django-comments-xtd Documentation

Release 2.7.2

Daniel Rus Morales

Sep 08, 2020

Contents

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

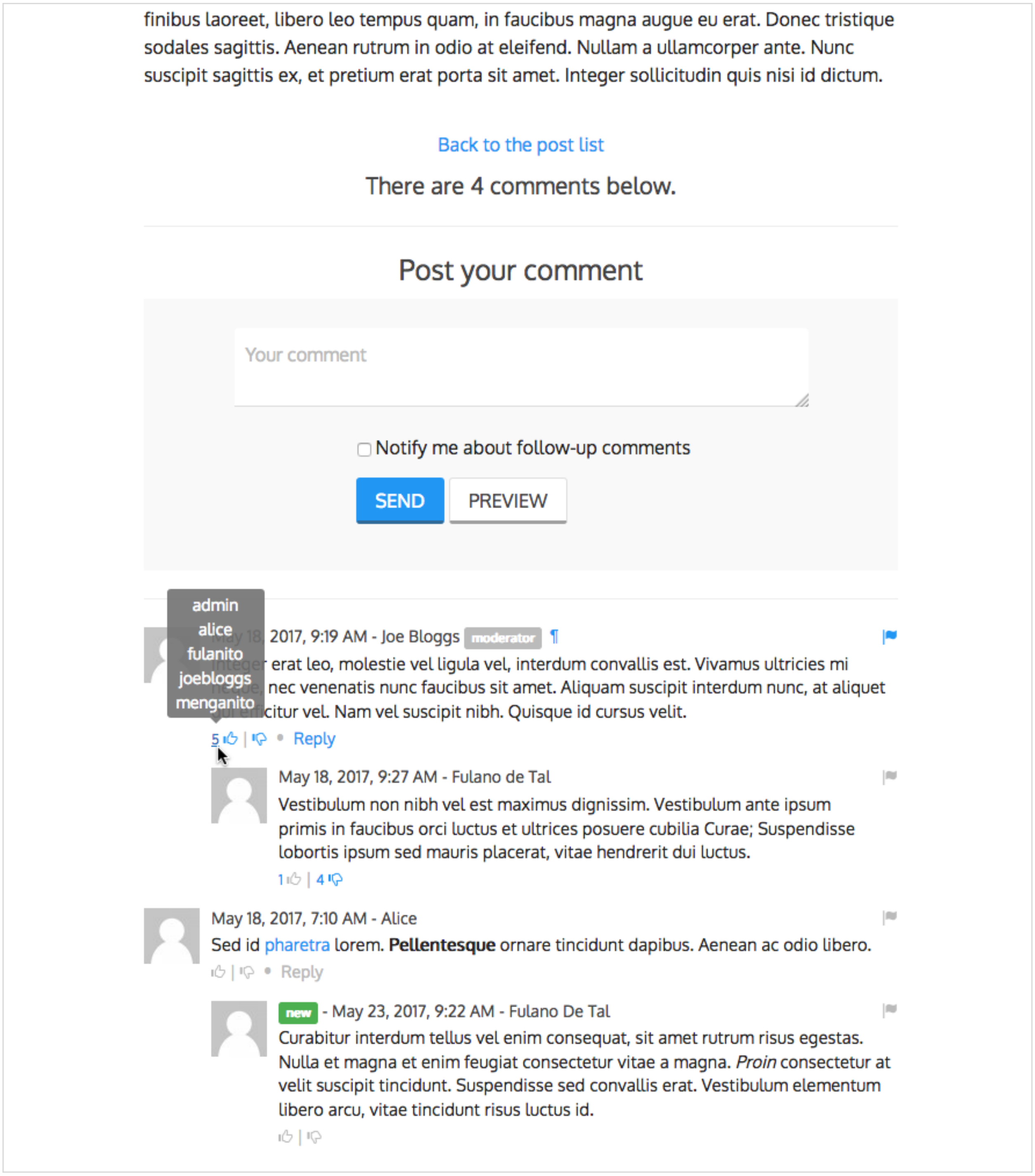
Note: This documentation represents the current version, v2.7.2, of django-comments-xtd. For old versions of the documentation:

- v2.6.2: https://django-comments-xtd.readthedocs.io/en/2.6.2/
- v2.5.1: https://django-comments-xtd.readthedocs.io/en/2.5.1/
- v2.4.3: https://django-comments-xtd.readthedocs.io/en/2.4.3/
- v2.3.1: https://django-comments-xtd.readthedocs.io/en/2.3.1/
- v2.2.1: https://django-comments-xtd.readthedocs.io/en/2.2.1/
- v2.1.0: https://django-comments-xtd.readthedocs.io/en/2.1.0/
- v2.0.10: https://django-comments-xtd.readthedocs.io/en/2.0.10/
- v1.7.1: https://django-comments-xtd.readthedocs.io/en/1.7.1/
- v1.6.7: https://django-comments-xtd.readthedocs.io/en/1.6.7/
- v1.5.1: https://django-comments-xtd.readthedocs.io/en/1.5.1/

CHAPTER 1

Features

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



finibus laoreet, libero leo tempus quam, in faucibus magna augue eu erat. Donec tristique sodales sagittis. Aenean rutrum in odio at eleifend. Nullam a ullamcorper ante. Nunc suscipit sagittis ex, et pretium erat porta sit amet. Integer sollicitudin quis nisi id dictum.

Back to the post list

There are 4 comments below.

Post your comment

Your comment

Notify me about follow-up comments

SEND PREVIEW

admin
alice
fulanito
joebloggs
menganito

May 18, 2017, 9:19 AM - Joe Bloggs moderator ¶

Integer erat leo, molestie vel ligula vel, interdum convallis est. Vivamus ultricies mi neque, nec venenatis nunc faucibus sit amet. Aliquam suscipit interdum nunc, at aliquet dui efficitur vel. Nam vel suscipit nibh. Quisque id cursus velit.

5 | • Reply

May 18, 2017, 9:27 AM - Fulano de Tal

Vestibulum non nibh vel est maximus dignissim. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Suspendisse lobortis ipsum sed mauris placerat, vitae hendrerit dui luctus.

1 | 4

May 18, 2017, 7:10 AM - Alice

Sed id pharetra lorem. **Pellentesque** ornare tincidunt dapibus. Aenean ac odio libero.

| • Reply

new - May 23, 2017, 9:22 AM - Fulano De Tal

Curabitur interdum tellus vel enim consequat, sit amet rutrum risus egestas. Nulla et magna et enim feugiat consectetur vitae a magna. *Proin* consectetur at velit suscipit tincidunt. Suspendisse sed convallis erat. Vestibulum elementum libero arcu, vitae tincidunt risus luctus id.

|

CHAPTER 2

Getting started

Start with these documents to get you up and running:

2.1 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. To allow a quick start django-comments-xtd makes use of twitter-bootstrap. From django-comments-xtd v2.3 on it uses Twitter-Bootstrap v4. From django-comments-xtd v1.7.1 to v2.2 it uses Twitter-Bootstrap v3. 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.

2.2 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

2.2.1 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.

2.2.2 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/raw/master/
↪example/tutorial.tar.gz
(venv)$ tar -xvzf tutorial.tar.gz
(venv)$ cd tutorial
```

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

- A **blog** app with a **Post** model. It uses two generic class-based views to list the posts and show a post in detail.
- The **templates** directory, with a **base.html** and **home.html**, and the templates for the blog app: **blog/post_list.html** and **blog/post_detail.html**.
- The **static** directory with a **css/bootstrap.min.css** file (this file is a static asset available, when the app is installed, under the path **django_comments_xtd/css/bootstrap.min.css**).
- The **tutorial** directory containing the **settings** and **urls** modules.
- And a **fixtures** directory with data files to create the *admin* superuser (with *admin* password), the default site and some blog posts.

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

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

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

Note: Remember to implement the *get_absolute_url* in the model class whose objects you want to receive comments, like the class *Post* in this tutorial. It is so because the permanent URL of each comment uses the *shortcut* view of *django.contrib.contenttypes* which in turn uses the *get_absolute_url* method.

2.2.3 Configuration

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

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

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

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

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

```
from django.urls import include, path

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

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

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

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

Comment confirmation

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

It is so because django-comments-xtd does not store comments in the server until 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_EMAIL* to `False`.

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

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

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

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

2.2.4 Comments tags

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

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

Now we will change the blog post detail template to:

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

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

```
{% get_comment_count for object as comment_count %}
<div class="py-4 text-center">
  <a href="{% url 'blog:post-list' %}">Back to the post list</a>
  &nbsp;&sdot;&nbsp;
  {{ comment_count }} comment{{ comment_count|pluralize }}
  ha{{ comment_count|pluralize:"s,ve" }} been posted.
</div>
```

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

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

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

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

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

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

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

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

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

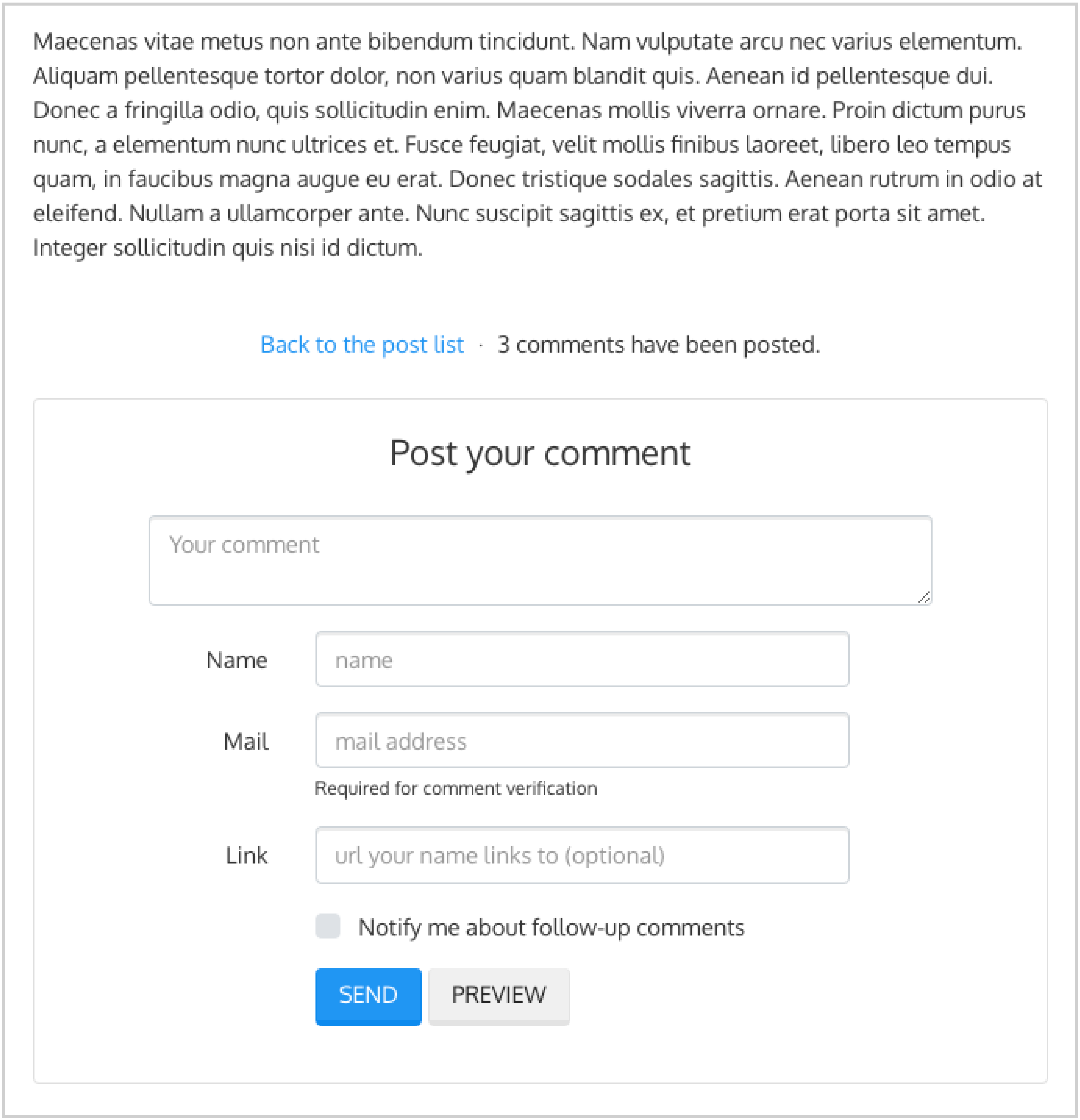
```
{% if object.allow_comments %}
<div class="card card-block mb-5">
  <div class="card-body">
    <h4 class="card-title text-center pb-3">Post your comment</h4>
      {% render_comment_form for object %}
  </div>
</div>
{% endif %}
```

