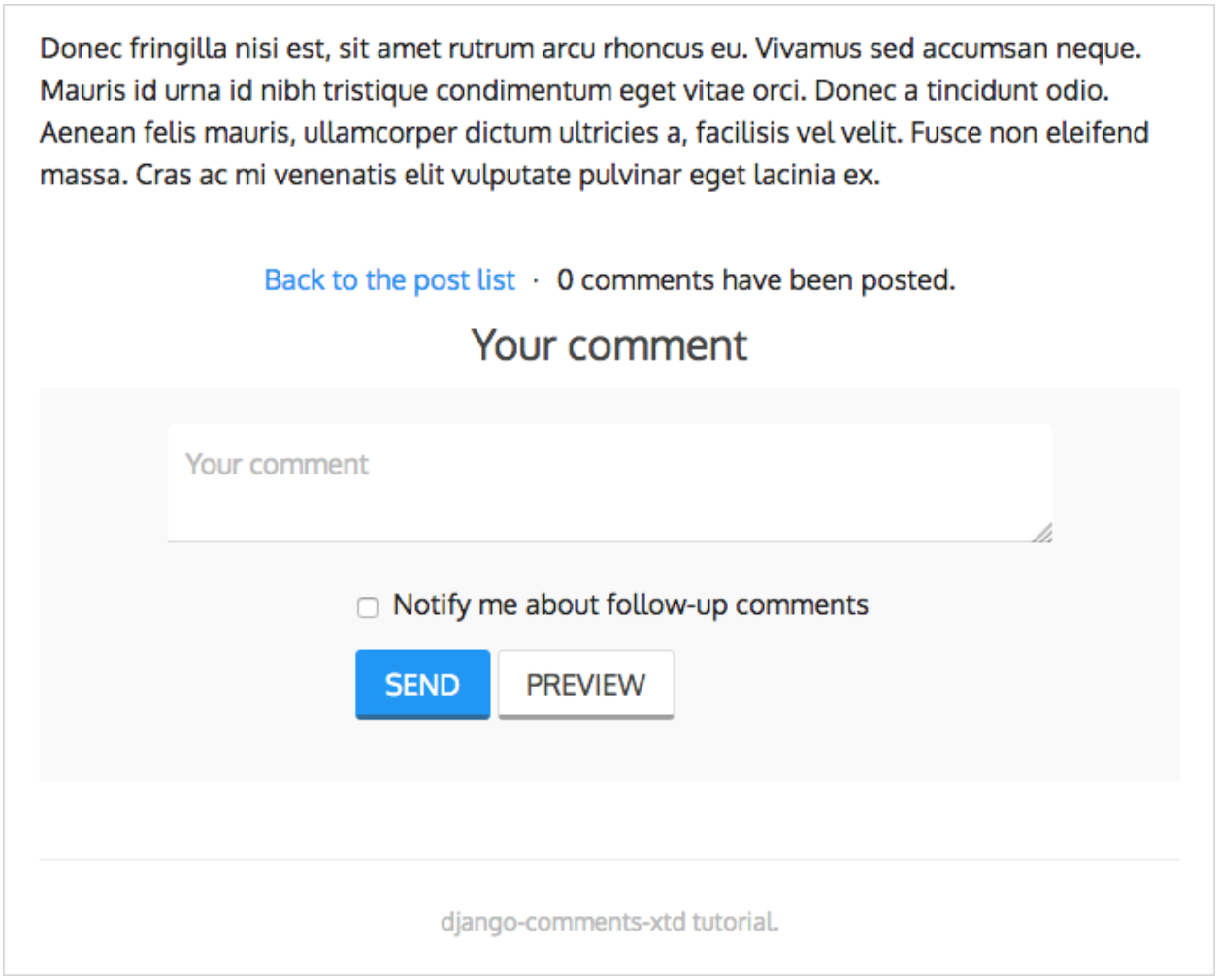
```
{% if object.allow_comments %}
<div class="comment">
  <h4 class="text-center">Your comment</h4>
  <div class="well">
    {% render_comment_form for object %}
  </div>
</div>
{% endif %}
```

Finally, before completing this first set of changes, we could show the number of comments along with post titles in the blog's home page. For this we have to edit `blog/post_list.html` and make the following changes:

```
{% extends "base.html" %}
{% load comments %}

...
<p class="date">
  {% get_comment_count for object as comment_count %}
  Published {{ object.publish }}
  {% if comment_count %}
  &sdot; {{ comment_count }} comments
  {% endif %}
</p>
```

Now we are ready to send comments. If you are logged in the admin site, your comments won't need to be confirmed

by mail. To test the confirmation URL do logout of the admin interface. Bear in mind that EMAIL_BACKEND is set up to send mail messages to the console, so look in the console after you post the comment and find the first long URL in the message. To confirm the comment copy the link and paste it in the location bar of the browser.



The setting *COMMENTS_XTD_MAX_THREAD_LEVEL* is 0 by default, which means comments can not be nested. Later in the threads section we will enable nested comments. Now we will set up comment moderation.

Moderation

One of the differences between django-comments-xtd and other commenting applications is the fact that by default it requires comment confirmation by email when users are not logged in, a very effective feature to discard unwanted comments. However there might be cases in which you would prefer a different approach. Django Comments Framework comes with moderation capabilities included upon which you can build your own comment filtering.

Comment moderation is often established to fight spam, but may be used for other purposes, like triggering actions based on comment content, rejecting comments based on how old is the subject being commented and whatnot.

In this section we want to set up comment moderation for our blog application, so that comments sent to a blog post older than a year will be automatically flagged for moderation. Also we want Django to send an email to registered MANAGERS of the project when the comment is flagged.

Let's start adding our email address to the MANAGERS in the `tutorial/settings.py` module:

```
MANAGERS = (
    ('Joe Bloggs', 'joe.bloggs@example.com'),
)
```

Now we will create a new `Moderator` class that inherits from Django Comments Frammework's `CommentModerator`. This class enables moderation by defining a number of class attributes. Read more about it in moderation options, in the official documentation of the Django Comments Framework.

We will also register our `Moderator` class with the django-comments-xtd's `moderator` object. We use django-comments-xtd's object instead of django-contrib-comments' because we still want to have confirmation by email for non-registered users, nested comments, follow-up notifications, etc.

Let's add those changes to the `blog/model.py` file:

```
...
# Append these imports below the current ones.
from django_comments.moderation import CommentModerator
from django_comments_xtd.moderation import moderator

...

# Add this code at the end of the file.
class PostCommentModerator(CommentModerator):
    email_notification = True
    auto_moderate_field = 'publish'
    moderate_after = 365


moderator.register(Post, PostCommentModerator)
```

That makes it, moderation is ready. Visit any of the blog posts with a `publish` datetime older than a year and try to send a comment. After confirming the comment you will see the `django_comments_xtd/moderated.html` template, and your comment will be put on hold for approval.

If on the other hand you send a comment to a blog post created within the last year your comment will not be put in moderation. Give it a try as a logged in user and as an anonymous user.

When sending a comment as a logged-in user the comment won't have to be confirmed and will be put in moderation immediately. However, when you send it as an anonymous user the comment will have to be confirmed by clicking on the confirmation link, immediately after that the comment will be put on hold pending for approval.

In both cases, due to the attribute `email_notification = True` above, all mail addresses listed in the `MANAGERS` setting will receive a notification about the reception of a new comment. If you did not received such message, you might need to review your email settings, or the console output. Read about the mail settings above in the *Configuration* section. The mail message received is based on the `comments/comment_notification_email.txt` template provided with django-comments-xtd.

A last note on comment moderation: comments pending for moderation have to be reviewed and eventually approved. Don't forget to visit the comments-xtd app in the admin interface. Filter comments by *is public: No* and *is removed: No*. Tick the box of those you want to approve, choose **Approve selected comments** in the **action** dropdown, at the top left of the comment list, and click on the **Go** button.

Disallow black listed domains

In case you wanted to disable comment confirmation by mail you might want to set up some sort of control to reject spam.

In this section we will go through the steps to disable comment confirmation while enabling a comment filtering solution based on Joe Wein's blacklist of spamming domains. We will also add a moderation function that will put in moderation comments containing badwords.

Let us first disable comment confirmation. Edit the `tutorial/settings.py` file and add:

```
COMMENTS_XTD_CONFIRM_EMAIL = False
```

django-comments-xtd comes with a **Moderator** class that inherits from `CommentModerator` and implements a method `allow` that will do the filtering for us. We just have to change `blog/models.py` and replace `CommentModerator` with `SpamModerator`, as follows:

```
# Remove the CommentModerator imports and leave only this:
from django_comments_xtd.moderation import moderator, SpamModerator

# Our class Post PostCommentModerator now inherits from SpamModerator
class PostCommentModerator(SpamModerator):
    ...

moderator.register(Post, PostCommentModerator)
```

Now we can add a domain to the `BlackListed` model in the admin interface. Or we could download a blacklist from Joe Wein's website and load the table with actual spamming domains.

Once we have a `BlackListed` domain, try to send a new comment and use an email address with such a domain. Be sure to log out before trying, otherwise django-comments-xtd will use the logged in user credentials and ignore the email given in the comment form.

Sending a comment with an email address of the blacklisted domain triggers a **Comment post not allowed** response, which would have been a HTTP 400 Bad Request response with `DEBUG = False` in production.

Moderate on bad words

Let's now create our own Moderator class by subclassing `SpamModerator`. The goal is to provide a `moderate` method that looks in the content of the comment and returns `False` whenever it finds a bad word in the message. The effect of returning `False` is that comment's `is_public` attribute will be put to `False` and therefore the comment will be in moderation.

The blog application comes with a bad word list in the file `blog/badwords.py`

We assume we already have a list of `BlackListed` domains and we don't need further spam control. So we will disable comment confirmation by email. Edit the `settings.py` file:

```
COMMENTS_XTD_CONFIRM_EMAIL = False
```

Now edit `blog/models.py` and add the code corresponding to our new `PostCommentModerator`:

```
# Below the other imports:
from django_comments_xtd.moderation import moderator, SpamModerator
from blog.badwords import badwords

...

class PostCommentModerator(SpamModerator):
    email_notification = True

    def moderate(self, comment, content_object, request):
        # Make a dictionary where the keys are the words of the message and
```

```
        # the values are their relative position in the message.
        def clean(word):
            ret = word
            if word.startswith('.') or word.startswith(','):
                ret = word[1:]
            if word.endswith('.') or word.endswith(','):
                ret = word[:-1]
            return ret

        lowcase_comment = comment.comment.lower()
        msg = dict([(clean(w), i)
                    for i, w in enumerate(lowcase_comment.split())])
        for badword in badwords:
            if isinstance(badword, str):
                if lowcase_comment.find(badword) > -1:
                    return True
            else:
                lastindex = -1
                for subword in badword:
                    if subword in msg:
                        if lastindex > -1:
                            if msg[subword] == (lastindex + 1):
                                lastindex = msg[subword]
                            else:
                                lastindex = msg[subword]
                        else:
                            break
                if msg.get(badword[-1]) and msg[badword[-1]] == lastindex:
                    return True
        return super(PostCommentModerator, self).moderate(comment,
                                                          content_object,
                                                          request)

moderator.register(Post, PostCommentModerator)
```

Now we can try to send a comment with any of the bad words listed in badwords. After sending the comment we will see the content of the `django_comments_xtd/moderated.html` template and the comment will be put in moderation.

If you enable comment confirmation by email, the comment will be put on hold after the user clicks on the confirmation link in the email.

Threads

Up until this point in the tutorial django-comments-xtd has been configured to disallow nested comments. Every comment is at thread level 0. It is so because by default the setting *COMMENTS_XTD_MAX_THREAD_LEVEL* is set to 0.

When the *COMMENTS_XTD_MAX_THREAD_LEVEL* is greater than 0, comments below the maximum thread level may receive replies that will nest inside each other up to the maximum thread level. A comment in a the thread level below the *COMMENTS_XTD_MAX_THREAD_LEVEL* can show a **Reply** link that allows users to send nested comments.

In this section we will enable nested comments by modifying *COMMENTS_XTD_MAX_THREAD_LEVEL* and apply some changes to our `blog_detail.html` template.

We can make use of two template tags, *render_xtdcomment_tree* and *get_xtdcomment_tree*. The former

renders a template with the comments while the latter put the comments in a nested data structure in the context of the template.

