django-comments-xtd Documentation

Release 1.6.7

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. Install the Django Comments Framework by running `pip install django-contrib-comments`.
2. Install the django-comments-xtd app by running `pip install django-comments-xtd`.
3. Enable the "sites" framework by adding `'django.contrib.sites'` to `INSTALLED_APPS` and defining `SITE_ID`. 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 by django_comments_xtd to create comment verification URLs, follow-up cancellation URLs, etc.
4. Install the comments framework by adding `'django_comments'` to `INSTALLED_APPS`.
5. Install the comments-xtd app by adding `'django_comments_xtd'` to `INSTALLED_APPS`.
6. Set the `COMMENTS_APP` setting to `'django_comments_xtd'`.
7. 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.
# 1: Nested up to level one.
# 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, so that different models have enabled different thread levels. ie: no nested comments for food recipes, up to thread level one for blog posts, etc.

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

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

11. 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>"
```

12. If you wish to allow comments written in a markup language like Markdown or reStructuredText, install django-markup by running `pip install django-markup`.
13. 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.
14. 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 in a glance 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. Use it at your will to apply the changes while reading each section.

1.2.1 Preparation

If you opt for coding the examples, download the bare Django project tarball from this GitHub page and decompress it in a directory of your choice.

The most comfortable approach to set it up consist of creating a virtualenv and installing all the dependencies within it. The dependencies are just a bunch of lightweight packages.

```
$ virtualenv -p python3.5 ~/venv/comments-xtd-tutorial
$ source ~/venv/comments-xtd-tutorial/bin/activate
$ cd ~/src
$ wget https://github.com/danirus/django-comments-xtd/demo/democx.tar.gz
$ tar -xvzf democx.tar.gz
$ cd democx
```

Take a look at the project. The starting point of this tutorial is a simple Django project with a blog application and a few posts. During the following sections we will configure the project to enable comments to the `Post` model and try every possible commenting scenario.

1.2.2 Configuration

Configure the project to support django-comments and django-comments-xtd, start installing the packages and then let's change the minimum number of settings to enable comments.

```
$ pip install Django pytz django-contrib-comments django-comments-xtd \
              docutils Markdown django-markup
```

Edit the settings module of the project, `democx/democx/settings.py`, and make the following changes:

```
INSTALLED_APPS = [
    ...
    'django_comments',
    'django_comments_xtd',
    ...
]
...
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 and launch the development server:

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

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 being used by django_comments_xtd to create comment verification URLs, follow-up cancellation URLs, etc. You can edit the site instance in the admin interface to set it to the right value.

After these simple changes the project is ready to use comments, we just need to modify the templates to include the `comments` templatetag module.

1.2.3 Changes in templates

The tutorial project comes ready with a blog application that contains a Post model. Our goal is to provide blog stories with comments, so that people can post comments to the stories and read the comments other people have posted.

The blog application, located in `democx/blog` contains a `blog_detail.html` template in the `templates/blog` directory. We have to edit the template and load the `comments` templatetag module, which is provided by the Django Comments Framework:

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

Let's insert now the tags 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 people can post comments.

By using the `get_comment_count` tag we will show the number of comments posted, right below the text of the blog post. The last part of the template should look like this:

```
{% get_comment_count for post 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 do the changes to list the comments. 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
```

Let's use `render_comment_list` in our `blog/blog_detail.html` template to add the list of comments at the end of the page, before the `endblock` tag:

```
<div class="comments">
  {% render_comment_list for post %}
</div>
```

Below the list of comments we want to display the comment form (later we will put the form first), so that users can send their own comments. There are two tags available for such 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`:

```
<div class="comment">
  <h4 class="text-center">Your comment</h4>
  <div class="well">
    {% render_comment_form for post %}
  </div>
</div>
```

Finally, before completing this first set of changes, we could show the number of comments along each post title in the blog's home page. We would have to edit the `blog/home.html` template and make the following changes:

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

...
<p class="date">
  {% get_comment_count for post as comment_count %}
  Published {{ post.publish }} by {{ post.author }}
  {% 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 reception of the mail confirmation request do logout of the admin interface.

By default the setting *COMMENTS_XTD_MAX_THREAD_LEVEL* is 0, which means comments can not be nested. In the following sections we will enable threaded comments, we will allow users to flag comments and we will set up comment moderation.

1.2.4 Template Style Customization

The `democx` project uses the fronted web framework, Bootstrap, which allows fast interface development. There are other client side frameworks, this example uses Bootstrap because it's probably the most popular.

We will adapt our templates to integrate the list of comments and the comment form with the look provided by Bootstrap CSS classes.

Comment list

We must create a template `list.html` inside the `democx/templates/comments` directory with the following content:

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

<ul class="media-list" id="comments">
  {% for comment in comment_list %}
  <li class="media" id="c{{ comment.id }}">
    <div class="media-left">
```

(continues on next page)

(continued from previous page)