Note: The `{% if object.allow_comments %}` and corresponding `{% endif %}` are not necessary in your code. I use it in this tutorial (and in the demo sites) as a way to disable comments whenever the author of a blog post decides so. It has been mentioned here too.

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

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

...
    {% for object in object_list %}
    ...
    {% get_comment_count for object as comment_count %}
    <p class="date">Published {{ object.publish }}
      {% if comment_count %}
      &sdot;&nbsp;{{ comment_count }} comment{{ comment_count|pluralize }}
      {% endif %}
    </p>
    ...
    {% endfor %}
```

Now we are ready to send comments. If you are logged in 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.


Maecenas vitae metus non ante bibendum tincidunt. Nam vulputate arcu nec varius elementum. Aliquam pellentesque tortor dolor, non varius quam blandit quis. Aenean id pellentesque dui. Donec a fringilla odio, quis sollicitudin enim. Maecenas mollis viverra ornare. Proin dictum purus nunc, a elementum nunc ultrices et. Fusce feugiat, velit mollis finibus laoreet, libero leo tempus quam, in faucibus magna augue eu erat. Donec tristique sodales sagittis. Aenean rutrum in odio at eleifend. Nullam a ullamcorper ante. Nunc suscipit sagittis ex, et pretium erat porta sit amet. Integer sollicitudin quis nisi id dictum.

Back to the post list · 3 comments have been posted.

Post your comment

	Your comment
Name	name
Mail	mail address
	Required for comment verification
Link	url your name links to (optional)
	Notify me about follow-up comments

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.

2.2.5 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 you would prefer a different approach. Django Comments Framework comes with moderation capabilities included upon which you can build your own comment filtering.

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

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

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

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

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

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

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

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

...

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


moderator.register(Post, PostCommentModerator)
```

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

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

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

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

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

Disallow black listed domains

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

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

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

```
COMMENTS_XTD_CONFIRM_EMAIL = False
```

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

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

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

moderator.register(Post, PostCommentModerator)
```

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

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

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

Moderate on bad words

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

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

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

```
COMMENTS_XTD_CONFIRM_EMAIL = False
```

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

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

...
```

(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 lowcase_comment.find(badword) > -1:
                    return True
            else:
                lastindex = -1
                for subword in badword:
                    if subword in msg:
                        if lastindex > -1:
                            if msg[subword] == (lastindex + 1):
                                lastindex = msg[subword]
                        else:
                            lastindex = msg[subword]
                    else:
                        break
                if msg.get(badword[-1]) and msg[badword[-1]] == lastindex:
                    return True
        return super(PostCommentModerator, self).moderate(comment,
                                                          content_object,
                                                          request)

moderator.register(Post, PostCommentModerator)
```

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

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

2.2.6 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 nest inside each other up to the maximum thread level. A comment in a the thread level below the *COMMENTS_XTD_MAX_THREAD_LEVEL* can show a **Reply** link that allows users to send nested comments.

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

We can make use of two template tags, *render_xtdcomment_tree* and *get_xtdcomment_tree*. The former renders a template with the comments while the latter put the comments in a nested data structure in the context of the template.

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

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

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

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

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

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

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

{% block content %}
<div class="pb-3">
  <h1 class="page-header text-center">{{ object.title }}</h1>
  <p class="small text-center">{{ object.publish|date:"l, j F Y" }}</p>
</div>
<div>
  {{ object.body|linebreaks }}
</div>

{% get_comment_count for object as comment_count %}
<div class="py-4 text-center>
  <a href="{% url 'blog:post-list' %}">Back to the post list</a>
  &nbsp;&sdot;&nbsp;
  {{ comment_count }} comment{{ comment_count|pluralize }}
  ha{{ comment_count|pluralize:"s,ve"}} 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 %}
```

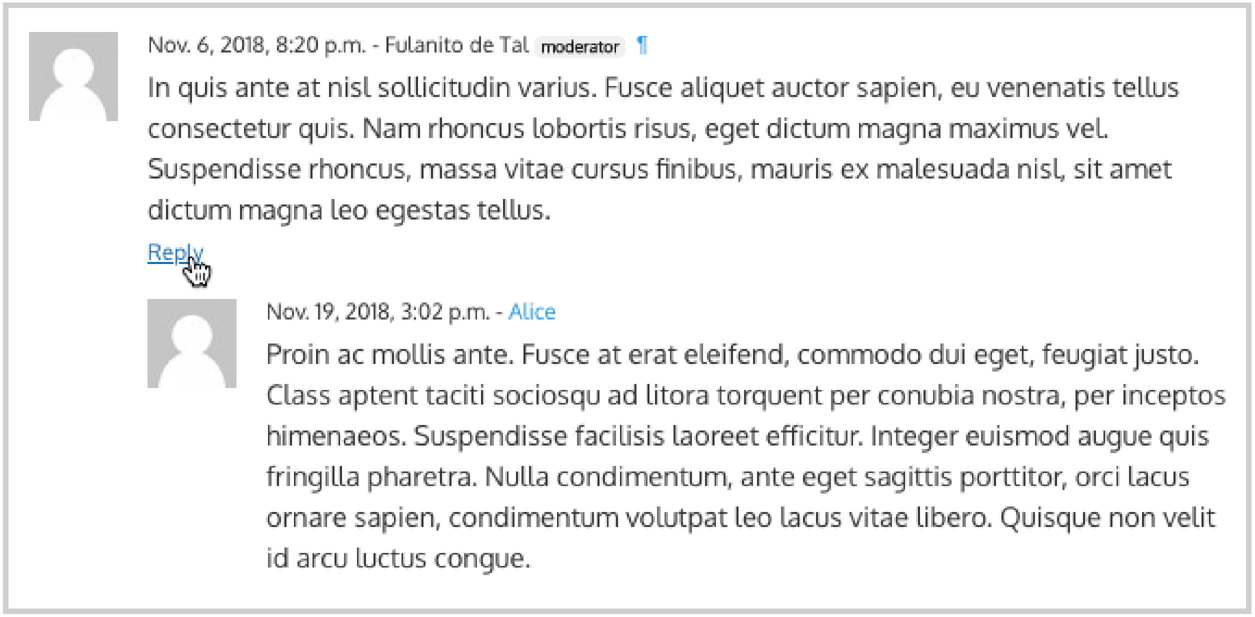
(continues on next page)

(continued from previous page)

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

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

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


Nov. 6, 2018, 8:20 p.m. - Fulanito de Tal moderator ¶

In quis ante at nisl sollicitudin varius. Fusce aliquet auctor sapien, eu venenatis tellus consectetur quis. Nam rhoncus lobortis risus, eget dictum magna maximus vel. Suspendisse rhoncus, massa vitae cursus finibus, mauris ex malesuada nisl, sit amet dictum magna leo egestas tellus.

Reply

Nov. 19, 2018, 3:02 p.m. - Alice

Proin ac mollis ante. Fusce at erat eleifend, commodo dui eget, feugiat justo. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Suspendisse facilisis laoreet efficitur. Integer euismod augue quis fringilla pharetra. Nulla condimentum, ante eget sagittis porttitor, orci lacus ornare sapien, condimentum volutpat leo lacus vitae libero. Quisque non velit id arcu luctus congue.

Different max thread levels

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

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

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

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

2.2.7 Flags

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

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

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

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

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

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

Commenting options

As of version 2.0 django-comments-xtd has a new setting *COMMENTS_XTD_APP_MODEL_OPTIONS* that must be used to allow comment flagging. The purpose of it is to give an additional level of control about what actions users can perform on comments: flag them as inappropriate, like/dislike them, retrieve the list of users who liked/disliked them, and whether visitors can post comments or only registered users can do it.

It defaults to:

```
COMMENTS_XTD_APP_MODEL_OPTIONS = {
    'default': {
        'allow_flagging': False,
        'allow_feedback': False,
        'show_feedback': False,
        'who_can_post': 'all'  # Valid values: 'all', users'
    }
}
```

We will go through the first three options in the following sections. As for the last option, *who_can_post*, I recommend you to read the special use case *Only signed in users can comment*, that explains the topic in depth.