We will also introduce the setting *COMMENTS_XTD_LIST_ORDER*, that allows altering the default order in which the comments are sorted in the list. By default comments are sorted by thread and their position inside the thread, which turns out to be in ascending datetime of arrival. In this example we will list newer comments first.

Let's start by editing `tutorial/settings.py` to set up the maximum thread level to 1 and a comment ordering such that newer comments are retrieve first:

```
COMMENTS_XTD_MAX_THREAD_LEVEL = 1  # default is 0
COMMENTS_XTD_LIST_ORDER = ('-thread_id', 'order')  # default is ('thread_id',
↪ 'order')
```

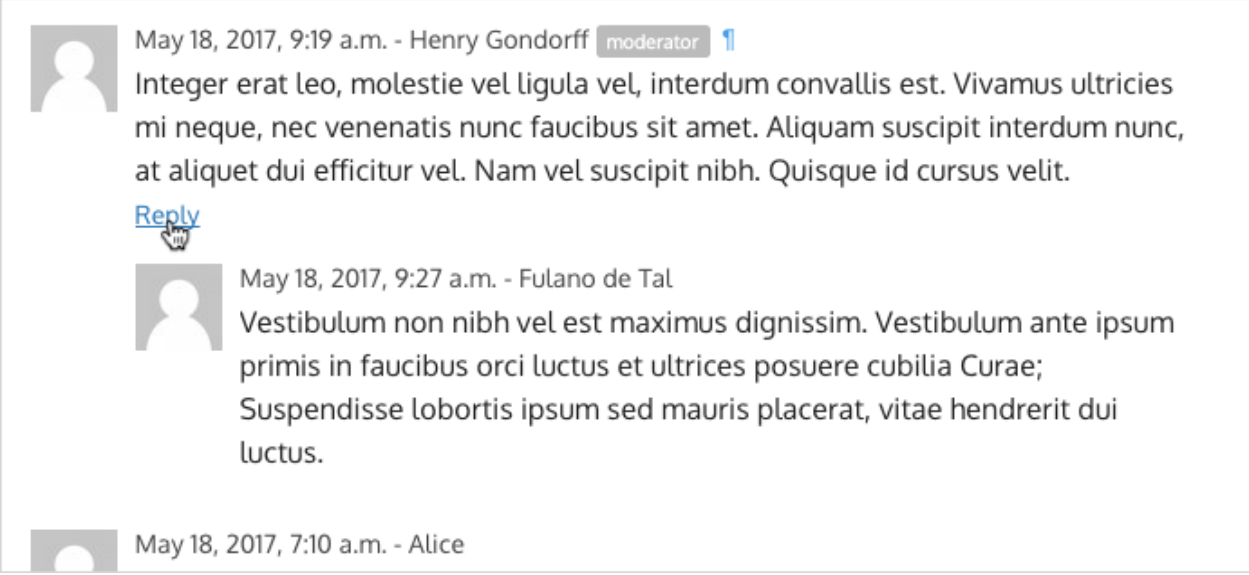
Now we have to modify the blog post detail template to load the `comments_xtd` templatetag and make use of *render_xtdcomment_tree*. We also want to move the comment form from the bottom of the page to a more visible position right below the blog post, followed by the list of comments.

Edit `blog/post_detail.html` to make it look like follows:

```
{% extends "base.html" %}
{% load comments %}
{% load comments_xtd %}

{% block title %}{{ object.title }}{% endblock %}

{% block content %}
<h3 class="page-header text-center">{{ object.title }}</h3>
<p class="small text-center">{{ object.publish|date:"l, j F Y" }}</p>
<p>
  {{ object.body|linebreaks }}
</p>

{% get_comment_count for object as comment_count %}
<div class="text-center" style="padding-top:20px">
  <a href="{% url 'blog:post-list' %}">Back to the post list</a>
   &sdot; 
  {{ comment_count }} comments have been posted.
</div>

{% if object.allow_comments %}
<div class="comment">
  <h4 class="text-center">Your comment</h4>
  <div class="well">
    {% render_comment_form for object %}
  </div>
</div>
{% endif %}

{% if comment_count %}
<hr/>
<ul class="media-list">
  {% render_xtdcomment_tree for object %}
</ul>
{% endif %}
{% endblock %}
```

The tag *render_xtdcomment_tree* renders the template `django_comments_xtd/comment_tree.html`.

Now visit any of the blog posts to which you have already sent comments and see that a new *Reply* link shows up below each comment. Click on the link and post a new comment. It will appear nested inside the parent comment. The new comment will not show a *Reply* link because *COMMENTS_XTD_MAX_THREAD_LEVEL* has been set to 1. Raise it to 2 and reload the page to offer the chance to nest comments inside one level deeper.



Different max thread levels

There might be cases in which nested comments have a lot of sense and others in which we would prefer a plain comment sequence. We can handle both scenarios under the same Django project.

We just have to use both settings, *COMMENTS_XTD_MAX_THREAD_LEVEL* and *COMMENTS_XTD_MAX_THREAD_LEVEL_BY_APP_MODEL*. The former establishes the default maximum thread level site wide, while the latter sets the maximum thread level on *app.model* basis.

If we wanted to disable nested comments site wide, and enable nested comments up to level one for blog posts, we would set it up as follows in our `settings.py` module:

```
COMMENTS_XTD_MAX_THREAD_LEVEL = 0  # site wide default
COMMENTS_XTD_MAX_THREAD_LEVEL_BY_MODEL = {
    # Objects of the app blog, model post, can be nested
    # up to thread level 1.
        'blog.post': 1,
}
```

Flags

The Django Comments Framework supports comment flagging, so comments can be flagged for:

- **Removal suggestion**, when a registered user suggests the removal of a comment.
- **Moderator deletion**, when a comment moderator marks the comment as deleted.
- **Moderator approval**, when a comment moderator sets the comment as approved.

django-comments-xtd expands flagging with two more flags:

- **Liked it**, when a registered user likes the comment.

- **Disliked it**, when a registered user dislikes the comment.

In this section we will see how to enable a user with the capacity to flag a comment for removal with the **Removal suggestion** flag, how to express likeability, conformity, acceptance or acknowledgement with the **Liked it** flag and the opposite with the **Disliked it** flag.

One important requirement to mark comments is that the user flagging must be authenticated. In other words, comments can not be flagged by anonymous users.

Commenting options

As of version 2.0 of django-comments-xtd there is a new setting, *COMMENTS_XTD_APP_MODEL_OPTIONS*, that must be correctly setup to allow flagging. The purpose is to give an additional level of control about what action users can do on comments: flag them as inappropriate, like/dislike them, and retrieve the list of users who liked/disliked them.

It defaults to:

```
COMMENTS_XTD_APP_MODEL_OPTIONS = {
    'default': {
        'allow_flagging': False,
        'allow_feedback': False,
        'show_feedback': False,
    }
}
```

We will enable each option alongside the following sections.

Removal suggestion

Enabling the comment removal flag is about including the **allow_flagging** argument in the `render_xtdcomment_tree` template tag. Edit the `blog/post_detail.html` template and append the argument:

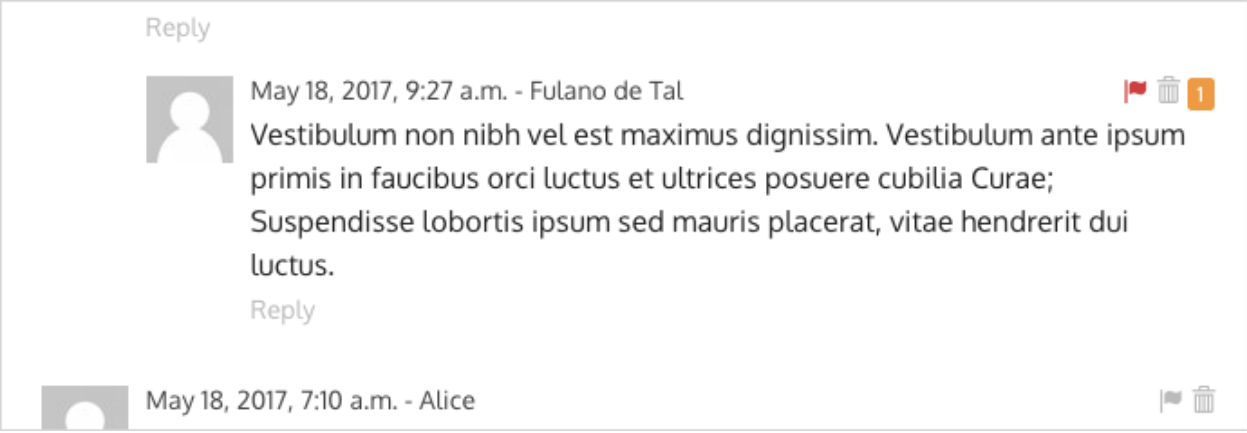
```
...
<ul class="media-list">
  {% render_xtdcomment_tree for object allow_flagging %}
</ul>
```

The **allow_flagging** argument makes the templatetag populate a variable `allow_flagging = True` in the context in which `django_comments_xtd/comment_tree.html` is rendered. Edit now the settings module and enable the `allow_flagging` option for the `blog.post` **app.label** pair:

```
COMMENTS_XTD_APP_MODEL_OPTIONS = {
    'blog.post': {
        'allow_flagging': True,
        'allow_feedback': False,
        'show_feedback': False,
    }
}
```

Now let's suggest a removal. First we need to login in the admin interface so that we are not an anonymous user. Then we can visit any of the blog posts we sent comments to. There is a flag at the right side of every comment's header. Clicking on it bring the user to a page in which she is requested to confirm the removal suggestion. Finally, clicking on the red **Flag** button confirms the request.

Users with the `django_comments.can_moderate` permission will see a yellow labelled counter near the flag button in each flagged comment, representing how many times comments have been flagged. Also notice that when a user flags a comment for removal the icon turns red for that user.



Administrators/moderators can find flagged comment entries in the admin interface, under the **Comment flags** model, within the Django Comments application.

Getting notifications

A user might want to flag a comment on the basis of a violation of the site's terms of use, hate speech, racism or the like. To prevent a comment from staying published long after it has been flagged we might want to receive notifications on flagging events.

For such purpose django-comments-xtd provides the class **XtdCommentModerator**, which extends django-contrib-comments' **CommentModerator**.

In addition to all the options of its parent class, **XtdCommentModerator** offers the `removal_suggestion_notification` attribute, that when set to `True` makes Django send a mail to all the `MANAGERS` on every **Removal suggestion** flag created.

To see an example let's edit `blog/models.py`. If you are already using the class **SpamModerator**, which inherits from **XtdCommentModerator**, just add `removal_suggestion_notification = True` to your `PostCommentModeration` class. Otherwise add the following code:

```
from django_comments_xtd.moderation import moderator, XtdCommentModerator

...
class PostCommentModerator(XtdCommentModerator):
    removal_suggestion_notification = True

moderator.register(Post, PostCommentModerator)
```

Be sure that `PostCommentModerator` is the only moderation class registered for the `Post` model, and be sure as well that the `MANAGERS` setting contains a valid email address. The message sent is based on the `django_comments_xtd/removal_notification_email.txt` template, already provided within django-comments-xtd. After these changes flagging a comment with a **Removal suggestion** will trigger a notification by mail.

Liked it, Disliked it

Django-comments-xtd adds two new flags: the **Liked it** and the **Disliked it** flags.

Unlike the **Removal suggestion** flag, the **Liked it** and **Disliked it** flags are mutually exclusive. A user can not like and dislike a comment at the same time. Users can like/dislike at any time and only the last action will prevail.

In this section we make changes to give our users the capacity to like or dislike comments. Following the same pattern as with the removal flag, enabling like/dislike buttons is about adding an argument to the `render_xtdcomment_tree`, the argument `allow_feedback`. Edit the `blog/post_detail.html` template and add the new argument:

```
<ul class="media-list">
  {% render_xtdcomment_tree for object allow_flagging allow_feedback %}
</ul>
```

The **allow_feedback** argument makes the templatetag populate a variable `allow_feedback = True` in the context in which `django_comments_xtd/comment_tree.html` is rendered. Edit the settings module and enable the `allow_feedback` option for the `blog.post` **app.label** pair:

```
COMMENTS_XTD_APP_MODEL_OPTIONS = {
    'blog.post': {
        'allow_flagging': True,
        'allow_feedback': True,
        'show_feedback': False,
    }
}
```

The blog post detail template is ready to show the like/dislike buttons, refresh your browser.

