django-comments-xtd Documentation

Release 1.7.1

Daniel Rus Morales

May 30, 2020

Contents

A Django pluggable application that can be used to add comments to your models. 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. Comments can be formatted in Markdown, reStructuredText, linebreaks or plain text.
10. Emails sent through threads (can be disable to allow other solutions, like a Celery app).

CHAPTER 1

Contents

1.1 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 1.8 django-comments-xtd comes with templates styled with twitter-bootstrap v3 that allows you to start using the app right away. 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.

1.2 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

1.2.1 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.

1.2.2 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/demo/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 to show a given post in detail.
- The **templates** directory, with a **base.html** template, a **home.html** template, and two 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. In the following section we will make changes to enable django-comments-xtd.

1.2.3 Configuration

Now that the project is up and 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, `democx/democx/urls.py`, to mount the URL patterns of django_comments_xtd to the path `/comments/`. The urls installed with django_comments_xtd include those required by django_comments 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.

After these simple changes the project is ready to use comments, we just need to modify the blog templates.

1.2.4 Comment confirmation

In order to make django-comments-xtd request comment confirmation by mail we need to set the *COMMENTS_XTD_SALT* setting. This setting helps obfuscating the comment before the user has approved its publication.

This 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 entry to the settings module to help obfuscating the comment before it is sent for confirmation:

```
COMMENTS_XTD_SALT = (b"Timendi causa est nescire. "
                     b"Aequam memento rebus in arduis servare mentem.")
```

1.2.5 Comments tags

In order to be able to post comments to blog stories we need to edit the template file `blog/post_detail.html` and load the `comments` templatetag module, which is provided by the Django Comments Framework:

```
{% load comments %}
```

We will apply changes in the the blog post detail template:

1. To show the number of comments posted to the blog story,
2. To list the comments already posted, and
3. To show the comment form, so that people can post comments.

By using the `get_comment_count` tag we will show the number of comments posted. Change the code around the link element so that it looks like:

```
{% 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>
  &nbsp;&sdot;&nbsp;
  {{ 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 of the list. The one in django-comments-xtd includes twitter-bootstrap styling. Django will use the first template found, which depends on what application is listed first in `INSTALLED_APPS`, django-comments-xtd in this case.

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

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

Below the list of comments we want to display the comment form, so that users can send their own comments. There are two tags available for the 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.

At the moment 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. Let's 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;&nbsp;{{ 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.

1.2.6 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 we would prefer to follow a different approach. The Django Comments Framework has the moderation capabilities upon which we can build our 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 to a blog post with a user logged in the comment doesn't have to be confirmed. However, when you send it logged out the comment has to be confirmed by clicking on the confirmation link. Right after clicking on the confirmation link the comment will be put on hold, pending for approval.

In both cases 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.

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. Tick the box to select 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 the remote 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. Also be sure to post the comment to a story with a publishing date within the last 365 days, otherwise it will enter in moderation regardless of the mail address domain.

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

```

(continues on next page)

(continued from previous page)

```
...

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 locase_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.

1.2.7 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 be nested 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 we get the list of comments. By default comments are ordered by thread and their position inside the thread, which turns out to be in ascending datetime of arrival. In this example we would like to list newer comments first.

Let's start by editing `tutorial/settings.py` to set up a maximum thread level of 1 and a comment ordering to retrieve newer comments 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>
  &nbsp;&sdot;&nbsp;
  {{ 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">
```

(continues on next page)

(continued from previous page)

```
  {% render_xtdcomment_tree for object %}
</ul>
{% endif %}
{% endblock %}
```

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

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 with django-comments-xtd.

We just have to use both settings, the *COMMENTS_XTD_MAX_THREAD_LEVEL* and *COMMENTS_XTD_MAX_THREAD_LEVEL_BY_APP_MODEL*. The former would be set to the default wide site thread level while the latter would be a dictionary of app.model keys and maximum thread level values.

If we wanted to disable nested comments site wide, and enable nested comments up to level one for blog posts, we would need to 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,
}
```

1.2.8 Flags

The Django Comments Framework supports flagging comments, 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 how to express the opposite with the **Disliked it** flag.

One important requirement to flag a comment is that the user setting the flag must be authenticated. In other words, comments can not be flagged by anonymous users.

Removal suggestion

Let us enable the comment removal flag. Edit the `blog/post_detail.html` template, and at the bottom of the file change the `render_xtdcomment_tree` templatetag by adding the argument **allow_flagging**:

```
...
<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.

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. When hovering the comments we must see a flag at the right side of the comment's header. After we click on it we land in a page in which we are requested to confirm our removal suggestion. Finally, click on the red **Flag** button to confirm the request.

Once we have flagged a comment we can find the flag entry in the admin interface, in the **Comment flags** model, under the Django Comments application.

Getting notifications

A user might want to flag a comment on the basis of a violation of our site's terms of use, maybe on hate speech content, 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.

Let's use `XtdCommentModerator`, edit `blog/models.py` and if you are already using the class `SpamModerator`, which alreadt 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. So that a user can't like and dislike a comment at the same time, only the last action counts. Users can like/dislike at any time and only the last action will prevail.