Removal suggestion

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

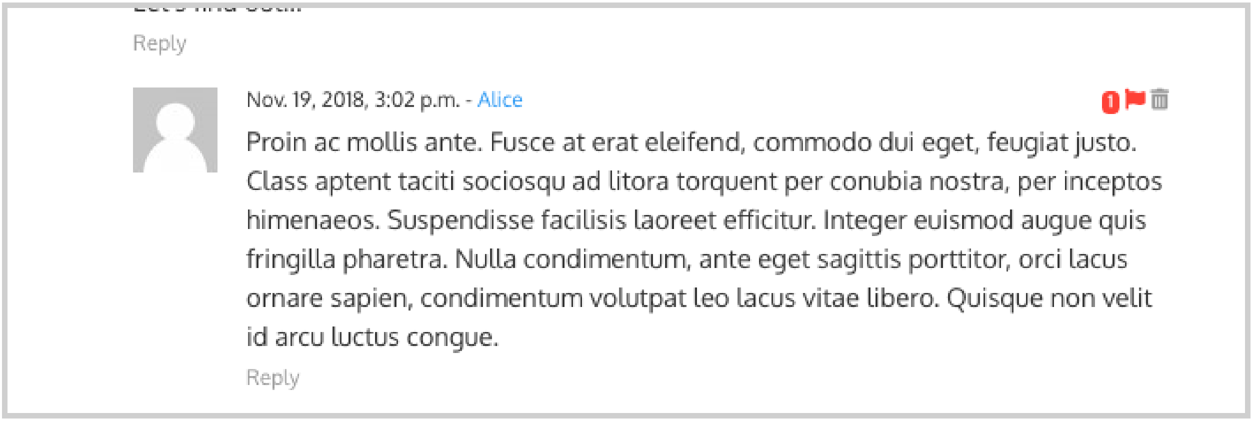
```
...
<ul class="media-list">
  {% render_xtdcomment_tree for object allow_flagging %}
</ul>
```

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

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

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

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


Reply

Nov. 19, 2018, 3:02 p.m. - Alice

Proin ac mollis ante. Fusce at erat eleifend, commodo dui eget, feugiat justo. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Suspendisse facilisis laoreet efficitur. Integer euismod augue quis fringilla pharetra. Nulla condimentum, ante eget sagittis porttitor, orci lacus ornare sapien, condimentum volutpat leo lacus vitae libero. Quisque non velit id arcu luctus congue.

Reply

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

Getting notifications

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

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

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

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

```
from django_comments_xtd.moderation import moderator, XtdCommentModerator

...
class PostCommentModerator(XtdCommentModerator):
```

(continues on next page)

(continued from previous page)

```
    removal_suggestion_notification = True

moderator.register(Post, PostCommentModerator)
```

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

Liked it, Disliked it

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

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

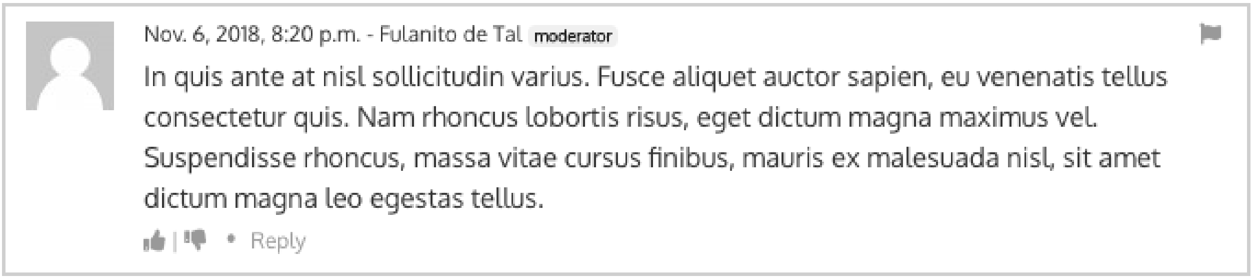
In this section we make changes to give our users the capacity to like or dislike comments. Following the same pattern as with the removal flag, enabling like/dislike buttons is about adding an argument to the `render_xtdcomment_tree`, the argument `allow_feedback`. Edit the `blog/post_detail.html` template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

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

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

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

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


Nov. 6, 2018, 8:20 p.m. - Fulanito de Tal moderator

In quis ante at nisl sollicitudin varius. Fusce aliquet auctor sapien, eu venenatis tellus consectetur quis. Nam rhoncus lobortis risus, eget dictum magna maximus vel. Suspendisse rhoncus, massa vitae cursus finibus, mauris ex malesuada nisl, sit amet dictum magna leo egestas tellus.

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

Show the list of users

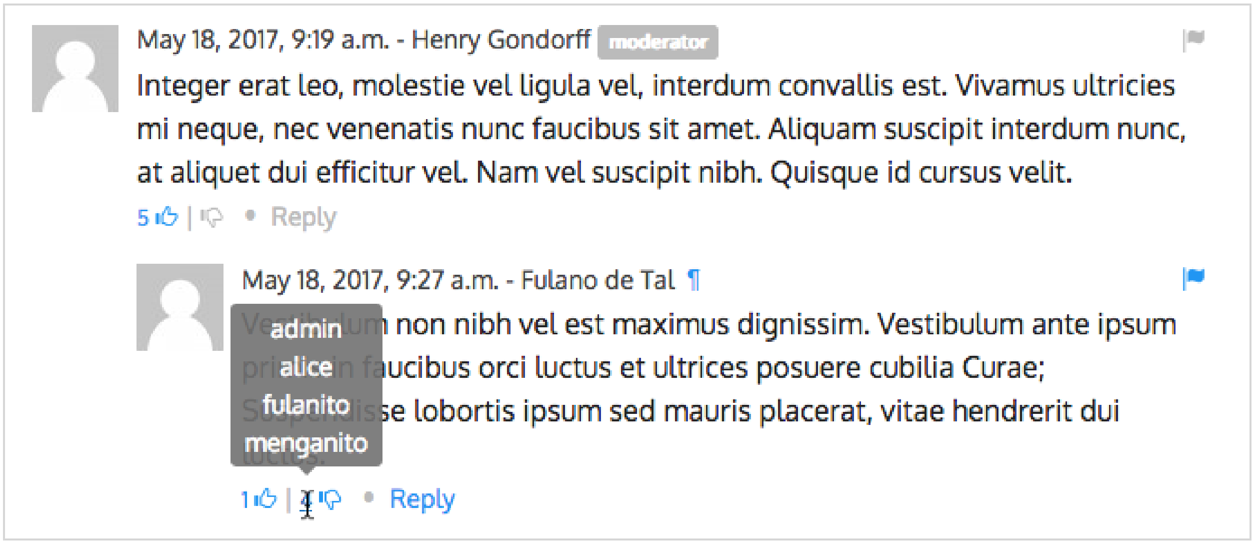
With the like/dislike buttons enabled we might as well consider to display the users who actually liked/disliked comments. Again addind an argument to the `render_xtdcomment_tree` will enable the feature. Change the `blog/post_detail.html` and add the argument `show_feedback` to the template tag:

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

{% block extra-js %}
<script
  src="https://code.jquery.com/jquery-3.3.1.min.js"
  crossorigin="anonymous"></script>
<script
  src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.
↪min.js"
  integrity="sha384-ZMP7rVo3mIykV+2+9J3UJ46jBk0WLaUAdn689aCwoqbBJiSnjAK/
↪l8WvCWPIPm49"
  crossorigin="anonymous"></script>
<script
  src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js
↪"
  integrity="sha384-ChfqqxuZUCnJSK3+MXmPNIyE6ZbWh2IMqE241rYiqJxyMiZ6OW/
↪JmZQ5stwEULTy"
  crossorigin="anonymous"></script>
<script>
  $(function() {
    $('[data-toggle="tooltip"]').tooltip({html: true});
  });
</script>
{% endblock %}
```

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

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

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


May 18, 2017, 9:19 a.m. - Henry Gondorff moderator

Integer erat leo, molestie vel ligula vel, interdum convallis est. Vivamus ultricies mi neque, nec venenatis nunc faucibus sit amet. Aliquam suscipit interdum nunc, at aliquet dui efficitur vel. Nam vel suscipit nibh. Quisque id cursus velit.

5 | • Reply

May 18, 2017, 9:27 a.m. - Fulano de Tal ¶

Vestibulum non nibh vel est maximus dignissim. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Suspendisse lobortis ipsum sed mauris placerat, vitae hendrerit dui luctus.

admin
alice
fulanito
menganito

1 | 4 • Reply

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

2.2.8 Markdown

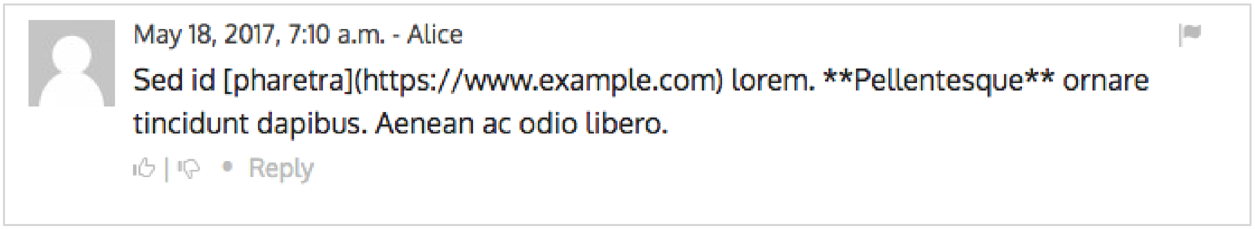
In versions prior to 2.0 django-comments-xtd required the installation of django-markup as a dependency. There was also a specific template filter called `render_markup_comment` to help rendering comment's content in the markup language of choice.

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

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

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

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


May 18, 2017, 7:10 a.m. - Alice

Sed id [pharetra](https://www.example.com) lorem. **Pellentesque** ornare tincidunt dapibus. Aenean ac odio libero.

| • Reply

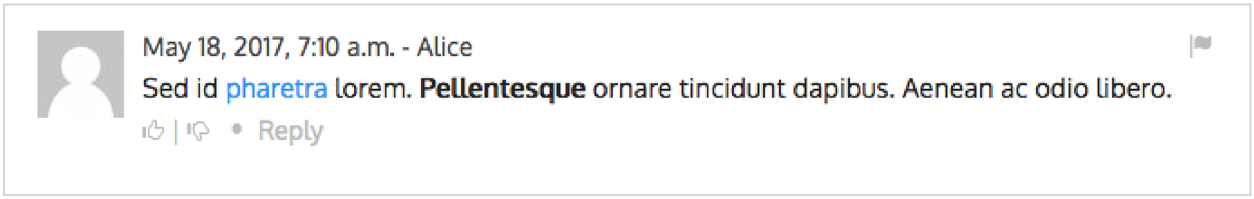
Now we will install django-markdown2, and create the template directory and the template file:

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

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

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

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


May 18, 2017, 7:10 a.m. - Alice

Sed id pharetra lorem. **Pellentesque** ornare tincidunt dapibus. Aenean ac odio libero.

2.2.9 JavaScript plugin

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

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

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

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

Enable Web API

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

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

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

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

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

Enable app.model options

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

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

The i18n JavaScript Catalog

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

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