```
      <a href="{{ comment.url }}">{{ comment.user_email|xtd_comment_gravatar
↪}}</a>
    </div>
    <div class="media-body">
      <h6 class="media-heading">
        <a class="permalink text-muted" href="{% get_comment_permalink
↪comment %}">¶</a> &sdot;
        {{ comment.submit_date }} - 
        {% if comment.url %}<a href="{{ comment.url }}" target="_new">
        {% endif %}{{ comment.name }}{% if comment.url %}</a>{% endif %}
      </h6>
      <p>{{ comment.comment }}</p>
    </div>
  </li>
  {% endfor %}
</ul>
```

Form class

In order to customize the fields of the comment form we will create a new form class inside the blog application and change the setting *COMMENTS_XTD_FORM_CLASS* to point to that new form class.

First, create a new file `forms.py` inside the `democx/blog` directory with the following content:

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

from django_comments_xtd.forms import XtdCommentForm


class MyCommentForm(XtdCommentForm):
    def __init__(self, *args, **kwargs):
        if 'comment' in kwargs:
            followup_suffix = ('_%d' % kwargs['comment'].pk)
        else:
            followup_suffix = ''
        super(MyCommentForm, self).__init__(*args, **kwargs)
        for field_name, field_obj in self.fields.items():
            if field_name == 'followup':
                field_obj.widget.attrs['id'] = 'id_followup%s' % followup_
↪suffix
                continue
            field_obj.widget.attrs.update({'class': 'form-control'})
            if field_name == 'comment':
                field_obj.widget.attrs.pop('cols')
                field_obj.widget.attrs.pop('rows')
                field_obj.widget.attrs['placeholder'] = _('Your comment')
                field_obj.widget.attrs['style'] = "font-size: 1.1em"
            if field_name == 'url':
                field_obj.help_text = _('Optional')
        self.fields.move_to_end('comment', last=False)
```

In `democx/democs/settings.py` add the following:

```
COMMENTS_XTD_FORM_CLASS = "blog.forms.MyCommentForm"
```

Form template

Now we must create a file `form.html` within the `democx/template/comments` directory containing the code that renders the comment form. It must include each and every visible form field: `comment`, `name`, `email`, `url` and `follow up`:

```
{% load i18n %}
{% load comments %}

<form method="POST" action="{% comment_form_target %}" class="form-horizontal
↪">
  {% csrf_token %}
  <fieldset>
    <div><input type="hidden" name="next" value="{% url 'comments-xtd-sent'
↪%}"/></div>

    <div class="alert alert-danger hidden" data-comment-element="errors">
    </div>

    {% for field in form %}
      {% if field.is_hidden %}<div>{{ field }}</div>{% endif %}
    {% endfor %}

    <div style="display:none">{{ form.honeypot }}</div>

    <div class="form-group {% if 'comment' in form.errors %}has-error{%␣
↪endif %}">
      <div class="col-lg-offset-1 col-md-offset-1 col-lg-10 col-md-10">
        {{ form.comment }}
      </div>
    </div>

    <div class="form-group {% if 'name' in form.errors %}has-error{% endif %}
↪">
      <label for="id_name" class="control-label col-lg-3 col-md-3">
        {{ form.name.label }}
      </label>
      <div class="col-lg-7 col-md-7">
        {{ form.name }}
      </div>
    </div>

    <div class="form-group {% if 'email' in form.errors %}has-error{% endif
↪%}">
      <label for="id_email" class="control-label col-lg-3 col-md-3">
        {{ form.email.label }}
      </label>
      <div class="col-lg-7 col-md-7">
        {{ form.email }}
        <span class="help-block">{{ form.email.help_text }}</span>
      </div>
    </div>

    <div class="form-group {% if 'url' in form.errors %}has-error{% endif %}
↪">
      <label for="id_url" class="control-label col-lg-3 col-md-3">
        {{ form.url.label }}
      </label>
```

(continues on next page)

(continued from previous page)

```
      <div class="col-lg-7 col-md-7">
        {{ form.url }}
      </div>
    </div>

    <div class="form-group">
      <div class="col-lg-offset-3 col-md-offset-3 col-lg-7 col-md-7">
        <div class="checkbox">
          <label for="id_followup{% if cid %}_{{ cid }}{% endif %}">
            {{ form.followup }}  {{ form.followup.label }}
          </label>
        </div>
      </div>
    </div>
  </fieldset>

  <div class="form-group">
    <div class="col-lg-offset-3 col-md-offset-3 col-lg-7 col-md-7">
      <input type="submit" name="post" value="send" class="btn btn-primary" /
↪>
      <input type="submit" name="preview" value="preview" class="btn btn-
↪default" />
    </div>
  </div>
</form>
```

Preview template

When we click on the preview button Django looks for the `preview.html` template in different directories and with different names:

```
comments/blog_post_preview.html
comments/blog_preview.html
comments/blog/post/preview.html
comments/blog/preview.html
comments/preview.html
```

We will provide the last of them by adding the file `preview.html` to the `democx/templates/comments/` directory with the following code:

```
{% extends "base.html" %}
{% load i18n %}
{% load comments_xtd %}

{% block content %}
<h4>{% trans "Preview your comment:" %}</h4>
<div class="row">
  <div class="col-lg-offset-1 col-md-offset-1 col-lg-10 col-md-10">
    <div class="media">
      {% if not comment %}
      <em>{% trans "Empty comment." %}</em>
      {% else %}
      <div class="media-left">
        <a href="{{ form.cleaned_data.url }}">
          {{ form.cleaned_data.email|xtd_comment_gravatar }}
```

(continues on next page)

(continued from previous page)

```
        </a>
      </div>
      <div class="media-body">
        <h6 class="media-heading">
          {% now "N j, Y, P" %} - 
          {% if form.cleaned_data.url %}
          <a href="{{ form.cleaned_data.url }}" target="_new">{% endif %}
          {{ form.cleaned_data.name }}
          {% if form.cleaned_data.url %}</a>{% endif %}
        </h6>
        <p>{{ comment }}</p>
      </div>
      {% endif %}
    </div>
    <div class="visible-lg-block visible-md-block">
      <hr/>
    </div>
  </div>
</div>
<div class="well well-lg">
  {% include "comments/form.html" %}
</div>
{% endblock %}
```