Having the new like/dislike links in place, if we click on any of them we will end up in either the `django_comments_xtd/like.html` or the `django_comments_xtd/dislike.html` templates, which are meant to request the user a confirmation for the operation.

Show the list of users

With the like/dislike buttons enabled we might as well consider to display the users who actually liked/disliked comments. Again addind an argument to the `render_xtdcomment_tree` will enable the feature. Change the `blog/post_detail.html` and add the argument `show_feedback` to the template tag:

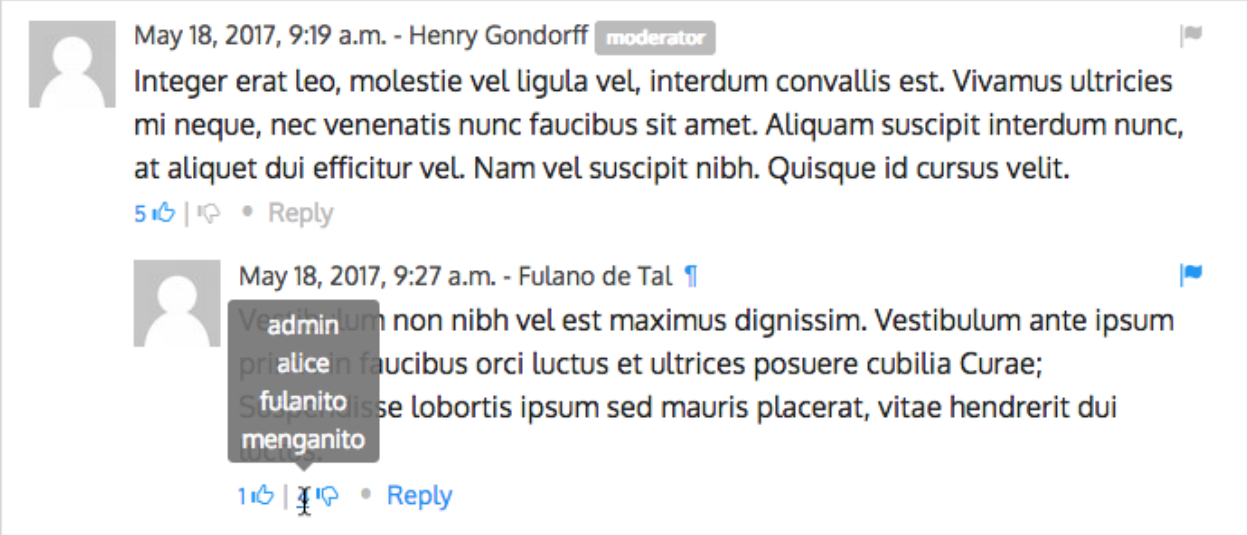
```
<ul class="media-list">
  {% render_xtdcomment_tree for object allow_flagging allow_feedback show_
↪feedback %}
</ul>

{% block extra-js %}
<script
    src="https://code.jquery.com/jquery-2.2.4.min.js"
    integrity="sha256-BbhdlvQf/xTY9gja0Dq3HiwQF8LaCRTXxZKRutelT44="
    crossorigin="anonymous"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.
↪min.js"
    integrity="sha384-
↪Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa"
    crossorigin="anonymous"></script>
<script>
$(function () {
```

```
  $('[data-toggle="tooltip"]').tooltip({html: true})
})</script>
{% endblock %}
```

Also change the settings and enable the `show_feedback` option for `blog.post`:

```
COMMENTS_XTD_APP_MODEL_OPTIONS = {
    'blog.post': {
        'allow_flagging': True,
        'allow_feedback': True,
        'show_feedback': True,
    }
}
```

We loaded jQuery and twitter-bootstrap libraries from their respective default CDNs as the code above uses bootstrap's tooltip functionality to show the list of users when the mouse hovers the numbers near the buttons, as shows the image:



Put the mouse over the counters near the like/dislike buttons to display the list of users.

Markdown

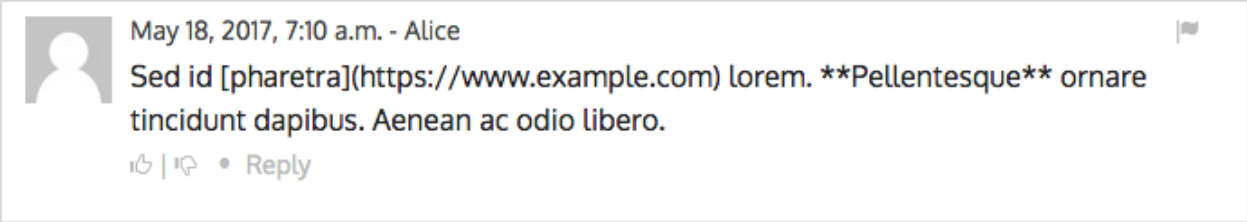
In versions prior to 2.0 django-comments-xtd required the installation of django-markup as a dependency. There was also a specific template filter called `render_markup_comment` to help rendering comment's content in the markup language of choice.

As of version 2.0 the backend side of the application does not require the installation of any additional package to parser comments' content, and therefore does not provide the `render_markup_comment` filter anymore. However, in the client side the JavaScript plugin uses Markdown by default to render comments' content.

As for the backend side, comment's content is presented by default in plain text, but it is easily customizable by overriding the template `includes/django_comments_xtd/render_comment.html`.

In this section we will send a Markdown formatted comment, and once published we will install support for Markdown, with django-markdown2. We'll then override the template mentioned above so that comments are interpreted as Markdown.

Send a comment formatted in Markdown, as the one in the following image.



Now we will install django-markdown2, and create the template directory and the template file:

```
(venv)$ pip install django-markdown2
(venv)$ mkdir -p templates/includes/django_comments_xtd/
(venv)$ touch templates/includes/django_comments_xtd/comment_content.html
```

We have to add `django_markdown2` to our `INSTALLED_APPS`, and add the following template code to the file `comment_content.html` we just created:

```
{% load md2 %}
{{ content|markdown:"safe, code-friendly, code-color" }}
```

Now our project is ready to show comments posted in Markdown. After reloading, the comment's page will look like this:

JavaScript plugin

Up until now we have used django-comments-xtd as a backend application. As of version 2.0 it includes a JavaScript plugin that helps moving part of the logic to the browser improving the overall usability. By making use of the JavaScript plugin users don't have to leave the blog post page to preview, submit or reply comments, or to like/dislike them. But it comes at the cost of assuming our project will use:

- ReactJS
- jQuery (to handle Ajax calls).
- Twitter-Bootstrap (for the UI).
- Remarkable (for Markdown support).

To know more about the client side of the application and the build process read the specific page on the *JavaScript plugin*.

In this section of the tutorial we go through the steps to make use of the JavaScript plugin.

Enable Web API

The JavaScript plugin uses the Web API provided by within the app. In order to enable it install the django-rest-framework:

```
(venv)$ pip install djangorestframework
```

Once installed, add it to our tutorial `INSTALLED_APPS` setting:

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

To know more about the Web API provided by django-comments-xtd read on the *Web API* page.

Enable app.model options

Be sure *COMMENTS_XTD_APP_MODEL_OPTIONS* includes the options we want to enable for comments sent to Blog posts. In this case we will allow users to flag comments for removal (allow_flagging option), to like/dislike comments (allow_feedback), and we want users to see the list of people who liked/disliked comments:

```
COMMENTS_XTD_APP_MODEL_OPTIONS = {
    'blog.post': {
        'allow_flagging': True,
        'allow_feedback': True,
        'show_feedback': True,
    }
}
```

The i18n JavaScript Catalog

Internationalization support (see Internationalization) has been included within the plugin by making use of the Django's JavaScript i18n catalog. If your project doesn't need i18n you can easily remove every mention to these functions (namespaced under the *django* object) from the source and change the `webpack.config.js` file to build the plugin without it.

Our tutorial doesn't have i18n enabled (the comp example project has it), but we will not remove its support from the plugin, we will simply enable the JavaScript Catalog URL, so that the plugin can access its functions. Edit `tutorial/urls.py` and add the following url:

```
from django.views.i18n import JavaScriptCatalog

urlpatterns = [
    ...
    url(r'^jsi18n/$', JavaScriptCatalog.as_view(), name='javascript-catalog
↪'),
]
```

In the next section we will use the new URL to load the i18n JavaScript catalog.

Load the plugin

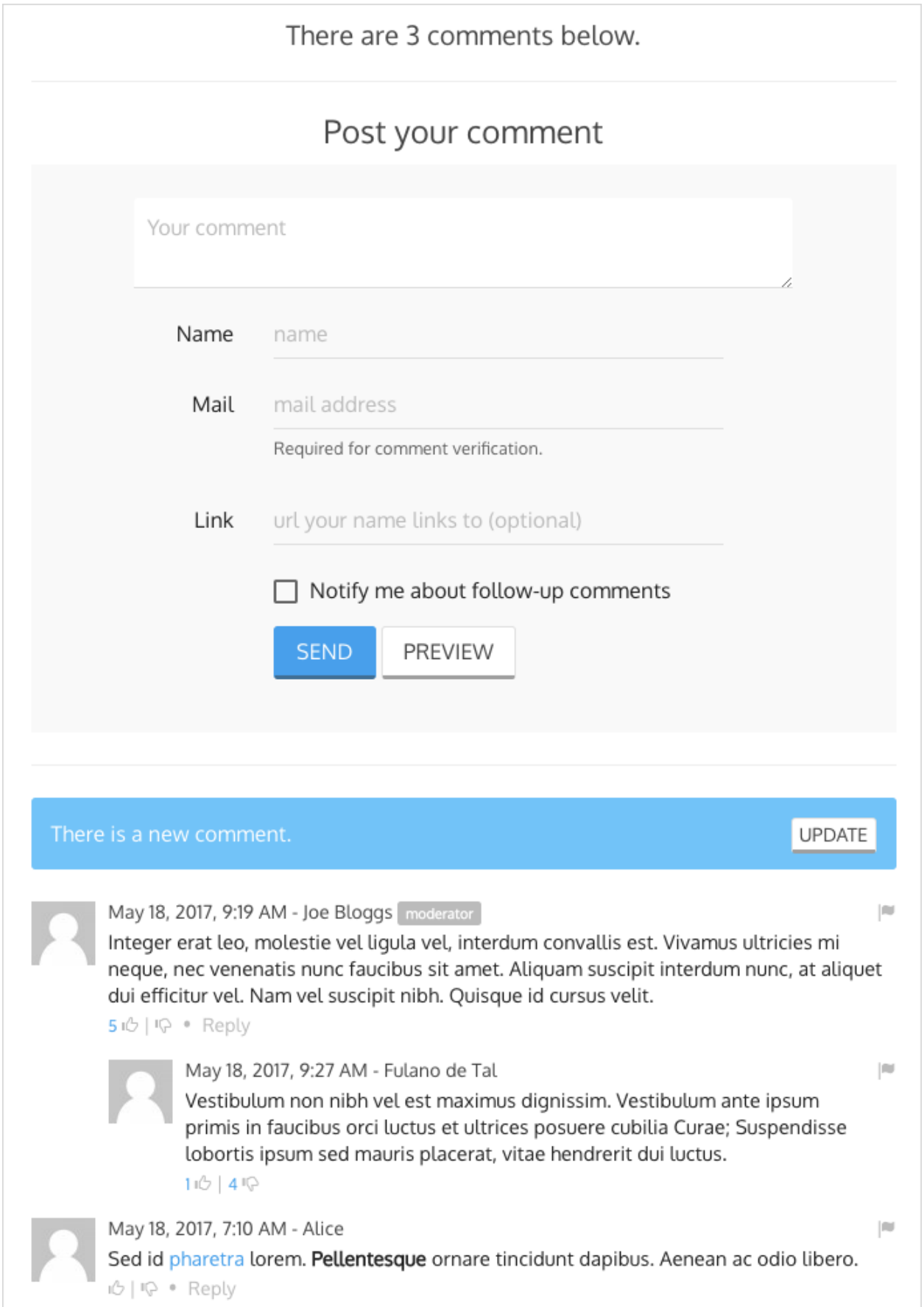
Now let's edit `blog/post_detail.html` and make it look as follows:

```
{% extends "base.html" %}
{% load static %}
{% load comments %}
{% load comments_xtd %}
```

```
{% block title %}{{ object.title }}{% endblock %}

{% block content %}
<h3 class="page-header text-center">{{ object.title }}</h3>
<p class="small text-center">{{ object.publish|date:"l, j F Y" }}</p>
<p>
  {{ object.body|linebreaks }}
</p>

<div class="text-center" style="padding-top:20px">
  <a href="{% url 'blog:post-list' %}">Back to the post list</a>
</div>

<div id="comments"></div>
{% endblock %}

{% block extra-js %}
<script>
  window.comments_props = {% get_commentbox_props for object %};
  window.comments_props_override = {
    allow_comments: {%if object.allow_comments%}true{%else%}false{%endif%},
    allow_feedback: true,
    show_feedback: true,
    allow_flagging: true,
    polling_interval: 5000  // In milliseconds.
  };
</script>
<script src="https://code.jquery.com/jquery-2.2.4.min.js"
        integrity="sha256-BbhdlvQf/xTY9gja0Dq3HiwQF8LaCRTXxZKRutelT44="
        crossorigin="anonymous"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.
↪min.js"
        integrity="sha384-
↪Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa"
        crossorigin="anonymous"></script>
<script type="text/javascript"
        src="{% url 'javascript-catalog' %}"></script>
<script src="{% static 'django_comments_xtd/js/vendor-2.0.3.js' %}"></script>
<script src="{% static 'django_comments_xtd/js/plugin-2.0.3.js' %}"></script>
{% endblock %}
```

The blog post page is now ready to handle comments through the JavaScript plugin, including the following features:

1. Post comments.
2. Preview comments, with instant preview update while typing.
3. Reply comment in the same page, with instant preview while typing.
4. Notifications of new incoming comments using active polling (override *polling_interval* parameter, see the content of first <*script*> tag in the code above).
5. Button to reload the tree of comments, highlighting new comments (see image below).
6. Immediate like/dislike actions.