```
from django.views.i18n import JavaScriptCatalog

urlpatterns = [
    ...
    path(r'jsi18n/', JavaScriptCatalog.as_view(), name='javascript-catalog'),
]
```

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

Load the plugin

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

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

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

{% block content %}
<div class="pb-3">
  <h1 class="text-center">{{ object.title }}</h1>
  <p class="small text-center">{{ object.publish|date:"l, j F Y" }}</p>
</div>
<div>
  {{ object.body|linebreaks }}
</div>

<div class="py-4 text-center">
  <a href="{% url 'blog:post-list' %}">Back to the post list</a>
</div>

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

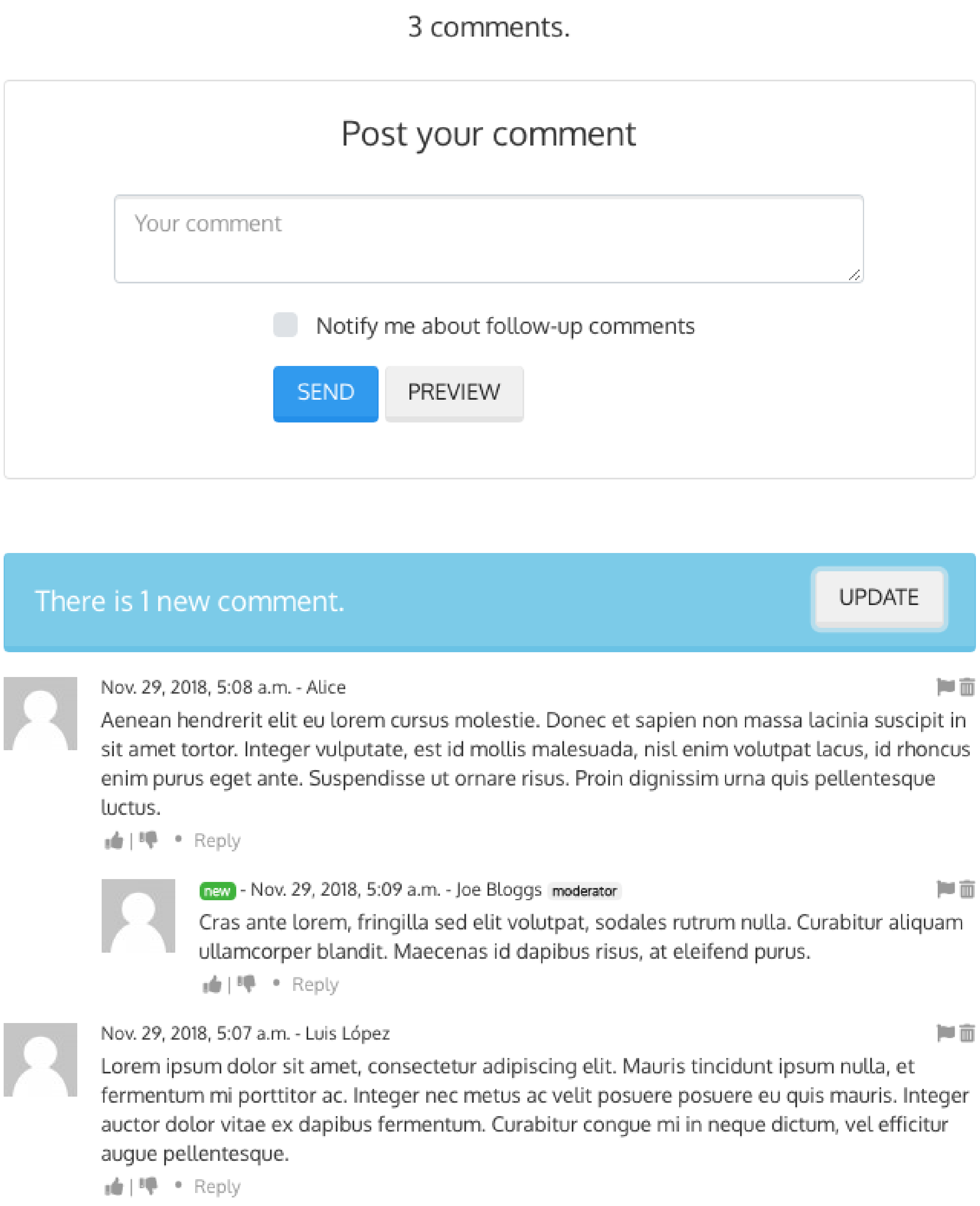
(continues on next page)

(continued from previous page)

```
{% block extra-js %}
<script crossorigin src="https://unpkg.com/react@16/umd/react.production.min.
↪js"></script>
<script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.
↪production.min.js"></script>
<script>
 window.comments_props = {% get_commentbox_props for object %};
 window.comments_props_override = {
     allow_comments: {%if object.allow_comments%}true{%else%}false{%endif%},
     allow_feedback: true,
     show_feedback: true,
     allow_flagging: true,
     polling_interval: 5000  // In milliseconds.
 };
</script>
<script
  src="https://code.jquery.com/jquery-3.3.1.min.js"
  crossorigin="anonymous"></script>
<script
  src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.
↪min.js"
  integrity="sha384-ZMP7rVo3mIykV+2+9J3UJ46jBk0WLaUAdn689aCwoqbBJiSnjAK/
↪l8WvCWPIPm49"
  crossorigin="anonymous"></script>
<script
  src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js
↪"
  integrity="sha384-ChfqqxuZUCnJSK3+MXmPNIyE6ZbWh2IMqE241rYiqJxyMiZ6OW/
↪JmZQ5stwEULTy"
  crossorigin="anonymous"></script>
<script
  type="text/javascript"
  src="{% url 'javascript-catalog' %}"></script>
<script src="{% static 'django_comments_xtd/js/vendor~plugin-2.7.2.js' %}"></
↪script>
<script src="{% static 'django_comments_xtd/js/plugin-2.7.2.js' %}"></script>
<script>
$(function() {
  $('[data-toggle="tooltip"]').tooltip({html: true});
});
</script>
{% endblock %}
```

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

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



Nov. 29, 2018, 5:08 a.m. - Alice

Aenean hendrerit elit eu lorem cursus molestie. Donec et sapien non massa lacinia suscipit in sit amet tortor. Integer vulputate, est id mollis malesuada, nisl enim volutpat lacus, id rhoncus enim purus eget ante. Suspendisse ut ornare risus. Proin dignissim urna quis pellentesque luctus.

new - Nov. 29, 2018, 5:09 a.m. - Joe Bloggs moderator

Cras ante lorem, fringilla sed elit volutpat, sodales rutrum nulla. Curabitur aliquam ullamcorper blandit. Maecenas id dapibus risus, at eleifend purus.

Nov. 29, 2018, 5:07 a.m. - Luis López

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris tincidunt ipsum nulla, et fermentum mi porttitor ac. Integer nec metus ac velit posuere posuere eu quis mauris. Integer auctor dolor vitae ex dapibus fermentum. Curabitur congue mi in neque dictum, vel efficitur augue pellentesque.

2.2.10 Final notes

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

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

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

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

2.3 Demo projects

There are three example projects available within django-comments-xtd's GitHub repository.

The **simple project** provides non-threaded comment support to articles. It's an only-backend project, meant as a test case of the basic features (confirmation by mail, follow-up notifications, mute link).

The **custom project** provides threaded comment support to articles using a new comment class that inherits from django-comments-xtd's. The new comment model adds a **title** field to the **XtdComment** class. Find more details in *Customizing django-comments-xtd*.

The **comp project** provides comments to an Article model and a Quote model. Comments for Quotes show how to use django-comments-xtd as a pure Django backend application. However comments for Articles illustrate how to use the app in combination with the provided JavaScript plugin. The project allows nested comments and defines the maximum thread level on per app.model basis. It uses moderation, removal suggestion flag, like/dislike flags, and list of users who liked/disliked comments.

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

Table of Contents

2.3.1 Setup

The recommended way to run a demo site is within 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/
(venv)$ python setup.py install
```

(continues on next page)

(continued from previous page)

```
(venv)$ npm install
(venv)$ node_modules/webpack/bin/webpack.js -p
(venv)$ cd django_comments_xtd
(venv)$ django-admin compilemessages -l fi
(venv)$ django-admin compilemessages -l fr
(venv)$ django-admin compilemessages -l es
(venv)$ cd ../example/[simple|custom|comp]
(venv)$ pip install -r requirements.txt
(venv)$ python manage.py migrate
(venv)$ python manage.py loaddata ../fixtures/auth.json
(venv)$ python manage.py loaddata ../fixtures/sites.json
(venv)$ python manage.py loaddata ../fixtures/articles.json
(venv)$ # The **comp** example project needs quotes.json too:
(venv)$ python manage.py loaddata ../fixtures/quotes.json
(venv)$ python manage.py runserver
```

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

Fixtures provide:

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

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

2.3.2 Simple project

The simple example project features:

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

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

Some hints:

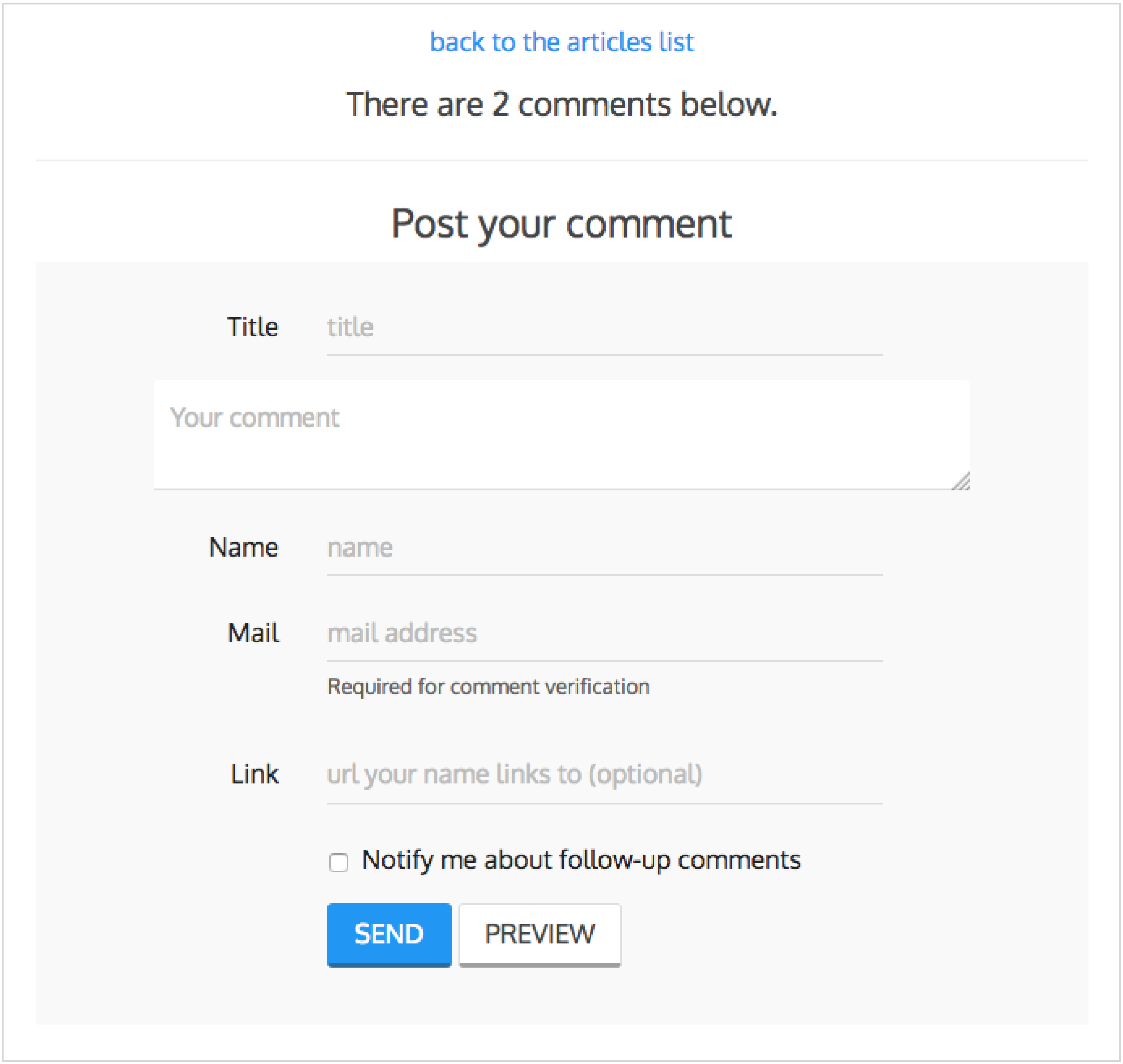
- Log out from the admin site to post comments, otherwise they will be automatically confirmed and no email will be sent.
- When adding new articles in the admin interface be sure to tick the box *allow comments*, otherwise comments won't be allowed.
- Send new comments with the Follow-up box ticked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