Posted template

Finally, when we hit the send button and the comment gets succesfully processed Django renders the template `comments/posted.html`. We can modify the look of this template by adding a new `posted.html` file to our `democx/templates/comments` directory with the following code:

```
{% extends "base.html" %}
{% load i18n %}

{% block header %}
<a href="{% url 'homepage' %}">{{ block.super }}</a> -
<a href="{% url 'blog:post_list' %}">Blog</a>
{% endblock %}

{% block content %}
<h3 class="text-center">{% trans "Comment confirmation requested." %}</h3>
<p>{% blocktrans %}A confirmation message has been sent to your
email address. Please, click on the link in the message to confirm
your comment.{% endblocktrans %}</p>
{% endblock %}
```

Now we have the form and the core templates integrated with the Bootstrap framework. You can visit a blog post and give it a try. Remember that to get the comment confirmation request by email you must sign out of the admin interface.

You might want to adapt the design of the rest of *Templates* provided by django-comments-xtd.

1.2.5 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 democx/democx/settings.py module:

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

Now we have to 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 also need to register our Moderator class with the django-comments-xtd's moderator object. We need to 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 democx/blog/model.py module:

```
...
# New imports to add below the current ones.
try:
    from django_comments.moderation import CommentModerator
except ImportError:
    from django.contrib.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)
```

We may want to customize the look of the moderated.html template. Let's create the directory django_comments_xtd under democx/templates and create inside the file moderated.html with the following code:

```
{% extends "base.html" %}
{% load i18n %}

{% block title %}{% trans "Comment requires approval." %}{% endblock %}

{% block header %}
<a href="{% url 'homepage' %}">{{ block.super }}</a> -
<a href="{% url 'blog:post_list' %}">Blog</a>
{% endblock %}
```

(continues on next page)

(continued from previous page)

```
{% block content %}
<h4 class="text-center">{% trans "Comment in moderation" %}</h4>
<p class="text-center">
{% blocktrans %}Your comment has to be reviewed before approbal.<br/>
  It has been put automatically in moderation.<br/>
  Thank you for your patience and understanding.{% endblocktrans %}
</p>
{% endblock %}
```

Additionally we need a `comments/comment_notification_email.txt` template. This template is used by django-contrib-comments to render the email message that `MANAGERS` receive when using the moderation option email_notification, as we do above in our `PostCommentModerator` class. Django-comments-xtd comes already with such a template.

Now we are ready to try the moderation in place. Let's visit the web page of a blog post with a `publish` datetime older than a year and try to send a comment. After confirming the comment you must be redirected to the `moderated.html` template and your comment must be put on hold for approval.

On the other hand if you send a comment to a blog post created within the last year your comment will not be hold for moderation. Exercise it with both, a user logged in (login in the admin site with username **admin** and password **admin** will suffice) and logged out (click on **log out** at the top-right corner of the admin site).

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 the user clicks on the link in the confirmation email the comment is put on hold pending for approval.

In both cases, if you have provided an active email address in the `MANAGERS` setting, you will receive a notification about the reception of a new comment, an email with a subject contaning the site domain within angle brackets. If you did not received such message, you might need to review your email settings. Read above the *Configuration* section and see what are the settings you must enable. Add a hash in front of the `EMAIL_BACKEND` setting to comment it, this way Django won't use the console to output emails but rather the default email backend along with the rest of email settings provided.

A reminder to finish this section: we need to review those comments put on hold. For such purpose we should visit the comments-xtd app in the admin interface. After reviewing the non-public comments, we must tick the box of those we want to approve, select the action **Approve selected comments** and click on the **Go** button.

Disallow black listed domains

In case you wanted to disable the comment confirmation by email you might be interested in setting up some sort of control to reject spammers. 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 on hold comments containing badwords.

Let us first disable comment confirmation, we need to edit the `settings.py` module:

```
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 our `democx/blog/models.py` module 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
```

(continues on next page)

(continued from previous page)

```
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 we can 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 us 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 the comment's `is_public` attribute will be put to `False` and therefore the comment will be on hold waiting for approval.

The blog application comes already with what we are going to consider a bad word (for the purpose of this tutorial, we will use this badwords list), which are listed in the `democx/blog/badwords.py` module.

We will assume that we already have a list of `BlackListed` domains in our database, as explained in the previous section, and we don't want further spam control, so we want to disable comment confirmation by email. Let's edit the `settings.py` module:

```
COMMENTS_XTD_CONFIRM_EMAIL = False
```

Then let's edit the `democx/blog/models.py` module and add the following code corresponding to our new `PostCommentModerator`:

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

        msg = dict([(clean(w), i)
                    for i, w in enumerate(comment.comment.lower().split())])
```

(continues on next page)