There are 3 comments below.
Post your comment
Your comment
Name
name
Mail
mail address
Required for comment verification.
Link
url your name links to (optional)
Notify me about follow-up comments
SEND
PREVIEW
There is a new comment.
UPDATE
May 18, 2017, 9:19 AM - Joe Bloggs
moderator
Integer erat leo, molestie vel ligula vel, interdum convallis est. Vivamus ultricies mi neque, nec venenatis nunc faucibus sit amet. Aliquam suscipit interdum nunc, at aliquet dui efficitur vel. Nam vel suscipit nibh. Quisque id cursus velit.
5 | • Reply
May 18, 2017, 9:27 AM - Fulano de Tal
Vestibulum non nibh vel est maximus dignissim. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Suspendisse lobortis ipsum sed mauris placerat, vitae hendrerit dui luctus.
1 | 4
May 18, 2017, 7:10 AM - Alice
Sed id pharetra lorem. Pellentesque ornare tincidunt dapibus. Aenean ac odio libero.
| • Reply

Final notes

We have reached the end of the tutorial. I hope you got enough to start using django-comments-xtd in your own project.

The following page introduces the **Demo projects**. The **simple** demo is a straightforward backend handled project that uses comment confirmation by mail, with follow-up notifications and mute links. The **custom** demo is an example about how to extend django-comments-xtd **Comment** model with new attributes. The **comp** demo shows a project using the complete set of features provided by both django-contrib-comments and django-comments-xtd.

Checkout the **Control Logic** page to understand how django-comments-xtd works along with django-contrib-comments. The **Web API** page details the API provided. The **JavaScript Plugin** covers every aspect regarding the frontend code. Read on **Filters and Template Tags** to see in detail the list of template tags and filters offered. The page on **Customizing django-comments-xtd** goes through the steps to extend the app with a quick example and little prose. Read the **Settings** page and the **Templates** page to get to know how you can customize the default behaviour and default look and feel.

If you want to help, please, report any bug or enhancement directly to the github page of the project. Your contributions are welcome.

Demo projects

There are three example projects available within django-comments-xtd:

1. **simple**: Provides non-threaded comment support to articles. It's an only-backend project, meant as a test case of the basic features (confirmation by mail, follow-up notifications, mute link).
2. **custom**: Provides threaded comment support to articles using a new Comment class that inherits from django-comments-xtd's. The new comment model adds a **title** field to the **XtdComment** class. Find more details in *Customizing django-comments-xtd*.
3. **comp**: This example project provides comment support to several models, defining the maximum thread level on per app.model basis. It uses moderation, removal suggestion flag, like/dislike flags, and list of users who liked/disliked comments. Comment support for Articles are frontend based while comments for Quotes are backend based.

Visit the **example** directory within the repository in GitHub for a quick look.

Table of Contents

Setup

The recommended way to run the demo sites is in its own virtualenv. Once in a new virtualenv, clone the code and cd into any of the 3 demo sites. Then run the migrate command and load the data in the fixtures directory:

```
$ virtualenv venv
$ source venv/bin/activate
(venv)$ git clone git://github.com/danirus/django-comments-xtd.git
(venv)$ cd django-comments-xtd/example/[simple|custom|comp]
(venv)$ pip install django-markdown2
(venv)$ python manage.py migrate
(venv)$ python manage.py loaddata ../fixtures/auth.json
(venv)$ python manage.py loaddata ../fixtures/sites.json
(venv)$ python manage.py loaddata ../fixtures/articles.json
(venv)$ # The **comp** example project needs quotes.json too:
(venv)$ python manage.py loaddata ../fixtures/quotes.json
(venv)$ python manage.py runserver
```

Example projects make use of the package django-markdown2, which in turn depends on Markdown2, to render comments using Markdown syntax.

Fixtures provide:

- A User admin, with password admin.
- A default Site with domain localhost:8000 so that comment confirmation URLs are ready to hit the Django development web server.
- A couple of article objects to which the user can post comments.

By default mails are sent directly to the console using the `console.EmailBackend`. Comment out `EMAIL_BACKEND` in the settings module to send actual mails. You will need to provide working values for all `EMAIL_*` settings.

Simple project

The simple example project features:

1. An Articles App, with a model `Article` whose instances accept comments.
2. Confirmation by mail is required before the comment hit the database, unless `COMMENTS_XTD_CONFIRM_EMAIL` is set to False. Authenticated users don't have to confirm comments.
3. Follow up notifications via mail.
4. Mute links to allow cancellation of follow-up notifications.
5. No nested comments.

This example project tests the initial features provided by django-comments-xtd. Setup the project as explained above.

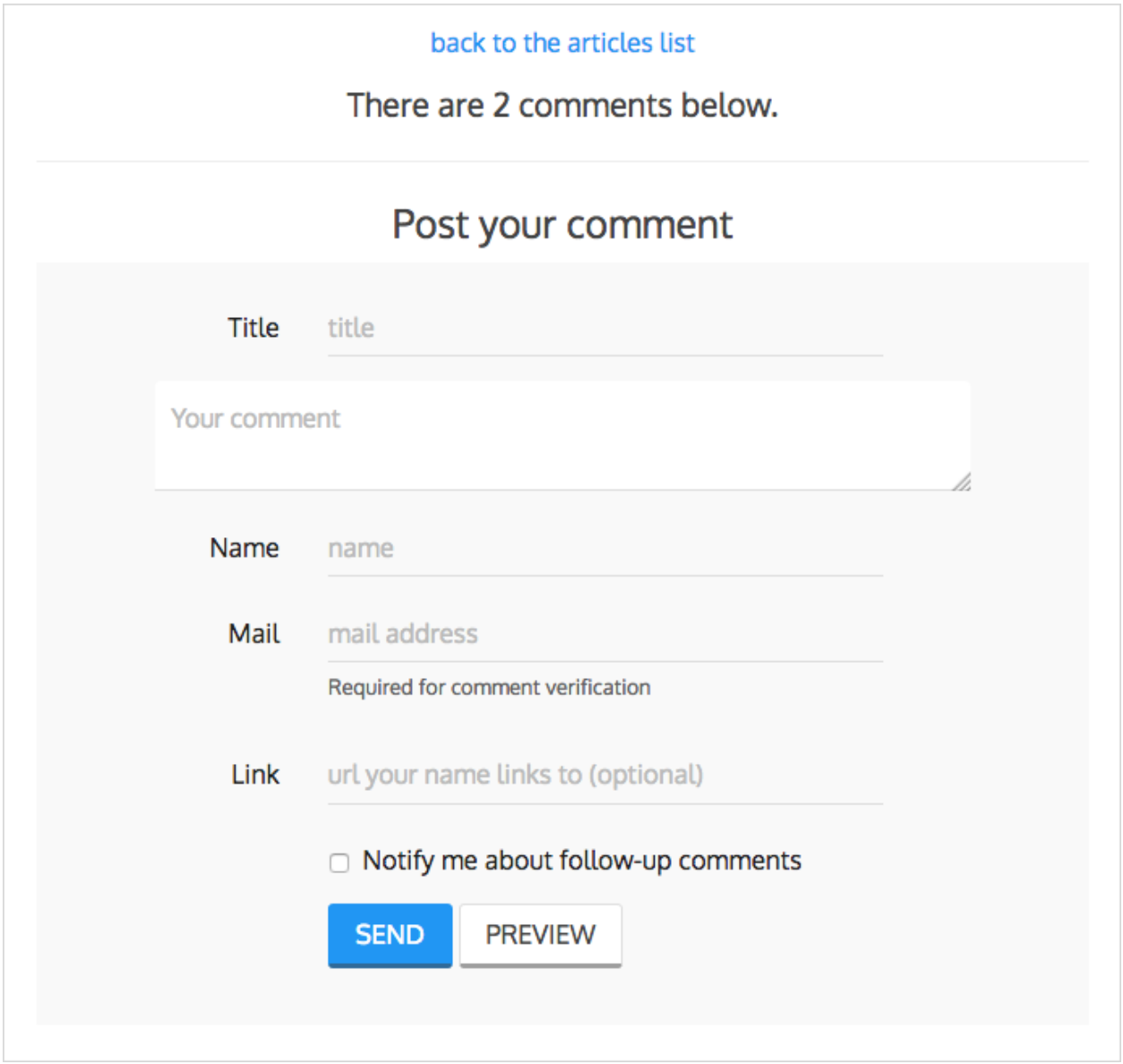
Some hints:

- Log out from the admin site to post comments, otherwise they will be automatically confirmed and no email will be sent.
- When adding new articles in the admin interface be sure to tick the box *allow comments*, otherwise comments won't be allowed.
- Send new comments with the Follow-up box ticked and a different email address. You won't receive follow-up notifications for comments posted from the same email address the new comment is being confirmed from.
- Click on the Mute link on the Follow-up notification email and send another comment. You will not receive further notifications.

Custom project

The **custom** example project extends the **simple** project functionality featuring:

- Thread support up to level 2
- A new comment class that inherits from **XtdComment** with a new **Title** field and a new form class.



Comp project

The Comp Demo implements two apps, each of which contains a model whose instances can received comments:

- App **articles** with the model **Article**
- App **quotes** with the model **Quote**

Features:

1. Comments can be nested, and the maximum thread level is established to 2.
2. Comment confirmation via mail when the users are not authenticated.
3. Comments hit the database only after they have been confirmed.
4. Follow up notifications via mail.
5. Mute links to allow cancellation of follow-up notifications.
6. Registered users can like/dislike comments and can suggest comments removal.
7. Registered users can see the list of users that liked/disliked comments.
8. The homepage presents the last 5 comments posted either to the *articles.Article* or the *quotes.Quote* model.

Threaded comments

The setting *COMMENTS_XTD_MAX_THREAD_LEVEL* is set to 2, meaning that comments may be threaded up to 2 levels below the the first level (internally known as level 0):

```
First comment (level 0)
    |-- Comment to "First comment" (level 1)
        |-- Comment to "Comment to First comment" (level 2)
```

render_xtdcomment_tree

By using the *render_xtdcomment_tree* templatetag, both, *article_detail.html* and *quote_detail.html*, show the tree of comments posted. *article_detail.html* makes use of the arguments *allow_feedback*, *show_feedback* and *allow_flagging*, while *quote_detail.html* only show the list of comments, with no extra arguments, so users can't flag comments for removal, and neither can submit like/dislike feedback.

render_last_xtdcomments

The **Last 5 Comments** shown in the block at the rigght uses the templatetag *render_last_xtdcomments* to show the last 5 comments posted to either *articles.Article* or *quotes.Quote* instances. The templatetag receives the list of pairs *app.model* from which we want to gather comments and shows the given N last instances posted. The templatetag renders the template *django_comments_xtd/comment.html* for each comment retrieve.