- Click on the Mute link on the Follow-up notification email and send another comment. You will not receive further notifications.

2.3.3 Custom project

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

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


back to the articles list

There are 2 comments below.

Post your comment

Title	title
	Your comment
Name	name
Mail	mail address
	Required for comment verification
Link	url your name links to (optional)
	☐ Notify me about follow-up comments

2.3.4 Comp project

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

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

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

Features:

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

Threaded comments

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

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

render_xtdcomment_tree

By using the *render_xtdcomment_tree* templatetag, *quote_detail.html*, show the tree of comments posted. Addind the argument *allow_feedback* users can send like/dislike feedback. Adding the argument *show_feedback* allow visitors see other users like/dislike feedback. And adding *allow_flagging* allow users flag comments for removal.

render_last_xtdcomments

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

JavaScript plugin

As opposed to the Quote model, the Article model receives comments via the provided JavaScript plugin. Check the *JavaScript plugin* page to know more.

CHAPTER 3

Advanced Use

Once you've got django-comments-xtd working, you may want to know more about specific features, or check out the use cases to see how others customize it.

3.1 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.

3.1.1 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).

3.1.2 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.
- **should_request_be_authorized**: Sent before the data in the form in a web API post comment request is validated. A receiver returning *True* will suffice to automatically add valid values to the CommentSecurityForm fields *timestamp* and *security_hash*. The intention is to combine a receiver with a django-rest-framework authentication class, and return *True* when the *request.auth* is not *None*.

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.

3.1.3 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.

3.2 Web API

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

There are 5 methods available to perform the following actions:

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

Finally there is the ability to generate a view action in `django_comments_xtd.api.frontend` to return the commentbox props as used by the *JavaScript plugin* plugin for use with an existing django-rest-framework project.

Table of Contents

3.2.1 Post a new comment

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

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

Meaning of the HTTP Response codes:

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

Note: Up until v2.6 fields `timestamp` and `security_hash`, related with the CommentSecurityForm, had to be provided in the post request. As of v2.7 it is possible to use a django-rest-framework's authentication class in combination with django-comments-xtd's signal `should_request_be_authorized` (*Signal and receiver*) to automatically pass the CommentSecurityForm validation.

Authorize the request

As pointed out in the note above, django-comments-xtd notifies receivers of the signal `should_request_be_authorized` to give the request the chance to pass the *CommentSecurityForm* validation. When a receiver returns `True`, the form automatically receives valid values for the `timestamp` and `security_hash` fields, and the request continues its processing.

These two fields, `timestamp` and `security_hash`, are part of the frontline against spam in django-comments. In a classic backend driven request-response cycle these two fields received their values during the GET request, where the comment form is rendered via the Django template.

However, when using the web API there is no such previous GET request, and thus both fields can in fact be ignored. In such cases, in order to enable some sort of spam control, the request can be authenticated via the Django REST Framework, what in combination with a receiver of the `should_request_be_authorized` signal has the effect of **authorizing** the POST request.

Example of authorization

In this section we go through the changes that will enable posting comments via the web API in the *Simple project*. We have to:

1. Modify the settings module.
2. Modify the urls module to allow login and logout via DRF's api-auth.
3. Create a new authentication class, in this case it will be an authentication scheme based on DRF's Custom authentication, but you could use any other one.
4. Create a new receiver function for the signal `should_request_be_authorized`.
5. Post a test comment as a visitor.
6. Post a test comment as a signed in user.

Modify the settings module

We will modify the `simple/settings.py` module to add `rest_framework` to `INSTALLED_APPS`. In addition we will create a custom setting that will be used later in the receiver function for the signal `should_request_be_authorized`. I call the setting `MY_DRF_AUTH_TOKEN`. And we will also add Django Rest Framework settings to enable request authentication.

Append the code to your `simple/settings.py` module:

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

# import os, binascii; binascii.hexlify(os.urandom(20)).decode()
MY_DRF_AUTH_TOKEN = "08d9fd42468aebbb8087b604b526ff0821ce4525"

REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.SessionAuthentication',
        'simple.apiauth.APIRequestAuthentication'
    ]
}
```

Modify the urls module

In order to send comments as a logged in user we will first login using the end point provided by Django REST Framework's urls module. Append the following to the `urlpatterns` in `simple/urls.py`:

```
urlpatterns = [
    ...

    re_path(r'^api-auth/', include('rest_framework.urls',
                              namespace='rest_framework')),
]
```

Create a new authentication class

In this step we create a class to validate that the request has a valid Authorization header. We follow the instructions about how to create a Custom authentication scheme in the Django REST Framework documentation.

In the particular case of this class we don't want to authenticate the user but merely the request. To authenticate the user we added the class `rest_framework.authentication.SessionAuthentication` to the **DEFAULT_AUTHENTICATION_CLASSES** of the **REST_FRAMEWORK** setting. So once we read the auth token we will return a tuple with an **AnonymousUser** instance and the content of the token read.

Create the module `simple/apiauth.py` with the following content:

```python
from django.contrib.auth.models import AnonymousUser

from rest_framework import HTTP_HEADER_ENCODING, authentication, exceptions


class APIRequestAuthentication(authentication.BaseAuthentication):
  def authenticate(self, request):
    auth = request.META.get('HTTP_AUTHORIZATION', b'')
    if isinstance(auth, str):
      auth = auth.encode(HTTP_HEADER_ENCODING)

    pieces = auth.split()
    if not pieces or pieces[0].lower() != b'token':
      return None

    if len(pieces) == 1:
      msg = _("Invalid token header. No credentials provided.")
      raise exceptions.AuthenticationFailed(msg)
    elif len(pieces) > 2:
      msg = _("Invalid token header."
          "Token string should not contain spaces.")
      raise exceptions.AuthenticationFailed(msg)

    try:
      auth = pieces[1].decode()
    except UnicodeError:
      msg = _("Invalid token header. "
          "Token string should not contain invalid characters.")

    return (AnonymousUser(), auth)
```

The class doesn't validate the token. We will do it with the receiver function in the next section.

Create a receiver for `should_request_be_authorized`

Now let's create the receiver function. The receiver function will be called when the comment is posted, from the validate method of the **WriteCommentSerializer**. If the receiver returns True the request is considered authorized.

Append the following code to the `simple/articles/models.py` module:

```python
from django.dispatch import receiver
from django_comments_xtd.signals import should_request_be_authorized

[...]
```

(continues on next page)

(continued from previous page)

```
@receiver(should_request_be_authorized)
def my_callback(sender, comment, request, **kwargs):
    if (
        (request.user and request.user.is_authenticated) or
        (request.auth and request.auth == settings.MY_DRF_AUTH_TOKEN)
    ):
        return True
```

The left part of the *if* is True when the `rest_framework.authentication.SessionAuthentication` recognizes the user posting the comment as a signed in user. However if the user sending the comment is a mere visitor and the request contains a valid **Authorization** token, then our class `simple.apiauth.APIRequestAuthentication` will have put the auth token in the request. If the auth token contains the value given in the setting **MY_DRF_AUTH_TOKEN** we can considered the request authorized.

Post a test comment as a visitor

Now with the previous changes in place launch the Django development server and let's try to post a comment using the web API.

These are the fields that have to be sent:

- **content_type**: A string with the content_type ie: `content_type="articles.article"`.
- **object_pk**: The object ID we are posting the comment to.
- **name**: The name of the person posting the comment.
- **email**: The email address of the person posting the comment. It's required when the comment has to be confirmed via email.
- **followup**: Boolean to indicate whether the user wants to receive follow-up notification via email.
- **reply_to**: When threading is enabled, reply_to is the comment ID being responded with the comment being sent. If comments are not threaded the reply_to must be 0.
- **comment**: The content of the comment.

I will use the excellent HTTPie command line client:

```
$ http POST http://localhost:8000/comments/api/comment/ \
       'Authorization:Token 08d9fd42468aebbb8087b604b526ff0821ce4525' \
       content_type="articles.article" object_pk=1 name="Joe Bloggs" \
       followup=false reply_to=0 email="joe@bloggs.com" \
       comment="This is the body, the actual comment..."