(continued from previous page)

```
        for badword in badwords:
            if isinstance(badword, str):
                if badword in msg:
                    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 send a comment to a blog post and put any of the words listed in the badwords list in the message. After clicking on the send button we must see the `moderated.html` template and the comment must be put on hold for approval.

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.

1.2.6 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* will 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 will use the tag *get_xtdcomment_tree* that retrieves the comments in a nested data structure, and we will create a new template to render the nested comments.

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 the `democx/democx/settings.py` module 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 module and make use of the *get_xtdcomment_tree* tag. 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.

Let's edit `democx/blog/templates/blog/blog_detail.html` to make it look like follows:

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

{% block title %}{{ post.title }}{% endblock %}

{% block header %}
<a href="{% url 'homepage' %}">{{ block.super }}</a> -
<a href="{% url 'blog:post_list' %}">Blog</a>
{% endblock %}

{% block content %}
<h3 class="page-header text-center">My blog</h3>
<h4>{{ post.title }}</h4>
<p class="date">
  Published {{ post.publish }} by {{ post.author }}
</p>
{{ post.body|linebreaks }}

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

<div class="well">
  {% render_comment_form for post %}
</div>

{% if comment_count %}
<hr/>
<ul class="media-list">
  {% get_xtdcomment_tree for post as comments_tree %}
  {% include "blog/comments_tree.html" with comments=comments_tree %}
</ul>
{% endif %}
{% endblock %}
```

At the end of the file we use another template to render the list of comments. This template will render all the comments in the same thread level and will call itself to render those in nested levels. Let's create the template `blog/comments_tree.html` and add the following code to it:

```
{% load i18n %}
{% load comments %}
{% load comments_xtd %}

{% for item in comments %}
{% if item.comment.level == 0 %}
<li class="media">{% else %}<div class="media">{% endif %}
  <a name="c{{ item.comment.id }}"></a>
  <div class="media-left">{{ item.comment.user_email|xtd_comment_gravatar }}
↪</div>
  <div class="media-body">
```

(continues on next page)

(continued from previous page)

```
    <div class="comment">
      <h6 class="media-heading">
        {{ item.comment.submit_date }} - {% if item.comment.url␣
↪and not item.comment.is_removed %}<a href="{{ item.comment.url }}" target=
↪"_new">{% endif %}{{ item.comment.name }}{% if item.comment.url %}</a>{%␣
↪endif %}  <a class="permalink" title="comment permalink" href="{
↪% get_comment_permalink item.comment %}">¶</a>
      </h6>
      {% if item.comment.is_removed %}
      <p>{% trans "This comment has been removed." %}</p>
      {% else %}
      <p>
        {{ item.comment.comment|render_markup_comment }}
        <br/>
        {% if item.comment.allow_thread and not item.comment.is_removed %}
        <a class="small mutedlink" href="{{ item.comment.get_reply_url }}">
          {% trans "Reply" %}
        </a>
        {% endif %}
      </p>
      {% endif %}
    </div>
    {% if not item.comment.is_removed and item.children %}
    <div class="media">
      {% include "blog/comments_tree.html" with comments=item.children %}
    </div>
    {% endif %}
  </div>
{% if item.comment.level == 0 %}
</li>{% else %}</div>{% endif %}
{% endfor %}
```

This template uses the tag *xtd_comment_gravatar* included within the `comments_xtd.py` templatetag module, that loads the gravatar image associated with an email address. It also uses *render_markup_comment*, that will render the comment using either markdown, restructuredtext, or linebreaks.

Another important remark on this template is that it calls itself recursively to render nested comments for each comment. The tag *get_xtdcomment_tree* retrieves a list of dictionaries. Each dictionary contains two attributes: `comment` and `children`. The attribute `comment` is the `XtdComment` object and the attribute `children` is another list of dictionaries with the nested comments.

We don't necessarily have to use *get_xtdcomment_tree* to render nested comments. It is possible to render them by iterating over the list of comments and accessing the level attribute. Take a look at the `simple_threaded` demo project, the `list.html` template iterates over the list of comments adding an increasing left padding depending on the thread level the comment belongs to.

Finally we might want to adapt the `django_comments_xtd/reply.html` template, that will be rendered when the user clicks on the reply link:

```
{% extends "base.html" %}
{% load i18n %}
{% load comments %}
{% load comments_xtd %}

{% block title %}{% trans "Comment reply" %}{% endblock %}

{% block header %}
```

(continues on next page)

(continued from previous page)

```
<a href="{% url 'homepage' %}">{{ block.super }}</a> -
<a href="{% url 'blog:post_list' %}">Blog</a> -
<a href="{{ comment.content_object.get_absolute_url }}">{{ comment.content_
↪object }}</a>
{% endblock %}

{% block content %}
<h4 class="page-header text-center">{% trans "Reply to comment" %}</h4>
<div class="row">
  <div class="col-lg-offset-1 col-md-offset-1 col-lg-10 col-md-10">
    <div class="media">
      <div class="media-left">
        {% if comment.user_url %}
        <a href="{{ comment.user_url }}">
          {{ comment.user_email|xtd_comment_gravatar }}
        </a>
        {% else %}
        {{ comment.user_email|xtd_comment_gravatar }}
        {% endif %}
      </div>
      <div class="media-body">
        <h6 class="media-heading">
          {{ comment.submit_date|date:"N j, Y, P" }} - 
          {% if comment.user_url %}
          <a href="{{ comment.user_url }}" target="_new">{% endif %}
          {{ comment.user_name }}{% if comment.user_url %}</a>{% endif %}
        </h6>
        <p>{{ comment.comment }}</p>
      </div>
    </div>
    <div class="visible-lg-block visible-md-block">
      <hr/>
    </div>
  </div>
</div>
<div class="well well-lg">
  {% include "comments/form.html" %}
</div>
{% endblock %}
```

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 literals as keys and the corresponding maximum thread level as 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 = {
```

(continues on next page)

(continued from previous page)