Control logic

Following is the application control logic described in 4 actions:

1. The user visits a page that accepts comments. Your app or a 3rd. party app handles the request:

1. Your template shows content that accepts comments. It loads the `comments` templatetag and using tags as `render_comment_list` and `render_comment_form` the template shows the current list of comments and the *post your comment* form.

2. The user **clicks on preview**. Django Comments Framework `post_comment` view handles the request:

1. Renders `comments/preview.html` either with the comment preview or with form errors if any.

3. The user **clicks on post**. Django Comments Framework `post_comment` view handles the request:

    1. If there were form errors it does the same as in point 2.

2. Otherwise creates an instance of `TmpXtdComment` model: an in-memory representation of the comment.
3. Send signal `comment_will_be_posted` and `comment_was_posted`. The *django-comments-xtd* receiver `on_comment_was_posted` receives the second signal with the `TmpXtdComment` instance and does as follows:
   - If the user is authenticated or confirmation by email is not required (see *Settings*):
   - An instance of `XtdComment` hits the database.
   - An email notification is sent to previous comments followers telling them about the new comment following up theirs. Comment followers are those who ticked the box *Notify me about follow up comments via email*.
   - Otherwise a confirmation email is sent to the user with a link to confirm the comment. The link contains a secured token with the `TmpXtdComment`. See below *Creating the secure token for the confirmation URL*.
4. Pass control to the `next` parameter handler if any, or render the `comments/posted.html` template:

- If the instance of `XtdComment` has already been created, redirect to the the comments's absolute URL.
- Otherwise the template content should inform the user about the confirmation request sent by email.

4. The user **clicks on the confirmation link**, in the email message. *Django-comments-xtd* `confirm` view handles the request:

1. Checks the secured token in the URL. If it's wrong returns a 404 code.
2. Otherwise checks whether the comment was already confirmed, in such a case returns a 404 code.
3. Otherwise sends a `confirmation_received` signal. You can register a receiver to this signal to do some extra process before approving the comment. See *Signal and receiver*. If any receiver returns False the comment will be rejected and the template `django_comments_xtd/discarded.html` will be rendered.
4. Otherwise an instance of `XtdComment` finally hits the database, and
5. An email notification is sent to previous comments followers telling them about the new comment following up theirs.

Creating the secure token for the confirmation URL

The Confirmation URL sent by email to the user has a secured token with the comment. To create the token Django-comments-xtd uses the module `signed.py` authored by Simon Willison and provided in Django-OpenID.

`django_openid.signed` offers two high level functions:

- **dumps**: Returns URL-safe, sha1 signed base64 compressed pickle of a given object.
- **loads**: Reverse of dumps(), raises ValueError if signature fails.

A brief example:

```
>>> signed.dumps("hello")
'UydoZWxsbycKcDAKLg.QLtjWHYe7udYuZeQyLlafPqAx1E'

>>> signed.loads('UydoZWxsbycKcDAKLg.QLtjWHYe7udYuZeQyLlafPqAx1E')
'hello'
```

```
>>> signed.loads('UydoZWxsbycKcDAKLg.QLtjWHYe7udYuZeQyLlafPqAx1E-modified')
BadSignature: Signature failed: QLtjWHYe7udYuZeQyLlafPqAx1E-modified
```

There are two components in dump's output `UydoZWxsbycKcDAKLg.QLtjWHYe7udYuZeQyLlafPqAx1E`, separatad by a '.'. The first component is a URLsafe base64 encoded pickle of the object passed to dumps(). The second component is a base64 encoded hmac/SHA1 hash of "$first_component.$secret".

Calling signed.loads(s) checks the signature BEFORE unpickling the object -this protects against malformed pickle attacks. If the signature fails, a ValueError subclass is raised (actually a BadSignature).

Signal and receiver

In addition to the signals sent by the Django Comments Framework, django-comments-xtd sends the following signal:

- **confirmation_received**: Sent when the user clicks on the confirmation link and before the `XtdComment` instance is created in the database.
- **comment_thread_muted**: Sent when the user clicks on the mute link, in a follow-up notification.

Sample use of the `confirmation_received` signal

You might want to register a receiver for `confirmation_received`. An example function receiver could check the time stamp in which a user submitted a comment and the time stamp in which the confirmation URL has been clicked. If the difference between them is over 7 days we will discard the message with a graceful *"sorry, it's a too old comment"* template.

Extending the demo site with the following code will do the job:

```
#----------------------------------------
# append the below code to demos/simple/views.py:

from datetime import datetime, timedelta
from django_comments_xtd import signals

def check_submit_date_is_within_last_7days(sender, data, request, **kwargs):
    plus7days = timedelta(days=7)
        if data["submit_date"] + plus7days < datetime.now():
            return False
    signals.confirmation_received.connect(check_submit_date_is_within_last_
↪7days)


#------------------------------------------------------
# change get_comment_create_data in django_comments_xtd/forms.py to cheat a
# bit and make Django believe that the comment was submitted 7 days ago:

def get_comment_create_data(self):
        from datetime import timedelta                                    #␣
↪ADD THIS

    data = super(CommentForm, self).get_comment_create_data()
    data['followup'] = self.cleaned_data['followup']
    if settings.COMMENTS_XTD_CONFIRM_EMAIL:
        # comment must be verified before getting approved
        data['is_public'] = False
```

```
        data['submit_date'] = datetime.datetime.now() - timedelta(days=8)   #␣
↪ADD THIS
    return data
```

Try the simple demo site again and see that the *django_comments_xtd/discarded.html* template is rendered after clicking on the confirmation URL.

Maximum Thread Level

Nested comments are disabled by default, to enable them use the following settings:

- `COMMENTS_XTD_MAX_THREAD_LEVEL`: an integer value
- `COMMENTS_XTD_MAX_THREAD_LEVEL_BY_APP_MODEL`: a dictionary

Django-comments-xtd inherits the flexibility of django-contrib-comments framework, so that developers can plug it to support comments on as many models as they want in their projects. It is as suitable for one model based project, like comments posted to stories in a simple blog, as for a project with multiple applications and models.

The configuration of the maximum thread level on a simple project is done by declaring the `COMMENTS_XTD_MAX_THREAD_LEVEL` in the `settings.py` file:

```
COMMENTS_XTD_MAX_THREAD_LEVEL = 2
```

Comments then could be nested up to level 2:

```
<In an instance detail page that allows comments>

First comment (level 0)
  |-- Comment to First comment (level 1)
    |-- Comment to Comment to First comment (level 2)
```

Comments posted to instances of every model in the project will allow up to level 2 of threading.

On a project that allows users posting comments to instances of different models, the developer may want to declare a maximum thread level on a per `app.model` basis. For example, on an imaginary blog project with stories, quotes, diary entries and book/movie reviews, the developer might want to define a default, project wide, maximum thread level of 1 for any model and an specific maximum level of 5 for stories and quotes:

```
COMMENTS_XTD_MAX_THREAD_LEVEL = 1
COMMENTS_XTD_MAX_THREAD_LEVEL_BY_APP_MODEL = {
    'blog.story': 5,
    'blog.quote': 5,
}
```

So that `blog.review` and `blog.diaryentry` instances would support comments nested up to level 1, while `blog.story` and `blog.quote` instances would allow comments nested up to level 5.

Web API

django-comments-xtd uses django-rest-framework to expose a Web API that provides developers with access to the same functionalities offered through the web user interface. The Web API has been designed to cover the needs required by the *JavaScript plugin*, and it's open to grow in the future to cover additional functionalities.

There are 5 methods available to perform the following actions:

1. Post a new comment.
2. Retrieve the list of comments posted to a given content type and object ID.
3. Retrieve the number of comments posted to a given content type and object ID.
4. Post user's like/dislike feedback.
5. Post user's removal suggestions.

Table of Contents

Post a new comment

URL name: **comments-xtd-api-create**
Mount point: **<comments-mount-point>/api/comment/**
HTTP Methods: POST
HTTP Responses: 201, 202, 204, 403
Serializer: `django_comments_xtd.api.serializers.WriteCommentSerializer`

This method expects the same fields submitted in a regular django-comments-xtd form. The serializer uses the function `django_comments.get_form` to verify data validity.

Meaning of the HTTP Response codes:

- **201**: Comment created.
- **202**: Comment in moderation.
- **204**: Comment confirmation has been sent by mail.
- **403**: Comment rejected, as in *Disallow black listed domains*.

Retrieve comment list

URL name: **comments-xtd-api-list**
Mount point: **<comments-mount-point>/api/<content-type>/<object-pk>/**
<content-type> is a hyphen separated lowecase pair app_label-model
<object-pk> is an integer representing the object ID.
HTTP Methods: GET
HTTP Responses: 200
Serializer: `django_comments_xtd.api.serializers.ReadCommentSerializer`

This method retrieves the list of comments posted to a given content type and object ID:

```
$ http http://localhost:8000/comments/api/blog-post/4/

HTTP/1.0 200 OK
Allow: GET, HEAD, OPTIONS
Content-Length: 2707
Content-Type: application/json
Date: Tue, 23 May 2017 11:59:09 GMT
Server: WSGIServer/0.2 CPython/3.6.0
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

[
    {
        "allow_reply": true,
        "comment": "Integer erat leo, ...",
        "flags": {
            "dislike": {
                "active": false,
                "users": []
            },
            "like": {
                "active": false,
                "users": [
                    "1:admin",
                    "5:alice",
                    "2:fulanito",
                    "4:joebloggs",
                    "3:menganito"
                ]
            },
            "removal": {
                "active": false,
                "count": null
            }
        },
        "id": 10,
        "is_removed": false,
        "level": 0,
        "parent_id": 10,
        "permalink": "/comments/cr/8/4/#c10",
        "submit_date": "May 18, 2017, 9:19 AM",
        "user_avatar": "http://www.gravatar.com/avatar/7dad9576 ...",
        "user_moderator": true,
        "user_name": "Joe Bloggs",
        "user_url": ""
    },
    {
        ...
    }
]
```

Retrieve comments count

URL name: **comments-xtd-api-count**

Mount point: **<comments-mount-point>/api/<content-type>/<object-pk>/count/**

<content-type> is a hyphen separated lowecase pair app_label-model

<object-pk> is an integer representing the object ID.

HTTP Methods: GET

HTTP Responses: 200

Serializer: `django_comments_xtd.api.serializers.ReadCommentSerializer`

This method retrieves the number of comments posted to a given content type and object ID:

```
$ http http://localhost:8000/comments/api/blog-post/4/count/

HTTP/1.0 200 OK
Allow: GET, HEAD, OPTIONS
Content-Length: 11
Content-Type: application/json
Date: Tue, 23 May 2017 12:06:38 GMT
Server: WSGIServer/0.2 CPython/3.6.0
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

{
    "count": 4
}
```

Post like/dislike feedback

URL name: **comments-xtd-api-feedback**

Mount point: **<comments-mount-point>/api/feedback/**

HTTP Methods: POST

HTTP Responses: 201, 204, 403

Serializer: `django_comments_xtd.api.serializers.FlagSerializer`

This method toggles flags like/dislike for a comment. Successive calls set/unset the like/dislike flag:

```
$ http -a admin:admin POST http://localhost:8000/comments/api/feedback/␣
↪comment=10 flag="like"

HTTP/1.0 201 Created
Allow: POST, OPTIONS
Content-Length: 34
Content-Type: application/json
Date: Tue, 23 May 2017 12:27:00 GMT
Server: WSGIServer/0.2 CPython/3.6.0
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

{
    "comment": 10,
    "flag": "I liked it"
}
```

Calling it again unsets the *"I liked it"* flag:

```
$ http -a admin:admin POST http://localhost:8000/comments/api/feedback/␣
↪comment=10 flag="like"

HTTP/1.0 204 No Content
Allow: POST, OPTIONS
```

```
Content-Length: 0
Date: Tue, 23 May 2017 12:26:56 GMT
Server: WSGIServer/0.2 CPython/3.6.0
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN
```

It requires the user to be logged in:

```
$ http POST http://localhost:8000/comments/api/feedback/ comment=10 flag=
↪"like"

HTTP/1.0 403 Forbidden
Allow: POST, OPTIONS
Content-Length: 58
Content-Type: application/json
Date: Tue, 23 May 2017 12:27:31 GMT
Server: WSGIServer/0.2 CPython/3.6.0
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

{
    "detail": "Authentication credentials were not provided."
}
```