HTTP/1.1 204 No Content
Allow: POST, OPTIONS
Content-Length: 2
Content-Type: application/json
Date: Fri, 24 Jul 2020 20:06:02 GMT
Server: WSGIServer/0.2 CPython/3.8.0
Vary: Accept
```

Check that in the terminal where you are running `python manage.py runserver` you have got the content of the mail message that would be sent to **joe@bloggs.com**. Copy the confirmation URL and visit it to confirm the comment.

Post a test comment as a signed in user

To post a comment as a logged in user we first have to obtain the csrftoken:

```
$ http localhost:8000/api-auth/login/ --session=session1 -h

HTTP/1.1 200 OK
Cache-Control: max-age=0, no-cache, no-store, must-revalidate, private
Content-Length: 4253
Content-Type: text/html; charset=utf-8
Date: Fri, 24 Jul 2020 21:00:35 GMT
Expires: Fri, 24 Jul 2020 21:00:35 GMT
Server: WSGIServer/0.2 CPython/3.8.0
Server-Timing: SQLPanel_sql_time;dur=0;desc="SQL 0 queries"
Set-Cookie:␣
↪csrftoken=nEJczcG2M3LrcxIKiHbkxDFy2gmplPtn87pAFhp0CQz47TvZ58v8S2eCpWD9Zadm;
↪ expires=Fri, 23 Jul 2021 21:00:35 GMT; Max-Age=31449600; Path=/;␣
↪SameSite=Lax
Vary: Cookie
```

Now we copy the value of csrftoken and attach it to the login HTTP request:

```
$ http -f POST localhost:8000/api-auth/login/ username=admin password=admin \
          X-
↪CSRFToken:nEJczcG2M3LrcxIKiHbkxDFy2gmplPtn87pAFhp0CQz47TvZ58v8S2eCpWD9Zadm␣
↪\
          --session=session1

HTTP/1.1 302 Found
Cache-Control: max-age=0, no-cache, no-store, must-revalidate, private
Content-Length: 0
Content-Type: text/html; charset=utf-8
Date: Fri, 24 Jul 2020 21:06:11 GMT
Expires: Fri, 24 Jul 2020 21:06:11 GMT
Location: /accounts/profile/
Server: WSGIServer/0.2 CPython/3.8.0
Set-Cookie:␣
↪csrftoken=z3FtVTPWudwYrWrqSQLOb2HZ0JNAmoA3P8M4RSDhTtJr7LrSVVAbfDp847Xetuwm;
↪ expires=Fri, 23 Jul 2021 21:06:11 GMT; Max-Age=31449600; Path=/;␣
↪SameSite=Lax
Set-Cookie: sessionid=iyq0q9kqxhjwsgnq95taqbdw2p35v4jb; expires=Fri, 07 Aug␣
↪2020 21:06:11 GMT; HttpOnly; Max-Age=1209600; Path=/; SameSite=Lax
Vary: Cookie
```

Finally we send the comment with the new csrftoken:

```
$ http POST http://localhost:8000/comments/api/comment/ \
            content_type="articles.article" object_pk=1 followup=false \
            reply_to=0 comment="This is the body, the actual comment..." \
            name="Administrator" email="admin@example.com" \
            X-
↪CSRFToken:z3FtVTPWudwYrWrqSQLOb2HZ0JNAmoA3P8M4RSDhTtJr7LrSVVAbfDp847Xetuwm␣
↪\
            --session=session1

HTTP/1.1 201 Created
Allow: POST, OPTIONS
```

(continues on next page)

(continued from previous page)

```
Content-Length: 282
Content-Type: application/json
Date: Fri, 24 Jul 2020 21:06:58 GMT
Server: WSGIServer/0.2 CPython/3.8.0
Vary: Accept, Cookie

{
    "comment": "This is the body, the actual comment...",
    "content_type": "articles.article",
    "email": "admin@example.com",
    "followup": false,
    "honeypot": "",
    "name": "Administrator",
    "object_pk": "1",
    "reply_to": 0,
    "security_hash": "9da968a7ff000f2bd4aa1a669bb70d18934be574",
    "timestamp": "1595624818"
}
```

And the comment must have been posted as the user `admin`.

3.2.2 Retrieve comment list

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

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

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

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

HTTP Methods: GET

HTTP Responses: 200

Serializer: `django_comments_xtd.api.serializers.ReadCommentSerializer`

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

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

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

[
    {
        "allow_reply": true,
        "comment": "Integer erat leo, ...",
        "flags": [
            {
                "flag": "like",
                "id": 1,
                "user": "admin"
            },
```

(continues on next page)

(continued from previous page)

```
            {
                "flag": "like",
                "id": 2,
                "user": "fulanito"
            },
            {
                "flag": "removal",
                "id": 2,
                "user": "fulanito"
            }
        ],
        "id": 10,
        "is_removed": false,
        "level": 0,
        "parent_id": 10,
        "permalink": "/comments/cr/8/4/#c10",
        "submit_date": "May 18, 2017, 9:19 AM",
        "user_avatar": "http://www.gravatar.com/avatar/7dad9576 ...",
        "user_moderator": true,
        "user_name": "Joe Bloggs",
        "user_url": ""
    },
    {
        ...
    }
]
```

3.2.3 Retrieve comments count

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

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

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

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

HTTP Methods: GET

HTTP Responses: 200

Serializer: `django_comments_xtd.api.serializers.ReadCommentSerializer`

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

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

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

{
    "count": 4
}
```

3.2.4 Post like/dislike feedback

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

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

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

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

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

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

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

HTTP/1.0 204 No Content
Allow: POST, OPTIONS
Content-Length: 0
Date: Tue, 23 May 2017 12:26:56 GMT
Server: WSGIServer/0.2 CPython/3.6.0
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN
```

It requires the user to be logged in:

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

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

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

3.2.5 Post removal suggestions

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

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

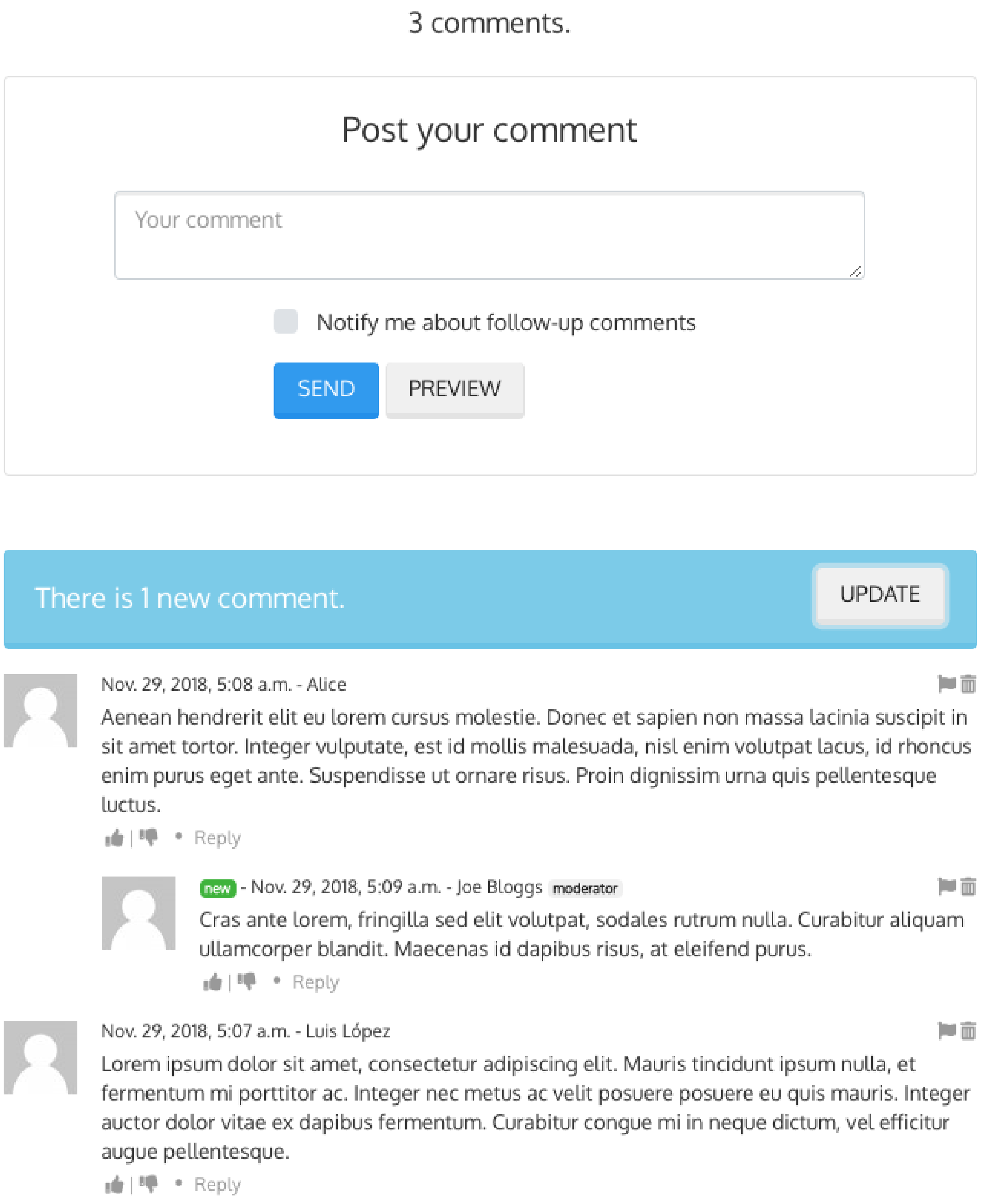
```
$ http POST http://localhost:8000/comments/api/flag/ comment=10 flag="report"

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

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

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

3.3 JavaScript plugin

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

Note: Future v3 of django-comments-xtd will offer a vanilla JavaScript plugin free of frontend choices, to replace the current plugin based on ReactJS, jQuery and Twitter-bootstrap.


3 comments.

Post your comment

Your comment

Notify me about follow-up comments

SEND PREVIEW

There is 1 new comment. UPDATE

Nov. 29, 2018, 5:08 a.m. - Alice

Aenean hendrerit elit eu lorem cursus molestie. Donec et sapien non massa lacinia suscipit in sit amet tortor. Integer vulputate, est id mollis malesuada, nisl enim volutpat lacus, id rhoncus enim purus eget ante. Suspendisse ut ornare risus. Proin dignissim urna quis pellentesque luctus.

| • Reply

new - Nov. 29, 2018, 5:09 a.m. - Joe Bloggs moderator

Cras ante lorem, fringilla sed elit volutpat, sodales rutrum nulla. Curabitur aliquam ullamcorper blandit. Maecenas id dapibus risus, at eleifend purus.

| • Reply

Nov. 29, 2018, 5:07 a.m. - Luis López

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris tincidunt ipsum nulla, et fermentum mi porttitor ac. Integer nec metus ac velit posuere posuere eu quis mauris. Integer auctor dolor vitae ex dapibus fermentum. Curabitur congue mi in neque dictum, vel efficitur augue pellentesque.

| • Reply

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

3.3.1 Frontend opinions

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

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

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

The JavaScript Plugin is based on:

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

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

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

3.3.2 Build process

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

Set up the backend

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

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

Check whether the app passes the battery of tests:

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

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

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

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

Install frontend packages

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

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

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

```
$ webpack --watch
```

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

```
{% block extra-js %}
[...]
<script src="{% static 'django_comments_xtd/js/vendor~plugin-2.7.2.js' %}"></
↪script>
<script src="{% static 'django_comments_xtd/js/plugin-2.7.2.js' %}"></script>
{% endblock extra-js %}
```

3.3.3 Code structure

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

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

django_comments_xtd/static/django_comments_xtd/js
├── src
│   ├── comment.jsx
│   ├── commentbox.jsx
│   ├── commentform.jsx
│   ├── index.js
│   └── lib.js
├── vendor~plugin-2.7.2.js
└── plugin-2.7.2.js

1 directory, 7 files
```

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

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

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

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

To use without the template, you can set up an endpoint to get the props by generating a view action within the *Web API*.