```
    # Objects of the app blog, model post, can be nested
    # up to thread level 1.
        'blog.post': 1,
}
```

1.2.7 Flags

The Django Comments Framework comes with support for flagging comments, so that a comment can receive the following flags:

- **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 extends the functionality provided by django-contrib-comments 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 start by enabling the link that allows a user to suggest a comment removal. This functionality is already provided by django-contrib-comments. We will simply put it in the template.

To place the flag link we need to edit the `blog/comments_tree.html` template. We will show the flag link at the right side of the comment's header:

```
...
{% for item in comments %}
  ...
        <h6 class="media-heading">
          {{ item.comment.submit_date }} - 
          {% if item.comment.url and not item.comment.is_removed %}
          <a href="{{ item.comment.url }}" target="_new">{% endif %}
            {{ item.comment.name }}{% if item.comment.url %}
          </a>{% endif %}  
          <a class="permalink" href="{% get_comment_permalink item.comment %}
↪">¶</a>

          <!-- Add this to enable flagging a comment -->
          {% if request.user.is_authenticated %}
          <div class="pull-right">
            <a class="mutedlink" href="{% url 'comments-flag' item.comment.
↪pk %}">
              <span class="glyphicon glyphicon-flag" title="flag comment"></
↪span>
            </a>
```

(continues on next page)

(continued from previous page)

```
          </div>
          {% endif %}
        </h6>
  ...
```

Additionally we might want to adapt the style of two related templates: `comments/flag.html` and `comments/flagged.html`. The first presents a form to the user to confirm the removal suggestion, while the second renders a confirmation message once the user has flagged the comment.

Let's create the template `flag.html` in the directory `democx/templates/comments` with this content:

```
{% extends "base.html" %}
{% load i18n %}
{% load comments_xtd %}

{% block title %}{% trans "Flag this comment" %}{% endblock %}

{% block header %}
<a href="{% url 'homepage' %}">{{ block.super }}</a> -
<a href="{% url 'blog:post_list' %}">Blog</a> -
<a href="{{ comment.content_object.get_absolute_url }}">{{ comment.content_
↪object }}</a>
{% endblock %}

{% block content %}
<h4 class="page-header text-center">{% trans "Really flag this comment?" %}</
↪h4>
<p class="text-center">{% trans "Click on the flag button if you want to␣
↪suggest the removal of the following comment:" %}</p>
<div class="row">
  <div class="col-lg-offset-1 col-md-offset-1 col-lg-10 col-md-10">
    <div class="media">
      <div class="media-left">
        {% if comment.user_url %}
        <a href="{{ comment.user_url }}">
          {{ comment.user_email|xtd_comment_gravatar }}
        </a>
        {% else %}
        {{ comment.user_email|xtd_comment_gravatar }}
        {% endif %}
      </div>
      <div class="media-body">
        <h6 class="media-heading">
          {{ comment.submit_date|date:"N j, Y, P" }} - 
          {% if comment.user_url %}
          <a href="{{ comment.user_url }}" target="_new">{% endif %}
            {{ comment.user_name }}
            {% if comment.user_url %}
          </a>{% endif %}
        </h6>
        <p>{{ comment.comment }}</p>
      </div>
    </div>
    <div class="visible-lg-block visible-md-block">
      <hr/>
    </div>
  </div>
```

(continues on next page)

(continued from previous page)

```
</div>
<div class="row">
  <div class="col-lg-offset-1 col-md-offset-1 col-lg-10 col-md-10">
    <div class="well well-lg">
      <form action="." method="post" class="form-horizontal">{% csrf_token %}
        <div class="form-group">
          <div class="col-lg-offset-3 col-md-offset-3 col-lg-7 col-md-7">
            <input type="submit" name="submit" class="btn btn-danger" value="
↪{% trans "Flag" %}"/>
            <a class="btn btn-default" href="{{ comment.get_absolute_url }}">
↪cancel</a>
          </div>
        </div>
      </form>
    </div>
  </div>
</div>
{% endblock %}
```

And the template `flagged.html` in the same directory `democx/templates/comments` with the code:

```
{% extends "base.html" %}
{% load i18n %}
{% load comments_xtd %}

{% block title %}{% trans "Thanks for flagging" %}.{% endblock %}

{% block header %}
<a href="{% url 'homepage' %}">{{ block.super }}</a> -
<a href="{% url 'blog:post_list' %}">Blog</a> -
<a href="{{ comment.content_object.get_absolute_url }}">{{ comment.content_
↪object }}</a>
{% endblock %}

{% block content %}
<h4 class="page-header text-center">Thanks for flagging</h4>
<p class="text-center">{% trans "Thank you for taking the time to improve␣
↪the quality of discussion in our site." %}<p>
{% endblock %}
```

Now we can try it, 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 to which we have sent comments. When hovering the comments we must see a flag at the right side of the comment's header. If we click on it we will land in the page where we are requested to confirm our suggestion to remove the comment. If we click on the red **Flag** button we will create the **Removal suggestion** flag for the comment.

Once we have flagged a comment we can find the flag entry in the admin interface, under 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 offered by the parent class `XtdCommentModerator` exposes the attribute `removal_suggestion_notification`. When this attribute is set to `True` Django will send an email to the `MANAGERS` on every **Removal suggestion** flag created.

Let's use `XtdCommentModerator` in our demo. If you are using the class `SpamModerator` already in the `democx/blog/models.py` module, then simply add `removal_suggestion_notification = True` to your Moderation class, as `SpamModerator` already inherits from `XtdCommentModerator`. 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 email is based on the template `django_comments_xtd/removal_notification_email.txt` already provided within the django-comments-xtd app. After these changes flagging a comment with a **Removal suggestion** must trigger an email.

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 click on the links at any time and only the last action will prevail.

In this section we will make changes in the `democx` project to give our users the capacity to like or dislike comments. We can start by adding the links to the `comments_tree.html` template. The links could go immediately after rendering the comment content, at the left side of the Reply link:

```
...
        <p>
          {{ item.comment.comment|render_markup_comment }}
          <br/>
          <!-- Add here the links to let users express whether they like the␣
↪comment. -->
          <a href="{% url 'comments-xtd-like' item.comment.pk %}" class=
↪"mutedlink">
            <span class="small">{{ item.likedit|length }}</span> 
            <span class="small glyphicon glyphicon-thumbs-up"></span>
          </a>
          <span class="text-muted">&sdot;</span>
          <a href="{% url 'comments-xtd-dislike' item.comment.pk %}" class=
↪"mutedlink">
            <span class="small">{{ item.dislikedit|length }}</span> 
            <span class="small glyphicon glyphicon-thumbs-down"></span>
          </a>
          <span class="text-muted">&sdot;</span>
          <!-- And the reply link -->
          {% if item.comment.allow_thread and not item.comment.is_removed %}
```

(continues on next page)

(continued from previous page)

```
          <a class="small mutedlink" href="{{ item.comment.get_reply_url }}">
            {% trans "Reply" %}
          </a>
          {% endif %}
        </p>
...
```

Having the links in place, if we click on any of them we will end up in either the `like.html` or the `dislike.html` templates. These two templates are new, and meant to request the user to confirm the operation.

We can create new versions of these templates in the `democx/templates/django_comments_xtd` directory to adapt them to the look of the project. Let's create first `like.html` with the following content:

```
{% extends "base.html" %}
{% load i18n %}
{% load comments_xtd %}

{% block title %}{% trans "Confirm your opinion" %}{% endblock %}

{% block header %}
<a href="{% url 'homepage' %}">{{ block.super }}</a> -
<a href="{% url 'blog:post_list' %}">Blog</a> -
<a href="{{ comment.content_object.get_absolute_url }}">{{ comment.content_
↪object }}</a>
{% endblock %}

{% block content %}
<h4 class="page-header text-center">
  {% if already_liked_it %}
  {% trans "You liked this comment, do you want to change it?" %}
  {% else %}
  {% trans "Do you like this comment?" %}
  {% endif %}
</h4>
<p class="text-center">{% trans "Please, confirm your opinion on this␣
↪comment:" %}</p>
<div class="row">
  <div class="col-lg-offset-1 col-md-offset-1 col-lg-10 col-md-10">
    <div class="media">
      <div class="media-left">
        {% if comment.user_url %}
        <a href="{{ comment.user_url }}">
          {{ comment.user_email|xtd_comment_gravatar }}
        </a>
        {% else %}
        {{ comment.user_email|xtd_comment_gravatar }}
        {% endif %}
      </div>
      <div class="media-body">
        <h6 class="media-heading">
          {{ comment.submit_date|date:"N j, Y, P" }} - 
          {% if comment.user_url %}
          <a href="{{ comment.user_url }}" target="_new">{% endif %}
            {{ comment.user_name }}
            {% if comment.user_url %}
          </a>{% endif %}
        </h6>
```

(continues on next page)

(continued from previous page)

```
        <p>{{ comment.comment }}</p>
      </div>
    </div>
    <div class="visible-lg-block visible-md-block">
      <hr/>
    </div>
  </div>
</div>
<div class="row">
  <div class="col-lg-offset-1 col-md-offset-1 col-lg-10 col-md-10">
    {% if already_liked_it %}
    <div class="alert alert-warning">
      {% trans 'Click on the "withdraw" button if you want to withdraw your␣
↪positive opinion on this comment.' %}
    </div>
    {% endif %}
    <div class="well well-lg">
      <form action="." method="post" class="form-horizontal">{% csrf_token %}
        <input type="hidden" name="next" value="{{ comment.get_absolute_url }
↪}">
        <div class="form-group">
          <div class="col-lg-offset-3 col-md-offset-3 col-lg-7 col-md-7">
            <input type="submit" name="submit" class="btn btn-primary" value=
↪"{% if already_liked_it %}{% trans 'Withdraw' %}{% else %}{% trans 'I like␣
↪it' %}{% endif %}"/>
            <a class="btn btn-default" href="{{ comment.get_absolute_url }}">
↪{% trans "cancel" %}</a>
          </div>
        </div>
      </form>
    </div>
  </div>
</div>
{% endblock %}
```

And this could be the content for the `dislike.html` template:

```
{% extends "base.html" %}
{% load i18n %}
{% load comments_xtd %}

{% block title %}{% trans "Confirm your opinion" %}{% endblock %}

{% block header %}
<a href="{% url 'homepage' %}">{{ block.super }}</a> -
<a href="{% url 'blog:post_list' %}">Blog</a> -
<a href="{{ comment.content_object.get_absolute_url }}">{{ comment.content_
↪object }}</a>
{% endblock %}

{% block content %}
<h4 class="page-header text-center">
  {% if already_disliked_it %}
  {% trans "You didn't like this comment, do you want to change it?" %}
  {% else %}
  {% trans "Do you dislike this comment?" %}
  {% endif %}
```

(continues on next page)

(continued from previous page)