Post removal suggestions

URL name: **comments-xtd-api-flag**
Mount point: **<comments-mount-point>/api/flag/**
HTTP Methods: POST
HTTP Responses: 201, 403
Serializer: `django_comments_xtd.api.serializers.FlagSerializer`

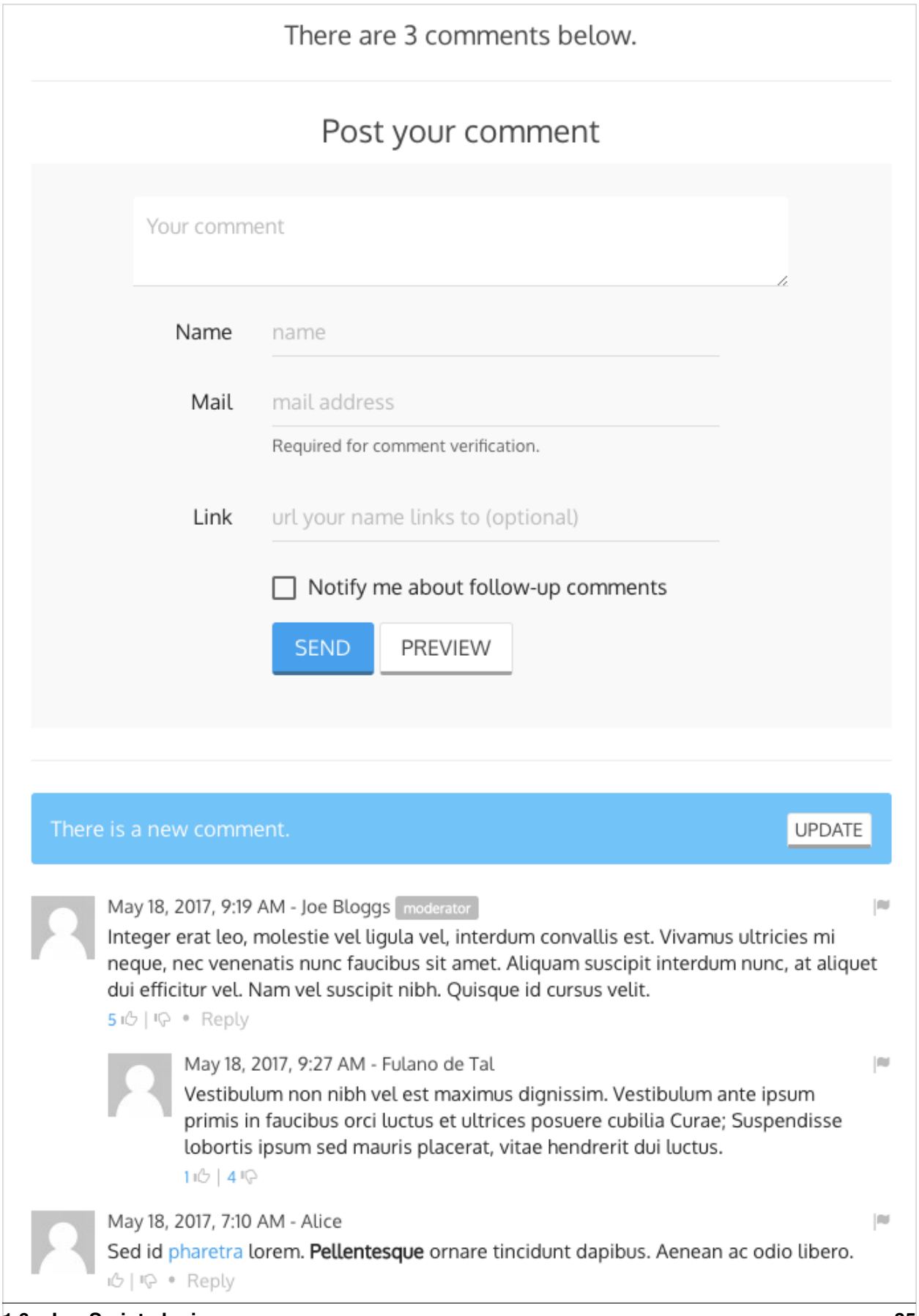
This method sets the *removal suggestion* flag on a comment. Once created for a given user successive calls return 201 but the flag object is not created again.

```
$ http POST http://localhost:8000/comments/api/flag/ comment=10 flag="report"

HTTP/1.0 201 Created
Allow: POST, OPTIONS
Content-Length: 42
Content-Type: application/json
Date: Tue, 23 May 2017 12:35:02 GMT
Server: WSGIServer/0.2 CPython/3.6.0
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

{
    "comment": 10,
    "flag": "removal suggestion"
}
```

As the previous method, it requires the user to be logged in.

JavaScript plugin

As of version 2.0 django-comments-xtd comes with a JavaScript plugin that enables comment support as in a Single Page Application fashion. Comments are loaded and sent in the background, as long as like/dislike opinions. There is an active verification, based on polling, that checks whether there are new incoming comments to show to the user, and an update button that allows the user to refresh the tree, highlighting new comments with a green label to indicate recently received comment entries.


There are 3 comments below.
Post your comment
Your comment
Name
name
Mail
mail address
Required for comment verification.
Link
url your name links to (optional)
Notify me about follow-up comments
SEND
PREVIEW
There is a new comment.
UPDATE
May 18, 2017, 9:19 AM - Joe Bloggs
moderator
Integer erat leo, molestie vel ligula vel, interdum convallis est. Vivamus ultricies mi neque, nec venenatis nunc faucibus sit amet. Aliquam suscipit interdum nunc, at aliquet dui efficitur vel. Nam vel suscipit nibh. Quisque id cursus velit.
5 | • Reply
May 18, 2017, 9:27 AM - Fulano de Tal
Vestibulum non nibh vel est maximus dignissim. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Suspendisse lobortis ipsum sed mauris placerat, vitae hendrerit dui luctus.
1 | 4
May 18, 2017, 7:10 AM - Alice
Sed id pharetra lorem. Pellentesque ornare tincidunt dapibus. Aenean ac odio libero.
| • Reply

This plugin is done by making choices that might not be the same you made in your own projects.

Frontend opinions

Django is a backend framework imposing little opinions regarding the frontend. It merely uses jQuery in the admin site. Nothing more. That leaves developers the choice to pick anything they want for the frontend to go along with the backend.

For backend developers the level of stability found in Python and Django contrasts with the active diversity of JavaScript libraries available for the frontend.

The JavaScript plugin included in the app is a mix of frontend decisions with the goal to provide a quick and full frontend solution. Doing so the app is ready to be plugged in a large number of backend projects, and in a reduced set of frontend stacks.

The JavaScript Plugin is based on:

- ReactJS
- jQuery (merely for Ajax)
- Remarkable (for Markdown markup support)
- Twitter-bootstrap (for the UI and the tooltip utility)

The build process is based on Webpack2 instead of any other as good a tool available in the JavaScript building tools landscape.

The decision of building a plugin based on these choices doesn't mean there can't be other ones. The project is open to improve its own range of JavaScript plugins through contributions. If you feel like improving the current plugin or providing additional ones, please, consider to integrate it using Webpack2 and try to keep the source code tree as clean and structured as possible.

Build process

In order to further develop the current plugin, fix potential bugs or install the the plugin from the sources, you have to use NodeJS and NPM.

Set up the backend

Before installing the frontend dependencies we will prepare a Python virtualenv in which we will have all the backend dependencies installed. Let's start by creating the virtualenv and fetching the sources:

```
$ virtualenv ~/venv/django-comments-xtd
$ source ~/venv/django-comments-xtd/bin/activate
(django-comments-xtd)$ cd ~/src/  # or cd into your sources dir of choice.
(django-comments-xtd)$ git clone https://github.com/danirus/django-comments-
↪xtd.git
(django-comments-xtd)$ cd django-comments-xtd
(django-comments-xtd)$ python setup.py develop
```

Check whether the app passes the battery of tests:

```
(django-comments-xtd)$ python setup.py test
```

As the sample Django project you can use the **comp** example site. Install first the django-markdown2 package (required by the comp example project) and setup the project:

```
(django-comments-xtd)$ cd example/comp
(django-comments-xtd)$ pip install django-markdown2
(django-comments-xtd)$ python manage.py migrate
(django-comments-xtd)$ python manage.py loaddata ../fixtures/auth.json
(django-comments-xtd)$ python manage.py loaddata ../fixtures/sites.json
(django-comments-xtd)$ python manage.py loaddata ../fixtures/articles.json
(django-comments-xtd)$ python manage.py runserver
```

Now the project is ready and the plugin will load from the existing bundle files. Check it out by visiting an article's page and sending some comments. No frontend source package has been installed so far.

Install frontend packages

At this point open another terminal and cd into django-comments-xtd source directory again, then install all the frontend dependencies:

```
$ cd ~/src/django-comments-xtd
$ npm install
```

It will install all the dependencies listed in the **package.json** file in the local *node_modules* directory. Once it's finished run webpack to build the bundles and watch for changes in the source tree:

```
$ webpack --watch
```

Webpack will put the bundles in the static directory of django-comments-xtd and Django will fetch them from there when rendering the article's detail page:

```
{% block extra-js %}
[...]
<script src="{% static 'django_comments_xtd/js/vendor-2.0.3.js' %}"></script>
<script src="{% static 'django_comments_xtd/js/plugin-2.0.3.js' %}"></script>
{% endblock extra-js %}
```

Code structure

Plugin sources live inside the **static** directory of django-comments-xtd:

```
$ cd ~/src/django-comments-xtd
$ tree django_comments_xtd/static/django_comments_xtd/js

django_comments_xtd/static/django_comments_xtd/js
- src
|   - comment.jsx
|   - commentbox.jsx
|   - commentform.jsx
|   - index.js
|   - lib.js
- plugin-2.0.3.js
- vendor-2.0.3.js

1 directory, 7 files
```

The intial development was inspired by the ReactJS Comment Box tutorial. Component names reflect those of the ReactJS tutorial.

The application entry point is located inside the `index.js` file. The `props` passed to the **CommentBox** object are those declared in the `var window.comments_props` defined in the django template:

```
<script>
  window.comments_props = {% get_commentbox_props for object %};
  window.comments_props_override = {
    allow_comments: {%if object.allow_comments%}true{%else%}false{%endif%},
    allow_feedback: true,
    show_feedback: true,
    allow_flagging: true,
    poll_interval: 2000,
  };
</script>
```

And are overriden by those declared in the `var window.comments_props_override`.

Improvements and contributions

The current ReactJS plugin could be ported to an Inferno plugin within a reasonable timeframe. Inferno offers a lighter footprint compared to ReactJS plus it is among the faster JavaScript frontend frameworks.

Another improvement pending for implementation would be a websocket based update. At the moment comment updates are received by active polling. See `commentbox.jsx`, method **load_count** of the **CommentBox** component.

Contributions are welcome, write me an email at mbox@danir.us or open an issue in the GitHub repository.

Filters and template tags

Django-comments-xtd provides 5 template tags and 3 filters. Load the module to make use of them in your templates:

```
{% load comments_xtd %}
```

Table of Contents

Tag `render_xtdcomment_tree`

Tag syntax:

```
{% render_xtdcomment_tree [for <object>] [with var_name_1=<obj_1> var_name_2=
↪<obj_2>]
                          [allow_flagging] [allow_feedback] [show_feedback]
                          [using <template>] %}
```

Renders the threaded structure of comments posted to the given object using the first template found from the list:

- `django_comments_xtd/<app>/<model>/comment_tree.html`
- `django_comments_xtd/<app>/comment_tree.html`
- `django_comments_xtd/comment_tree.html` (provided with the app)

It expects either an object specified with the `for <object>` argument, or a variable named `comments`, which might be present in the context or received as `comments=<comments-object>`. When the `for <object>` argument is specified, it retrieves all the comments posted to the given object, ordered by the `thread_id` and `order` within the thread, as stated by the setting *COMMENTS_XTD_LIST_ORDER*.