3.3.4 Improvements and contributions

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

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

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

3.4 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

3.4.1 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  %}
```

3.4.2 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 %}
```

3.4.3 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 %}
```

3.4.4 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 %}
```

3.4.5 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 %}
```

3.4.6 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.

You can custom the way of generating the avatar, like this:

> {{ comment.email|xtd_comment_gravatar:'48,mm' }}

3.4.7 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.

3.4.8 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`.

3.4.9 Filter `can_receive_comments_from`

Filter syntax:

```
{{ object|can_receive_comments_from:user }}
```

Returns True depending on the value of the `'who_can_post'` entry in the *COMMENTS_XTD_APP_MODEL_OPTIONS*.

3.5 Migrating to django-comments-xtd

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

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

3.5.1 Preparation

First of all, install django-comments-xtd:

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

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

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

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

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

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

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

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

Now create the tables for django-comments-xtd:

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

3.5.2 Populate comment data

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

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

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

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

3.6 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.

3.6.1 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.

3.6.2 Modifying comments with code

Here's an example of how to access the underlying model storing your comments:

```
from django_comments_xtd.models import XtdComment
from django.contrib.contenttypes.models import ContentType

def unbsubscribe_everyone(model_instance):
    content_type = ContentType.objects.get_for_model(model_instance)

    XtdComment.objects\
        .filter(content_type=content_type, object_pk=model_instance.pk)\
        .update(followup=False)
```

3.7 Internationalization

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

- English, **en** (default language)
- Finnish, **fi**
- French, **fr**
- Norwegian, **no**
- Spanish, **es**

3.7.1 Contributions

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

```
$ virtualenv venv
$ source venv/bin/activate
(venv)$ git clone git://github.com/danirus/django-comments-xtd.git
(venv)$ cd django-comments-xtd/example/comp
(venv)$ pip install django-rosetta django-markdown2
```

(continues on next page)

(continued from previous page)

```
(venv)$ python manage.py migrate
(venv)$ python manage.py loaddata ../fixtures/auth.json
(venv)$ python manage.py loaddata ../fixtures/sites.json
(venv)$ python manage.py loaddata ../fixtures/articles.json
(venv)$ python manage.py loaddata ../fixtures/quotes.json
(venv)$ python manage.py runserver
```

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

```
LANGUAGES = (
    ('nl', 'Dutch'),
    ('en', 'English'),
    ('fi', 'Finnish'),
    ('fr', 'French'),
    ('de', 'German'),
    ('no', 'Norwegian'),
    ('ru', 'Russian'),
    ('es', 'Spanish'),
    ...
)

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

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

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

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

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

```
$ source venv/bin/activate
(venv)$ cd django-comments-xtd/django_comments_xtd
(venv)$ django-admin makemessages -l de
(venv)$ django-admin makemessages -d djangojs -l de
```

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

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

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

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

3.8 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

3.8.1 COMMENTS_XTD_MAX_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_MAX_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.

3.8.2 COMMENTS_XTD_MAX_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 # Omit, 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*.

3.8.3 COMMENTS_XTD_CONFIRM_EMAIL

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`.

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

It defaults to `settings.DEFAULT_FROM_EMAIL`.

3.8.5 COMMENTS_XTD_CONTACT_EMAIL

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

An example:

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

It defaults to `settings.DEFAULT_FROM_EMAIL`.

3.8.6 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"*.

3.8.7 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"*.

3.8.8 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')`

3.8.9 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`.

3.8.10 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 = 'G0h5gt073h6gH4p25GS2g5AQ2Tm256yGt134tMP5TgCX$&HKOYRV'
```

It defaults to an empty string.

3.8.11 `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.

3.8.12 `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`.

3.8.13 `COMMENTS_XTD_APP_MODEL_OPTIONS`

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

- `allow_flagging`: Allow registered users to flag comments as inappropriate.
- `allow_feedback`: Allow registered users to like/dislike comments.
- `show_feedback`: Allow django-comments-xtd to report the list of users who liked/disliked the comment. The representation of each user in the list depends on the next setting :setting::*COMMENTS_XTD_API_USER_REPR*.
- `who_can_post`: Can be either 'all' or 'users'. When it is 'all', all users can post, whether registered users or mere visitors. When it is 'users', only registered users can post. Read the use case *Only signed in users can comment*, for details on how to set it up.

An example use:

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

Defaults to:

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

3.8.14 COMMENTS_XTD_API_USER_REPR

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

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

Defaults to:

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

3.8.15 COMMENTS_XTD_API_GET_USER_AVATAR

Optional. Path to the function used by the web API to retrieve the user's image URL of the user associated with a comment. By default django-comments-xtd tries to retrieve images from Gravatar. If you use the web API (the JavaScript plugin uses it) then you might want to write a function to provide the URL to the user's image from a comment object. You might be interested on the use case *Change user image or avatar*, which cover the topic in depth.

```
COMMENTS_XTD_API_GET_USER_AVATAR = "comp.utils.get_avatar_url"
```

The function used by default, **get_user_avatar** in `django_comments_xtd/utils.py`, tries to fetch every user's image from Gravatar:

```
COMMENTS_XTD_API_GET_USER_AVATAR = "django_comments_xtd.utils.get_user_avatar
↪"
```

3.9 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

- *like.html*
- *liked.html*
- *dislike.html*
- *disliked.html*
- *discarded.html*
- *email_followup_comment*
- *comment.html*
- *posted.html*
- *reply.html*
- *muted.html*
- *only_users_can_post.html*

3.9.1 `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.

3.9.2 `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.

3.9.3 `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.

3.9.4 `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.

3.9.5 `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.

3.9.6 `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.

3.9.7 `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.

3.9.8 `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.

3.9.9 `email_followup_comment`

As `.html` and `.txt`, this template represents the mail message sent to notify that comments have been sent after yours. It's sent to the user who posted the comment in the first place, when another comment arrives in the same thread or in a not nested list of comments. To receive this email the user must tick the box *Notify me 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.

3.9.10 `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.

3.9.11 `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.

3.9.12 `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.

3.9.13 `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.

3.9.14 `only_users_can_post.html`

django-comments-xtd can be customize so that only registered users can post comments. Read the use case *Only registered users can post*, for details. The purpose of this template is to allow customizing the HTML message displayed when a non-registered visitor gets to the comments page. The message is displayed instead of the comment form.

This template expects a context variable `html_id_suffix`.

3.10 Use cases

This page introduces more specific examples of how to use django-comments-xtd.

3.10.1 Change user image or avatar

By default django-comments-xtd uses its own template filters to fetch user profile images from Gravatar and put them in comments. It is a simple solution that makes the app dependant on an external service. But it is a good solution when the only confirmed record we get from the person who posted the comment is an email address.

This page describes how to setup django-comments-xtd so that user images displayed in comments come from other sources. For such purpose, in addition to the default filters, we will use django-avatar. Your Django project can use another application to procure user images, perhaps with a built-in solution or via a social service. Just adapt your case to the methodology exposed here.

Table of Contents

Purpose

This how-to will combine both solutions, template filters provided with django-comments-xtd and template tags provided with django-avatar, to fetch user images for two different type of comments, those posted by registered users and those posted by mere visitors:

- Avatar images in comments posted by **non-registered users** will be fetched from Gravatar via django-comments-xtd filters, as the only record we get from those visitors is their email address.
- On the other hand avatar images in comments posted by **registered users** will be provided by django-avatar templatetags, as django-avatar's templatetags require a user object to fetch the user's image.

Django-avatar will make those images available from different sources depending on how we customize the app. We can add user images directly using Django's admin interface or we can let the app fetch them via providers, from social networks or writing our own specific provider function.

For the purpose of this how-to we will use the *Comp project* introduced in the Demo projects page. Before you continue reading please, visit the samples *Setup* page and get the *Comp project* up and running.

Add django-avatar to the Comp project

Install django-avatar with `pip install django-avatar`, and be sure that the `comp/settings.py` module contains the following entries:

```
MEDIA_ROOT = os.path.join(PROJECT_DIR, "media")
MEDIA_URL = '/media/'
INSTALLED_APPS = [
```

(continues on next page)

(continued from previous page)

```
    ...
    'avatar',
    ...
]
```

Then run the migrate command to create django-avatar tables: `python manage.py migrate`.

Also change the `comp/urls.py` module to serve media files in development and get access to users' images stored with django-avatar:

```
from django.conf import settings
from django.conf.urls.static import static

[...]

# At the bottom of the module.
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

When using HTML templates to render the comments

If your project uses django-comments-xtd HTML templates to render comments, like the Comp project's **quotes app** does, then you only have to adapt your project's HTML templates to use django-avatar.

Let's go then through the following changes:

- Change the `comment_tree.html` template.
- Create the `comments/preview.html` template.
- Create the `django_comments_xtd/comment.html` template (bonus).

Change the `comment_tree.html` template

The *Comp project* uses the **render_xtdcomment_tree** template tag to render the tree of comments in `quotes/quote_detail.html`. **render_xtdcomment_tree** in turn renders the `django_comments_xtd/comment_tree.html` template.

The comp project overrides the `comment_tree.html` template. Let's edit it (in `comp/templates/django_comments_xtd`) to make it start as follows:

```
{% load l10n %}
{% load i18n %}
{% load comments %}
{% load avatar_tags %}
{% load comments_xtd %}

{% for item in comments %}
<div class="media">
  <a name="c{{ item.comment.id }}"></a>
  <img
    {% if item.comment.user and item.comment.user|has_avatar %}
      src="{% avatar_url item.comment.user 48 %}"
    {% else %}
      src="{{ item.comment.user_email|xtd_comment_gravatar_url }}"
```

(continues on next page)

(continued from previous page)

```
    {% endif %}
    class="mr-3" height="48" width="48"
  >
  <div class="media-body">
    [...]
```

Create the `comments/preview.html` template

We also want to apply the same logic to the `comments/preview.html` template. The preview template gets rendered when the user clicks on the preview button in the comment form.

The `preview.html` template is initially served by django-contrib-comments, but it is overriden by a copy provided from django-comments-xtd templates directory.