```
</h4>
<p class="text-center">{% trans "Please, confirm your opinion on this␣
↪comment:" %}</p>
<div class="row">
  <div class="col-lg-offset-1 col-md-offset-1 col-lg-10 col-md-10">
    <div class="media">
      <div class="media-left">
        {% if comment.user_url %}
        <a href="{{ comment.user_url }}">
          {{ comment.user_email|xtd_comment_gravatar }}
        </a>
        {% else %}
        {{ comment.user_email|xtd_comment_gravatar }}
        {% endif %}
      </div>
      <div class="media-body">
        <h6 class="media-heading">
          {{ comment.submit_date|date:"N j, Y, P" }} - 
          {% if comment.user_url %}
          <a href="{{ comment.user_url }}" target="_new">{% endif %}
            {{ comment.user_name }}
            {% if comment.user_url %}
          </a>{% endif %}
        </h6>
        <p>{{ comment.comment }}</p>
      </div>
    </div>
    <div class="visible-lg-block visible-md-block">
      <hr/>
    </div>
  </div>
</div>
<div class="row">
  <div class="col-lg-offset-1 col-md-offset-1 col-lg-10 col-md-10">
    {% if already_liked_it %}
    <div class="alert alert-warning">
      {% trans 'Click on the "withdraw" button if you want to withdraw your␣
↪negative opinion on this comment.' %}
    </div>
    {% endif %}
    <div class="well well-lg">
      <form action="." method="post" class="form-horizontal">{% csrf_token %}
        <input type="hidden" name="next" value="{{ comment.get_absolute_url }
↪}">
        <div class="form-group">
          <div class="col-lg-offset-3 col-md-offset-3 col-lg-7 col-md-7">
            <input type="submit" name="submit" class="btn btn-primary" value=
↪"{% if already_liked_it %}{% trans 'Withdraw' %}{% else %}{% trans 'I␣
↪dislike it' %}{% endif %}"/>
            <a class="btn btn-default" href="{{ comment.get_absolute_url }}">
↪{% trans "cancel" %}</a>
          </div>
        </div>
      </form>
    </div>
  </div>
</div>
```

(continues on next page)

(continued from previous page)

```
{% endblock %}
```

One last change we need to do consist in adding the argument `with_participants` to the tag *`get_xtdcomment_tree`* in the blog detail template. The immediate effect of this argument is that in addition to the comment and the children of the comment, in each dictionary of the list retrieved by the template tag we will have the list of users who liked the comment and the list of users who disliked it:

```
[
    {
        'comment': comment_object,
        'children': [ list, of, child, comment, dicts ],
        'likedit': [user_who_liked_it_1, user_who_liked_it_2, ...],
        'dislikedit': [user_who_disliked_it_1, user_who_disliked_it_2, ...],
    },
    ...
]
```

Now we have all the changes ready, we can like/dislike comments to see the feature in action.

1.3 Demo projects

Django-comments-xtd comes with three demo projects:

1. **simple**: Single model with **non-threaded** comments
2. **simple_threads**: Single model with **threaded** comments up to level 2
3. **multiple**: Several models with comments, and a maximum thread level defined for each app.model pair.
4. **custom_comments**: Single model with comments provided by a new app that extends django-comments-xtd. Comments have a new `title` field. Find more details in *Customizing django-comments-xtd*.

Click here for a quick look at the examples directory in the repository.

1.3.1 Demo sites 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 install script and launch the dev server:

```
$ git clone git://github.com/danirus/django-comments-xtd.git
$ cd django-comments-xtd/django_comments_xtd/demos/[simple|simple_thread|multiple]
$ pip install ../../requirements.pip
$ sh ./install.sh (to syncdb, migrate and loaddata)
$ python manage.py runserver
```

By default:

- There's an `admin` user, with password `admin`
- Emails are sent to the `console.EmailBackend`. Comment out `EMAIL_BACKEND` in the settings module to send actual emails.

1.3.2 Simple demo site

The **simple** demo site is a project with just one application called **articles** with an **Article** model whose instances accept comments. The example features:

- Comments have to be confirmed by email before they hit the database.
- Users may request follow-up notifications.
- Users may cancel follow-up notifications by clicking on the mute link.

Follow the next steps to give them a try:

1. Visit http://localhost:8000/ and look at your articles' detail page.
2. Log out of the admin site to post comments, otherwise they will be automatically confirmed and no email will be sent.
3. When adding new articles in the admin interface be sure to tick the box *allow comments*, otherwise comments won't be allowed.
4. 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.
5. Click on the Mute link on the Follow-up notification email and send another comment.

1.3.3 Simple with threads

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 Multiple demo site

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):
        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 comes with 4 tags and two filters:

- Tag *get_xtdcomment_tree*
- Tag *get_xtdcomment_count*
- Tag get_last_xtdcomments
- Filter *xtd_comment_gravatar*
- Tag render_last_xtdcomments
- Filter *render_markup_comment*

To use any of them in your templates you first need to load them:

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

1.5.1 Get Xtdcomment Tree

Tag syntax:

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

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:

```
{
    'xtdcomment': 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*.

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 %}
```

1.5.2 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 %}
```

1.5.3 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 %}
```

1.5.4 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.

1.5.5 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 %}
```

1.5.6 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 is extendable in the same way as the django-contrib-comments framework. There are only three different details you have to bear in mind:

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',
  'django_comments_xtd',
  '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 at least the `comments/list.html` template to include the `title` field in the template.

Also change the template `comments/form.html` if you want to customize the way the comment form is displayed.