It supports 4 optional arguments:

- `allow_flagging`, enables the comment removal suggestion flag. Clicking on the removal suggestion flag redirects to the login view whenever the user is not authenticated.
- `allow_feedback`, enables the like and dislike flags. Clicking on any of them redirects to the login view whenever the user is not authenticated.
- `show_feedback`, shows two list of users, of those who like the comment and of those who don't like it. By overriding `includes/django_comments_xtd/user_feedback.html` you could show the lists only to authenticated users.
- `using <template_path>`, makes the templatetag use a different template, instead of the default one, `django_comments_xtd/comment_tree.html`

Example usage

In the usual scenario the tag is used in the object detail template, i.e.: `blog/article_detail.html`, to include all comments posted to the article, in a tree structure:

```
{% render_xtdcomment_tree for article allow_flagging allow_feedback show_
↪feedback  %}
```

Tag `get_xtdcomment_tree`

Tag syntax:

```
{% get_xtdcomment_tree for [object] as [varname] [with_feedback] %}
```

Returns a dictionary to the template context under the name given in `[varname]` with the comments posted to the given `[object]`. The dictionary has the form:

```
{
    'comment': xtdcomment_object,
    'children': [ list_of_child_xtdcomment_dicts ]
}
```

The comments will be ordered by the `thread_id` and `order` within the thread, as stated by the setting *COMMENTS_XTD_LIST_ORDER*.

When the optional argument `with_feedback` is specified the returned dictionary will contain two additional attributes with the list of users who liked the comment and the list of users who disliked it:

```
{
    'xtdcomment': xtdcomment_object,
    'children': [ list_of_child_xtdcomment_dicts ],
    'likedit': [user_a, user_b, ...],
    'dislikedit': [user_n, user_m, ...]
}
```

Example usage

Get an ordered dictionary with the comments posted to a given blog story and store the dictionary in a template context variabled called `comment_tree`:

```
{% get_xtdcomment_tree for story as comments_tree with_feedback %}
```

Tag `render_last_xtdcomments`

Tag syntax:

```
{% render_last_xtdcomments [N] for [app].[model] [[app].[model] ...] %}
```

Renders the list of the last N comments for the given pairs `<app>.<model>` using the following search list for templates:

- `django_comments_xtd/<app>/<model>/comment.html`
- `django_comments_xtd/<app>/comment.html`
- `django_comments_xtd/comment.html`

Example usage

Render the list of the last 5 comments posted, either to the blog.story model or to the blog.quote model. See it in action in the *Multiple Demo Site*, in the *blog homepage*, template `blog/homepage.html`:

```
{% render_last_xtdcomments 5 for blog.story blog.quote %}
```

Tag `get_last_xtdcomments`

Tag syntax:

```
{% get_last_xtdcomments [N] as [varname] for [app].[model] [[app].[model] ...] %}
```

Gets the list of the last N comments for the given pairs `<app>.<model>` and stores it in the template context whose name is defined by the `as` clause.

Example usage

Get the list of the last 10 comments two models, `Story` and `Quote`, have received and store them in the context variable `last_10_comment`. You can then loop over the list with a `for` tag:

```
{% get_last_xtdcomments 10 as last_10_comments for blog.story blog.quote %}
{% if last_10_comments %}
  {% for comment in last_10_comments %}
    <p>{{ comment.comment|linebreaks }}</p> ...
  {% endfor %}
{% else %}
  <p>No comments</p>
{% endif %}
```

Tag `get_xtdcomment_count`

Tag syntax:

```
{% get_xtdcomment_count as [varname] for [app].[model] [[app].[model] ...] %}
```

Gets the comment count for the given pairs `<app>.<model>` and populates the template context with a variable containing that value, whose name is defined by the `as` clause.

Example usage

Get the count of comments the model `Story` of the app `blog` have received, and store it in the context variable `comment_count`:

```
{% get_xtdcomment_count as comment_count for blog.story %}
```

Get the count of comments two models, `Story` and `Quote`, have received and store it in the context variable `comment_count`:

```
{% get_xtdcomment_count as comment_count for blog.story blog.quote %}
```

Filter `xtd_comment_gravatar`

Filter syntax:

```
{{ comment.email|xtd_comment_gravatar }}
```

A simple gravatar filter that inserts the gravatar image associated to an email address.

This filter has been named `xtd_comment_gravatar` as oposed to simply `gravatar` to avoid potential name collisions with other gravatar filters the user might have opted to include in the template.

Filter `xtd_comment_gravatar_url`

Filter syntax:

```
{{ comment.email|xtd_comment_gravatar_url }}
```

A simple gravatar filter that inserts the gravatar URL associated to an email address.

This filter has been named `xtd_comment_gravatar_url` as oposed to simply `gravatar_url` to avoid potential name collisions with other gravatar filters the user might have opted to include in the template.

Filter `render_markup_comment`

Filter syntax:

```
{{ comment.comment|render_markup_comment }}
```

Renders a comment using a markup language specified in the first line of the comment. It uses django-markup to parse the comments with a markup language parser and produce the corresponding output.

Example usage

A comment posted with a content like:

```
#!markdown
An [example](http://url.com/ "Title")
```

Would be rendered as a markdown text, producing the output:

```
<p><a href="http://url.com/" title="Title">example</a></p>
```

Available markup languages are:

- Markdown, when starting the comment with `#!markdown`.
- reStructuredText, when starting the comment with `#!restructuredtext`.
- Linebreaks, when starting the comment with `#!linebreaks`.

Migrating to django-comments-xtd

If your project uses django-contrib-comments you can easily plug django-comments-xtd to add extra functionalities like comment confirmation by mail, comment threading and follow-up notifications.

This section describes how to make django-comments-xtd take over comments support in a project in which django-contrib-comments tables have received data already.

Preparation

First of all, install django-comments-xtd:

```
(venv)$ cd mysite
(venv)$ pip install django-comments-xtd
```

Then edit the settings module and change your `INSTALLED_APPS` so that django_comments_xtd and django_comments are listed in this order. Also change the `COMMENTS_APP` and add the `EMAIL_*` settings to be able to send mail messages:

```
INSTALLED_APPS = [
    ...
    'django_comments_xtd',
    'django_comments',
    ...
]
...
COMMENTS_APP = 'django_comments_xtd'
```

```
# Either enable sending mail messages to the console:
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'

# Or set up the EMAIL_* settings so that Django can send emails:
EMAIL_HOST = "smtp.mail.com"
EMAIL_PORT = "587"
EMAIL_HOST_USER = "alias@mail.com"
EMAIL_HOST_PASSWORD = "yourpassword"
EMAIL_USE_TLS = True
DEFAULT_FROM_EMAIL = "Helpdesk <helpdesk@yourdomain>"
```

Edit the urls module of the project and mount django_comments_xtd's URLs in the path in which you had django_comments' URLs, django_comments_xtd's URLs includes django_comments':

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

urlpatterns = [
    ...
    url(r'^comments/', include('django_comments_xtd.urls')),
    ...
]
```

Now create the tables for django-comments-xtd:

```
(venv)$ python manage.py migrate
```

Populate comment data

The following step will populate **XtdComment**'s table with data from the **Comment** model. For that purpose you can use the `populate_xtdcomments` management command:

```
(venv)$ python manage.py populate_xtdcomments
Added 3468 XtdComment object(s).
```

You can pass as many DB connections as you have defined in `DATABASES` and the command will run in each of the databases, populating the **XtdComment**'s table with data from the comments table existing in each database.

Now the project is ready to handle comments with django-comments-xtd.

Customizing django-comments-xtd

django-comments-xtd can be extended in the same way as django-contrib-comments. There are three points to observe:

1. The setting `COMMENTS_APP` must be `'django_comments_xtd'`.
2. The setting `COMMENTS_XTD_MODEL` must be your model class name, i.e.: `'mycomments.models.MyComment'`.
3. The setting `COMMENTS_XTD_FORM_CLASS` must be your form class name, i.e.: `'mycomments.forms.MyCommentForm'`.

In addition to that, write an `admin.py` module to see the new comment class in the admin interface. Inherit from `django_commensts_xtd.admin.XtdCommentsAdmin`. You might want to add your new comment fields to

the comment list view, by rewriting the `list_display` attribute of your admin class. Or change the details view customizing the `fieldsets` attribute.

Custom Comments Demo

The demo site `custom_comments` available with the source code in GitHub (directory `django_comments_xtd\demos\custom_comments`) implements a sample Django project with comments that extend django_comments_xtd with an additional field, a title.

`settings` Module

The `settings.py` module contains the following customizations:

```
INSTALLED_APPS = (
  # ...
  'django_comments_xtd',
  'django_comments',
  'articles',
  'mycomments',
  # ...
)

COMMENTS_APP = "django_comments_xtd"
COMMENTS_XTD_MODEL = 'mycomments.models.MyComment'
COMMENTS_XTD_FORM_CLASS = 'mycomments.forms.MyCommentForm'
```

`models` Module

The new class `MyComment` extends django_comments_xtd's `XtdComment` with a title field:

```
from django.db import models
from django_comments_xtd.models import XtdComment


class MyComment(XtdComment):
    title = models.CharField(max_length=256)
```

`forms` Module

The forms module extends `XtdCommentForm` and rewrites the method `get_comment_create_data`:

```
from django import forms
from django.utils.translation import ugettext_lazy as _

from django_comments_xtd.forms import XtdCommentForm
from django_comments_xtd.models import TmpXtdComment


class MyCommentForm(XtdCommentForm):
    title = forms.CharField(
        max_length=256,
        widget=forms.TextInput(attrs={'placeholder': _('title')})
    )
```

```
    def get_comment_create_data(self):
        data = super(MyCommentForm, self).get_comment_create_data()
        data.update({'title': self.cleaned_data['title']})
        return data
```

admin Module

The admin module provides a new class MyCommentAdmin that inherits from XtdCommentsAdmin and customize some of its attributes to include the new field `title`:

```
from django.contrib import admin
from django.utils.translation import ugettext_lazy as _

from django_comments_xtd.admin import XtdCommentsAdmin
from custom_comments.mycomments.models import MyComment


class MyCommentAdmin(XtdCommentsAdmin):
    list_display = ('thread_level', 'title', 'cid', 'name', 'content_type',
                    'object_pk', 'submit_date', 'followup', 'is_public',
                    'is_removed')
    list_display_links = ('cid', 'title')
    fieldsets = (
        (None,           {'fields': ('content_type', 'object_pk', 'site')}),
        (_('Content'),  {'fields': ('title', 'user', 'user_name', 'user_email',
                                    'user_url', 'comment', 'followup')}),
        (_('Metadata'), {'fields': ('submit_date', 'ip_address',
                                    'is_public', 'is_removed')}),
    )

admin.site.register(MyComment, MyCommentAdmin)
```

Templates

You will need to customize the following templates:

- `comments/form.html` to include new fields.
- `comments/preview.html` to preview new fields.
- `django_comments_xtd/email_confirmation_request.{txt|html}` to add the new fields to the confirmation request, if it was necessary. This demo overrides them to include the `title` field in the mail.
- `django_comments_xtd/comments_tree.html` to show the new field when displaying the comments. If your project doesn't allow nested comments you can use either this template or *comments/list.html*'.
- `django_comments_xtd/reply.html` to show the new field when displaying the comment the user is replying to.

Internationalization

django-comments-xtd is i18n ready. Please, consider extending support for your language if it's not listed below. At the moment it's available only in:

- English, **en** (default language)
- French, **fr**
- Spanish, **es**

Contributions

This is a step by step guide to help extending the internationalization of django-comments-xtd. Install the **comp** example site. It will be used along with django-rosetta to help with translations.

```
$ virtualenv venv
$ source venv/bin/activate
(venv)$ git clone git://github.com/danirus/django-comments-xtd.git
(venv)$ cd django-comments-xtd/example/comp
(venv)$ pip install django-rosetta django-markdown2
(venv)$ python manage.py migrate
(venv)$ python manage.py loaddata ../fixtures/auth.json
(venv)$ python manage.py loaddata ../fixtures/sites.json
(venv)$ python manage.py loaddata ../fixtures/articles.json
(venv)$ python manage.py loaddata ../fixtures/quotes.json
(venv)$ python manage.py runserver
```

Edit the **comp/settings.py** module. Add the ISO-639-1 code of the language you want to support to `LANGUAGES` and add `'rosetta'` to your `INSTALLED_APPS`.

```
LANGUAGES = (
    ('en', 'English'),
    ('fr', 'French'),
    ('es', 'Spanish'),
    ...
)

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

Note: When django-rosetta is enabled in the **comp** project, the homepage shows a selector to help switch languages. It uses the `language_tuple` filter, located in the **comp_filters.py** module, to show the language name in both, the translated form and the original language.

We have to create the translation catalog for the new language. Use the ISO-639-1 code to indicate the language. There are two catalogs to translate, one for the backend and one for the frontend.