In this section we will make changes in the tutorial project to give our users the capacity to like or dislike comments. We will make changes in the `blog/post_detail.html` template to introduce a new argument in the **render_xtdcomment_tree** tag:

```
<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.

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

Once the like/dislike flagging is enabled we might want to display the users who actually liked/disliked comments.

Again, by addind an argument to the `render_xtdcomment_tree` templatetag we can get rendered the `includes/django_comments_xtd/user_feedback.html` with the list of participants.

Change the `blog/post_detail.html` to add the argument `show_feedback`. For this functionality to work we have to add a bit of JavaScript code. As django-comments-xtd templates use twitter-bootstrap we will load jQuery and twitter-bootstrap JavaScript libraries from their respective default CDNs too:

```
<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-3.2.1.slim.min.js"
    integrity="sha256-k2WSCIexGzOj3Euiig+TlR8gA0EmPjuc79OEeY5L45g="
    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="popover"]').popover({'html':true})
})</script>
{% endblock %}
```

1.2.9 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 project to provide 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. 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.

1.3 Demo projects

Django-comments-xtd comes with three demo projects:

1. **simple**: Single model with **non-threaded** comments
2. **custom**: Single model with comments provided by a new app that extends django-comments-xtd. The new comment model adds a `title` field to the XtdComment class. Find more details in *Customizing django-comments-xtd*.
3. **comp**: Several models with maximum thread level defined on per app.model pair, moderation, removal suggestion flag, like/dislike flags, and list of users who liked/disliked comments.

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

Table of Contents

1.3.1 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)$ 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 runserver
```

Fixtures data provide:

- An `admin` **User**, with password `admin`
- A default **Site** with domain `localhost:8000` so that URLs sent in mail messages use already the URL of the development web server of Django.

- 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 working values for all the `EMAIL_` settings.

1.3.2 Simple demo

The simple example 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. It uses the template tag `render_markup_comment` to render comment content. So you can use line breaks, Markdown or reStructuredText to format comments. To use special formatting, start the comment with the line `#!<markup-lang>` being `<markup-lang>` either `markdown`, `restructuredtext` or `linebreaks`.
6. No nested comments.

Give it a try and test the features. Setup the project as explained above, run the development server, and visit http://localhost:8000/.

- 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.

1.3.3 Custom demo

The **simple_threads** demo site extends the **simple** demo functionality featuring:

- Thread support up to level 2

1. Visit http://localhost:8000/ and look at the first article page with 9 comments.
2. See the comments in the admin interface too:

- The first field represents the thread level.
- When in a nested comment the first field refers to the parent comment.

1.3.4 Comp demo

The **multiple** demo allows users post comments to three different type of instances: stories, quotes, and releases. Stories and quotes belong to the **blog app** while releases belong to the **projects app**. The demo shows the blog homepage with the last 5 comments posted to either stories or quotes and a link to the complete paginated list of comments posted to the blog. It features:

- Definition of maximum thread level on a per app.model basis.
- Use of comments_xtd template tags, `get_xtdcomment_count`, `render_last_xtdcomments`, `get_last_xtdcomments`, and the filter `render_markup_comment`.

1. Visit http://localhost:8000/ and take a look at the **Blog** and **Projects** pages.

- The **Blog** contains **Stories** and **Quotes**. Instances of both models have comments. The blog index page shows the **last 5 comments** posted to either stories or quotes. It also gives access to the **complete paginated list of comments**.
- Project releases have comments as well but are not included in the complete paginated list of comments shown in the blog.

2. To render the last 5 comments the site uses:
   - The templatetag `{% render_last_xtdcomments 5 for blog.story blog.quote %}`
   - And the following template files from the `demos/multiple/templates` directory:
   - `django_comments_xtd/blog/story/comment.html` to render comments posted to **stories**
   - `django_comments_xtd/blog/quote/comment.html` to render comments posted to **quotes**
   - You may rather use a common template to render comments:
   - For all blog app models: `django_comments_xtd/blog/comment.html`
   - For all the website models: `django_comments_xtd/comment.html`
3. To render the complete paginated list of comments the site uses:
   - An instance of a generic `ListView` class declared in `blog/urls.py` that uses the following queryset:
   - `XtdComment.objects.for_app_models("blog.story", "blog.quote")`
4. The comment posted to the story **Net Neutrality in Jeopardy** starts with a specific line to get the content rendered as reStructuredText. Go to the admin site and see the source of the comment; it's the one sent by Alice to the story 2.
   - To format and render a comment in a markup language, make sure the first line of the comment looks like: `#!<markup-language>` being `<markup-language>` any of the following options:
   - markdown
   - restructuredtext
   - linebreaks
   - Then use the filter `render_markup_comment` with the comment field in your template to interpret the content (see `demos/multiple/templates/comments/list.html`).

1.4 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:

a. 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:

a. 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:

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

   b. Otherwise creates an instance of `TmpXtdComment` model: an in-memory representation of the comment.

   c. 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*.

   d. 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:

a. Checks the secured token in the URL. If it's wrong returns a 404 code.

b. Otherwise checks whether the comment was already confirmed, in such a case returns a 404 code.

c. 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.

d. Otherwise an instance of `XtdComment` finally hits the database, and

e. An email notification is sent to previous comments followers telling them about the new comment following up theirs.

1.4.1 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.

1.4.2 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):
```

(continues on next page)

(continued from previous page)

```
        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.

1.5 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

1.5.1 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  %}
```

1.5.2 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 %}
```

1.5.3 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 %}
```

1.5.4 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 %}
```

1.5.5 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 %}
```

1.5.6 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.

1.5.7 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.

1.5.8 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`.

1.6 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.

1.6.1 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.

1.7 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

1.7.1 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.

1.7.2 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*.

1.7.3 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`.

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

It defaults to `settings.DEFAULT_FROM_EMAIL`.

1.7.5 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"*.

1.7.6 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"*.

1.7.7 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')`

1.7.8 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`.

1.7.9 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.

1.7.10 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.

1.7.11 `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`.

1.8 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

1.8.1 `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.

1.8.2 `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.

1.8.3 `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.

1.8.4 `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.

1.8.5 `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.

1.8.6 `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.

1.8.7 `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.

1.8.8 `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.

1.8.9 `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.

1.8.10 `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.

1.8.11 `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.

1.8.12 `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.

1.8.13 `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

Index