Both templates belong to the django-contrib-comments application.

1.7 Settings

To use django-comments-xtd it's necessary to declare the `COMMENTS_APP` setting:

```
COMMENTS_APP = "django_comments_xtd"
```

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

1.7.1 Maximum Thread Level

COMMENTS_XTD_MAX_THREAD_LEVEL - Maximum Thread Level

Optional

Indicate the maximum thread level for comments.

An example:

```
COMMENTS_XTD_MAX_THREAD_LEVEL = 8
```

Defaults to 0. What means threads are not permitted.

1.7.2 Maximum Thread Level per App.Model

COMMENTS_XTD_MAX_THREAD_LEVEL_BY_APP_MODEL - Maximum Thread Level per app.model basis

Optional

A dictionary with *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_MODEL = {
    'projects.release': 2,
    'blog.stories': 8, 'blog.quotes': 8,
    'blog.diarydetail': 0 # not required as it defaults to COMMENTS_XTD_MAX_THREAD_
↪LEVEL
}
```

1.7.3 Confirm Comment Post by Email

COMMENTS_XTD_CONFIRM_EMAIL - Confirm Comment Post by Email

Optional

This setting establishes whether a comment confirmation should be sent by email. If set to True a confirmation message is sent to the user with a link she has to click on. If the user is already authenticated the confirmation is not sent.

If is set to False the comment is accepted (unless your discard it by returning False when receiving the signal `comment_will_be_posted`, defined by the Django Comments Framework).

An example:

```
COMMENTS_XTD_CONFIRM_EMAIL = True
```

Defaults to True.

1.7.4 From Email Address

COMMENTS_XTD_FROM_EMAIL - From Email Address

Optional

This setting establishes the email address used in the *from* field when sending emails.

An example:

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

Defaults to `settings.DEFAULT_FROM_EMAIL`.

1.7.5 Comment Form Class

COMMENTS_XTD_FORM_CLASS - Form class to use when rendering comment forms.

Optional

A classpath to the form class that will be used for comments.

An example:

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

Defaults to *"django_comments_xtd.forms.XtdCommentForm"*.

1.7.6 Comment Model

COMMENTS_XTD_MODEL - Model to use

Optional

A classpath 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 Model Ordering

COMMENTS_XTD_LIST_ORDER - Field ordering in which comments are retrieve.

Optional

A tuple with field names, used as the `ordering` for the `XtdComment` mode.

Defaults to `('thread_id', 'order')`

1.7.8 Comment Markup Fallback Filter

COMMENTS_XTD_MARKUP_FALLBACK_FILTER - Default filter to use when rendering comments

Optional

Indicate 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'
```

Defaults to None.

1.7.9 Salt

COMMENTS_XTD_SALT - Extra key to salt the form

Optional

This setting establishes the ASCII string extra_key used by `signed.dumps` to salt the comment form hash. As `signed.dumps` docstring says, just in case you're worried that the NSA might try to brute-force your SHA-1 protected secret.

An example:

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

Defaults to an empty string.

1.7.10 Send HTML Email

COMMENTS_XTD_SEND_HTML_EMAIL - Enable/Disable HTML email messages

Optional

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 email messages will be sent only in Text format.

An example:

```
COMMENTS_XTD_SEND_HTML_EMAIL = True
```

Defaults to True.

1.7.11 Threaded Emails

COMMENTS_XTD_THREADED_EMAILS - Enable/Disable sending emails in separeate threads

Optional

For low traffic websites sending emails in separate threads is a fine solution. However, for medium to high traffic websites such overhead could be reduce by using other solutions, like a Celery application.

An example:

```
COMMENTS_XTD_THREADED_EMAILS = True
```

Defaults to True.

1.8 Templates

List of template files used by django-comments-xtd.

django_comments_xtd/email_confirmation_request.(html|txt) (included) Comment confirmation email sent to the user when she clicks on "Post" to send the comment. The user should click on the link in the message to confirm the comment. If the user is authenticated the message is not sent, as the comment is directly approved.

django_comments_xtd/discarded.html (included) Rendered if any receiver of the signal `confirmation_received` returns False. Tells the user the comment has been discarded.

django_comments_xtd/email_followup_comment.(html|txt) (included) Email message sent when there is a new comment following up the user's. To receive this email the user must tick the box *Notify me of follow up comments via email*.

django_comments_xtd/max_thread_level.html (included) Rendered when a nested comment is sent with a thread level over the maximum thread level allowed. The template in charge of rendering the list of comments can make use of `comment.allow_thread` (True when the comment accepts nested comments) to avoid adding the link "Reply" on comments that don't accept nested comments. See the simple_threads demo site, template `comment/list.html` to see a use example of `comment.allow_thread`.

django_comments_xtd/comment.html (included) Rendered when a logged in user sent a comment via Ajax. The comment gets rendered immediately. JavaScript client side code still has to handle the response.

django_comments_xtd/posted.html Rendered when a not logged-in user sends a comment. Django-comments-xtd try first to find the template in its own template directory, `django_comments_xtd/posted.html`. If it doesn't exist, it will use the template in Django Comments Framework directory: `comments/posted.html`. Look at the demo sites for sample uses.

django_comments_xtd/reply.html (included) Rendered when a user clicks on the *reply* link of a comment. Reply links are created with `XtdComment.get_reply_url` method.

django_comments_xtd/muted.html (included) Rendered when a user clicks on the *mute link* of a follow-up email message.

Python Module Index

d

Index

N

P

Q

R

S

T

X