The frontend catalog is produced out of the **plugin-X.Y.Z.js** file. It's a good idea to run the `webpack --watch` command if you change the messages in the sources of the plugin (placed in the **js/src/** directory). This way the plugin is built automatically and the Django `makemessages` command will fetch the new messages accordingly.

Keep the runserver command launched above running in one terminal and open another terminal to run the **makemessages** and **compilemessages** commands:

```
$ source venv/bin/activate
(venv)$ cd django-comments-xtd/django_comments_xtd
```

```
(venv)$ django-admin makemessages -l de
(venv)$ django-admin makemessages -d djangojs -l de
```

Now head to the rosetta page, under http://localhost:8000/rosetta/, do login with user `admin` and password `admin`, and proceed to translate the messages. Find the two catalogs for django-comments-xtd under the **Third Party** filter, at the top-right side of the page.

Django must have the catalogs compiled before the messages show up in the comp site. Run the compile message for that purpose:

```
(venv)$ django-admin compilemessages
```

The **comp** example site is now ready to show the messages in the new language. It's time to verify that the translation fits the UI. If everything looks good, please, make a Pull Request to add the new .po files to the upstream repository.

Settings

To use django-comments-xtd it is necessary to declare the `COMMENTS_APP` setting in your project's settings module as:

```
COMMENTS_APP = "django_comments_xtd"
```

A number of additional settings are available to customize django-comments-xtd behaviour.

Table of Contents

COMMENTS_XTD_MAXIMUM_THREAD_LEVEL

Optional. Indicates the **Maximum thread level** for comments. In other words, whether comments can be nested. This setting established the default value for comments posted to instances of every model instance in Django. It can be overriden on per app.model basis using the `COMMENTS_XTD_MAXIMUM_THREAD_LEVEL_BY_APP_MODEL`, introduced right after this section.

An example:

```
COMMENTS_XTD_MAX_THREAD_LEVEL = 8
```

It defaults to `0`. What means nested comments are not permitted.

COMMENTS_XTD_MAXIMUM_THREAD_LEVEL_BY_APP_MODEL

Optional. The **Maximum thread level on per app.model basis** is a dictionary with pairs *app_label.model* as keys and the maximum thread level for comments posted to instances of those models as values. It allows definition of max comment thread level on a per *app_label.model* basis.

An example:

```
COMMENTS_XTD_MAX_THREAD_LEVEL = 0
COMMENTS_XTD_MAX_THREAD_LEVEL_BY_APP_MODEL = {
    'projects.release': 2,
    'blog.stories': 8, 'blog.quotes': 8,
    'blog.diarydetail': 0 # not required as it defaults to COMMENTS_XTD_MAX_THREAD_
↪LEVEL
}
```

In the example, comments posted to `projects.release` instances can go up to level 2:

```
First comment (level 0)
    |-- Comment to "First comment" (level 1)
        |-- Comment to "Comment to First comment" (level 2)
```

It defaults to `{}`. What means the maximum thread level is setup with *COMMENTS_XTD_MAX_THREAD_LEVEL*.

COMMENTS_XTD_CONFIRM_MAIL

Optional. It specifies the **confirm comment post by mail** setting, establishing whether a comment confirmation should be sent by mail. If set to `True` a confirmation message is sent to the user with a link on which she has to click to confirm the comment. If the user is already authenticated the confirmation is not sent and the comment is accepted, if no moderation has been setup up, with no further confirmation needed.

If is set to False, and no moderation has been set up to potentially discard it, the comment will be accepted.

Read about the *Moderation* topic in the tutorial.

An example:

```
COMMENTS_XTD_CONFIRM_EMAIL = True
```

It defaults to `True`.

COMMENTS_XTD_FROM_EMAIL

Optional. It specifies the **from mail address** setting used in the *from* field when sending emails.

An example:

```
COMMENTS_XTD_FROM_EMAIL = "noreply@yoursite.com"
```

It defaults to `settings.DEFAULT_FROM_EMAIL`.

COMMENTS_XTD_CONTACT_EMAIL

Optional. It specifies a **contact mail address the user could use to get in touch with a helpdesk or support personnel. It's used in both templates, **email_confirmation_request.txt** and **email_confirmation_request.html**, from the **templates/django_comments_xtd** directory.

An example:

```
COMMENTS_XTD_FROM_EMAIL = "helpdesk@yoursite.com"
```

It defaults to `settings.DEFAULT_FROM_EMAIL`.

COMMENTS_XTD_FORM_CLASS

Optional, form class to use when rendering comment forms. It's a string with the class path to the form class that will be used for comments.

An example:

```
COMMENTS_XTD_FORM_CLASS = "mycomments.forms.MyCommentForm"
```

It defaults to *"django_comments_xtd.forms.XtdCommentForm"*.

COMMENTS_XTD_MODEL

Optional, represents the model class to use for comments. It's a string with the class path to the model that will be used for comments.

An example:

```
COMMENTS_XTD_MODEL = "mycomments.models.MyCommentModel"
```

Defaults to *"django_comments_xtd.models.XtdComment"*.

COMMENTS_XTD_LIST_ORDER

Optional, represents the field ordering in which comments are retrieve, a tuple with field names, used by the `get_queryset` method of `XtdComment` model's manager.

It defaults to `('thread_id', 'order')`

COMMENTS_XTD_MARKUP_FALLBACK_FILTER

Optional, default filter to use when rendering comments. Indicates the default markup filter for comments. This value must be a key in the `MARKUP_FILTER` setting. If not specified or None, comments that do not indicate an intended markup filter are simply returned as plain text.

An example:

```
COMMENTS_XTD_MARKUP_FALLBACK_FILTER = 'markdown'
```

It defaults to `None`.

COMMENTS_XTD_SALT

Optional, it is the **extra key to salt the comment form**. It establishes the bytes string extra_key used by `signed.dumps` to salt the comment form hash, so that there an additional secret is in use to encode the comment before sending it for confirmation within a URL.

An example:

```
COMMENTS_XTD_SALT = 'G0h5gt073h6gH4p25GS2g5AQ25hTm256yGt134tMP5TgCX$&HKOYRV'
```

It defaults to an empty string.

COMMENTS_XTD_SEND_HTML_EMAIL

Optional, enable/disable HTML mail messages. This boolean setting establishes whether email messages have to be sent in HTML format. By the default messages are sent in both Text and HTML format. By disabling the setting, mail messages will be sent only in text format.

An example:

```
COMMENTS_XTD_SEND_HTML_EMAIL = False
```

It defaults to True.

COMMENTS_XTD_THREADED_EMAILS

Optional, enable/disable sending mails in separated threads. For low traffic websites sending mails in separate threads is a fine solution. However, for medium to high traffic websites such overhead could be reduced by using other solutions, like a Celery application or any other detached from the request-response HTTP loop.

An example:

```
COMMENTS_XTD_THREADED_EMAILS = False
```

Defaults to `True`.

COMMENTS_XTD_APP_MODEL_OPTIONS

Optional. Allow enabling/disabling commenting options on per **app_label.model** basis. The options available are the following:

- `allow_flagging`: Allow registered users to flag comments as inappropriate.

- `allow_feedback`: Allow registered users to like/dislike comments.
- `show_feedback`: Allow django-comments-xtd to report the list of users who liked/disliked the comment. The representation of each user in the list depends on the next setting :setting::*COMMENTS_XTD_API_USER_REPR*.

An example use:

```
COMMENTS_XTD_APP_MODEL_OPTIONS = {
    'blog.post': {
        'allow_flagging': True,
        'allow_feedback': True,
        'show_feedback': True,
    }
}
```

Defaults to:

```
COMMENTS_XTD_APP_MODEL_OPTIONS = {
    'default': {
        'allow_flagging': False,
        'allow_feedback': False,
        'show_feedback': False,
    }
}
```

COMMENTS_XTD_API_USER_REPR

Optional. Function that receives a user object and returns its string representation. It's used to produced the list of users who liked/disliked comments. By default it outputs the username, but it could perfectly return the full name:

```
COMMENTS_XTD_API_USER_REPR = lambda u: u.get_full_name()
```

Defaults to:

```
COMMENTS_XTD_API_USER_REPR = lambda u: u.username
```

Templates

This page details the list of templates provided by django-comments-xtd. They are located under the `django_comments_xtd/` templates directory.

Table of Contents

- *disliked.html*
- *discarded.html*
- *email_followup_comment*
- *comment.html*
- *posted.html*
- *reply.html*
- *muted.html*

email_confirmation_request

As `.html` and `.txt`, this template represents the confirmation message sent to the user when the **Send** button is clicked to post a comment. Both templates are sent in a multipart message, or only in text format if the *COMMENTS_XTD_SEND_HTML_EMAIL* setting is set to `False`.

In the context of the template the following objects are expected:

- The `site` object (django-contrib-comments, and in turn django-comments-xtd, use the Django Sites Framework).
- The `comment` object.
- The `confirmation_url` the user has to click on to confirm the comment.

comment_tree.html

This template is rendered by the *Tag render_xtdcomment_tree* to represent the comments posted to an object.

In the context of the template the following objects are expected:

- A list of dictionaries called `comments` in which each element is a dictionary like:

```
{
    'comment': xtdcomment_object,
    'children': [ list_of_child_xtdcomment_dicts ]
}
```

Optionally the following objects can be present in the template:

- A boolean `allow_flagging` to indicate whether the user will have the capacity to suggest comment removal.
- A boolean `allow_feedback` to indicate whether the user will have the capacity to like/dislike comments. When `True` the special template `user_feedback.html` will be rendered.

user_feedback.html

This template is expected to be in the directory `includes/django_comments_xtd/`, and it provides a way to customized the look of the like and dislike buttons as long as the list of users who clicked on them. It is included from `comment_tree.html`. The template is rendered only when the *Tag render_xtdcomment_tree* is used with the argument `allow_feedback`.

In the context of the template is expected:

- The boolean variable `show_feedback`, which will be set to `True` when passing the argument `show_feedback` to the *Tag render_xtdcomment_tree*. If `True` the template will show the list of users who liked the comment and the list of those who disliked it.
- A comment `item`.

Look at the section *Show the list of users* to read on this particular topic.

like.html

This template is rendered when the user clicks on the **like** button of a comment.

The context of the template expects:

- A boolean `already_liked_it` that indicates whether the user already clicked on the like button of this comment. In such a case, if the user submits the form a second time the liked-it flag is withdrawn.
- The `comment` subject to be liked.

liked.html

This template is rendered when the user click on the submit button of the form presented in the `like.html` template. The template is meant to thank the user for the feedback. The context for the template doesn't expect any specific object.

dislike.html

This template is rendered when the user clicks on the **dislike** button of a comment.

The context of the template expects:

- A boolean `already_disliked_it` that indicates whether the user already clicked on the dislike button for this comment. In such a case, if the user submits the form a second time the disliked-it flag is withdrawn.
- The `comment` subject to be liked.

disliked.html

This template is rendered when the user click on the submit button of the form presented in the `dislike.html` template. The template is meant to thank the user for the feedback. The context for the template doesn't expect any specific object.

discarded.html

This template gets rendered if any receiver of the signal `confirmation_received` returns `False`. Informs the user that the comment has been discarded. Read the subsection *Signal and receiver* in the **Control Logic** to know about the `confirmation_received` signal.

email_followup_comment

As `.html` and `.txt`, this template represents the mail message sent when there is a new comment following up the user's. It's sent to the user who posted the comment that is being commented in a thread, or that arrived before the one being sent. To receive this email the user must tick the box *Notify me of follow up comments via email*.

The template expects the following objects in the context:

- The `site` object.
- The `comment` object about which users are being informed.
- The `mute_url` to offer the notified user the chance to stop receiving notifications on new comments.

comment.html

This template is rendered under any of the following circumstances:

- When using the *Tag render_last_xtdcomments*.
- When a logged in user sends a comment via Ajax. The comment gets rendered immediately. JavaScript client side code still has toe handle the response.

posted.html

Rendered when a not authenticated user sends a comment. It informs the user that a confirmation message has been sent and that the link contained in the mail must be clicked to confirm the publication of the comment.

reply.html

Rendered when a user clicks on the **reply** link of a comment. Reply links are created with `XtdComment.get_reply_url` method. They show up below the text of each comment when they allow nested comments.

muted.html

Rendered when a user clicks on the **mute link** received in a follow-up notification message. It informs the user that the site will not send more notifications on new comments sent to the object.

Python Module Index

d

Index