For our purpose we have to modify that version, let's copy it from django-comments-xtd's templates directory into the comp project templates directory:

```
$ cp django_comments_xtd/templates/comments/preview.html example/comp/templates/
↪comments/
```

And edit the template so that the `<div class="media">` starts like this:

```
{% load avatar_tags %}

[...]

      <div class="media">
        <img
          {% if request.user|has_avatar %}
            src="{% avatar_url request.user 48 %}"
          {% else %}
            src="{{ form.cleaned_data.user_email|xtd_comment_gravatar_url }}"
          {% endif %}
          class="mr-3" width="48" height="48"
        >
        <div class="media-body">

[...]
```

Create the `django_comments_xtd/comment.html` template

Additionally to the templates used by the **quotes app**, the *Comp project* displays a list with the last 5 comments posted to the site, regardless of whether they were sent to the quotes app or the articles app.

In order to get the appropriate avatar images in such a list we need to override the `django_comments_xtd/comment.html` template:

```
$ cp django_comments_xtd/templates/django_comments_xtd/comment.html example/comp/
↪templates/django_comments_xtd/
```

Now edit the template and make the following changes:

```
{% load avatar_tags %}

[...]

<div id="c{{ comment.id }}" class="media"><a name="c{{ comment.id }}"></a>
  <img
    {% if comment.user|has_avatar %}
      src="{% avatar_url comment.user 48 %}"
    {% else %}
      src="{{ comment.user_email|xtd_comment_gravatar_url }}"
    {% endif %}
    height="48" width="48" class="mr-3"
  >
  <div class="media-body">

[...]
```

Test the changes

These changes are enough when your project uses only Django templates to render comments.

Before we can test the solution, let's add an image for the admin user. Do login in the admin UI with user/password `admin/admin` and click on the avatar application. Add a squared dimensioned image to the admin user.

Now the project is ready to test the two types of comments, a comment sent as a logged-in user and another one sent as a mere visitor:

1. While you are still logged in in the admin interface, visit the quotes page, click on any of the links and send a comment as the admin user. Sending a comment as a logged in user does not require comment confirmation by email. Therefore you must see already the comment posted in the page and displaying the image you have added to the avatar model using the admin interface. Let's now send a comment as a mere visitor.
2. Logout from the admin interface and send another comment as a mere visitor. If you have an account in Gravatar, use an email address of that account for the comment. This way, when you post the comment, you already know what's the image that is going to be displayed from Gravatar. Then send the comment. The email message to confirm the comment is displayed in the console. Scroll up in the console to see the plain-text part of the message and copy the confirmation URL. Then paste it in the browser's location bar to confirm the comment. Once the message is confirmed the comment appears in the quotes page. It should show the image from your Gravatar account.

The message posted as the admin user gets the avatar image from the project's storage using django-avatar's template tag. On the other hand, the image sent as a mere visitor, comes directly from Gravatar using django-comments-xtd's template filter.

When using the web API

If your project uses the web API you have to customize *COMMENTS_XTD_API_GET_USER_AVATAR* to point to the function that will retrieve the avatar image when the REST API requires it.

The **articles app** of the *Comp project* uses the web API (actually, the JavaScript plugin does). We have to customize it so that avatar images for registered users are fetched using django-avatar, while avatar images for mere visitors are fetched using the standard Gravatar approach.

The default value of *COMMENTS_XTD_API_GET_USER_AVATAR* points to the function **get_user_avatar** in `django_comments_xtd/utils.py`. That function only uses Gravatar to fetch user images.

To acomplish it we only need to do the following:

- Implement the function that fetches images' URLs.
- Override COMMENTS_XTD_API_GET_USER_AVATAR.
- Test the changes.

Implement the function that fetches images' URLs

We want to apply the following logic when fetching images' URLs:

- When a registered user sends a comment, the comment.user object points to an instance of that user. There we will use **django-avatar** to fetch that uses's image URL.
- When a mere visitor sends a comment, the comment.user object is None. But we still have the comment.user_email which contains the email address of the visitor. Here we will use django-comments-xtd (which in turn defaults to Gravatar).

Create the module comp/utils.py with the following content:

```
from avatar.templatetags.avatar_tags import avatar_url
from django_comments_xtd.utils import get_user_avatar


def get_avatar_url(comment):
    ret = None
    if comment.user is not None:
        try:
            return avatar_url(comment.user)
        except Exception as exc:
            pass
    return get_user_avatar(comment)
```

If the comment has a user, we return the result of the avatar_url function of django-avatar. This function goes through each of the django-avatar providers setup with AVATAR_PROVIDERS and returns the appropriate image's URL.

If on the hand the comment does not have a user, we return what Gravatar has on the comment.user_email. If that email address is not registered in Gravatar, it returns the default image (which you can customize too, read in that page from *Default Image* on).

Override COMMENTS_XTD_API_GET_USER_AVATAR

We have to add a reference to our new function in the settings, to override the content of *COMMENTS_XTD_API_GET_USER_AVATAR*. Append the following to the ‘‘comp/settings.py‘ module:

```
COMMENTS_XTD_API_GET_USER_AVATAR = "comp.utils.get_avatar_url"
```

Now the web API will use that function instead of the default one.

Test the changes

Now the **articles app** is ready. If you already added an avatar image for the admin user, as we did in the previous **Test the changes** section, then send two comments to any of the articles:

1. Login in as admin/admin in the admin UI, then visit any of the articles page and send a comment as the admin user. See also that the image displayed in the preview corresponds to the image added to the admin user.

2. Logout from the admin interface and send another comment as a mere visitor. If you have a Gravatar account, use the same email address when posting the comment. The Gravatar image associated should be displayed in the comment.

Conclusion

Images displayed in association with comments can come from customized sources. Adapting your project to use your own sources is a matter of adjusting the templates or writing a new function to feed web API calls.

3.10.2 Only signed in users can comment

This page describes how to setup django-comments-xtd so that only registered users can write comments or flag them. That means mere visitors will be able to see the comments but won't be able to send them. In order to do so a visitor must login first. The following instructions use the Django admin interface to login and logout users.

Table of Contents

Simple example using only the backend

A simple site using django-comments-xtd can be represented by the *Simple project*.

Customize the simple project

The Simple project is a basic example site that allows both, visitors and registered users, post comments to articles. It has been introduced in the Demo projects page: *Simple project*. The example loads a couple of articles to illustrate the functionality.

If you have already setup the *Simple project*, and have sent a few testing comments to see that visitors and registered users can comment, add the *COMMENTS_XTD_APP_MODEL_OPTIONS* entry at the bottom of the `settings.py` module to allow only registered users to post comments:

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

Once the change is saved and Django has reloaded check that, as before, registered users can comment without issues, however visitors get the HTTP-400 page (Bad Request).

As a final step to customize the simple example site either edit `templates/comments/form.html` or `templates/articles/article_detail.html` to display a message inviting visitors to login or register instead of showing the post comment form.

As an example, here is a modified version of the `article_detail.html` template of the Simple project that displays a message with a link to the login page when the user is not authenticated:

```
[...]

  {% if object.allow_comments %}
    {% if user.is_authenticated %}
      <div class="comment">
        <h5 class="text-center">Post your comment</h5>
        <div class="well my-4">
          {% render_comment_form for object %}
        </div>
      </div>
    {% else %}
      <p class="text-center">
        Only registered users can post comments. Please,
        <a href="{% url 'admin:login' %}?next={{ object.get_absolute_url }}">
↪login</a>.
      </p>
    {% endif %}
  {% else %}
    <h5 class="text-center">comments are disabled for this article</h5>
  {% endif %}

[...]
```

Full featured example using backend and frontend code

This section goes through the steps to customize a project that uses both, the backend and the frontend side of django-comments-xtd, to prevent that unregistered users can post comments.

We will use the *Comp project*.

The *Comp project* contains two very similar apps: articles and quotes. Both apps allow visitors and registered users to post nested comments. The main difference between articles and quotes in the Comp project is that the articles app uses the JavaScript plugin, while the quotes app uses merely the backend.

Customize the quotes app

If you have already setup the *Comp project*, and have sent a few testing comments to see that visitors and registered users can comment, edit the *COMMENTS_XTD_APP_MODEL_OPTIONS* at the bottom of the `settings.py` and append the pair `'who_can_post':  'users'` to the quotes app dictionary:

```
COMMENTS_XTD_APP_MODEL_OPTIONS = {
    'quotes.quote': {
        'allow_flagging': True,
        'allow_feedback': True,
        'show_feedback': True,
        'who_can_post': 'users'
    }
}
```

Once changes are saved and Django has restarted see that registered users can comment without issues. However visitors get the HTTP-400 page (Bad Request).

One last customization is required to prevent the HTTP-400 Bad Request. We have to edit the `templates/quotes/quote_detail.html` file and be sure that the block that renders the comment form is not displayed when the user browsing the site is a mere visitor. The following changes will make it:

```
[...] around line 41...

    {% if object.allow_comments %}
      {% if object|can_receive_comments_from:user %}
        <div class="card card-block mt-4 mb-5">
          <div class="card-body">
            <h4 class="card-title text-center pb-3">Post your comment</h4>
            {% render_comment_form for object %}
          </div>
        </div>
      {% else %}
        <p class="mt-4 mb-5 text-center">
          Only registered users can post comments. Please,
          <a href="{% url 'admin:login' %}?next={{ object.get_absolute_url }}
↪">login</a>.
        </p>
      {% endif %}
    {% else %}
      <h4 class="mt-4 mb-5 text-center text-secondary">
        Comments are disabled for this quote.
      </h4>
    {% endif %}

[...]
```

Note: See that in the previous snippet we use the template filter *`can_receive_comments_from`*. Using this filter you could change the setting `'who_can_post'` to `'all'` in your *`COMMENTS_XTD_APP_MODEL_OPTIONS`* to allow mere visitors to post comments, and your template would do as expected without further changes.

If we rather had used `{% if user.is_authenticated %}` the template would have still to be changed to display the comment form to all, visitors and registered users.

After the template changes are saved, mere users will see a message inviting them to login. Also, the **Reply** link to send nested comments is already aware of the value of the `'who_can_post'` setting and will redirect you to login if you have not done so yet.

Customize the articles app of the comp project

The articles app uses the JavaScript plugin. The only change required consist of adding the pair `'who_can_post': 'users'` to the `'articles.article'` dictionary entry of the *`COMMENTS_XTD_APP_MODEL_OPTIONS`*, as we did with the quotes app. That will make it work.

Run the site and check that as a mere visitor (logout first) you can not send comments to articles. Instead of the comment form there must be a message in blue saying that **Only registered users can post comments.** If you login and visit an article's page the comment form will be visible again.

The message in blue is the default response hardcoded in the `commentbox.jsx` module of the JavaScript plugin. The commentbox module controls whether the user in the session can post comments or not. If the user can not post comments it defaults to display that message in blue.

Most of the times we will want to customize that message. We will achieve it by modifying both, the `base.html` and the `articles/article_detail.html`, and by creating a new template in the `comp/templates/`

`django_comments_xtd` directory called `only_users_can_post.html`.

The changes in `templates/base.html` consist of adding a hidden block. We will put content in this hidden block in the `articles_detail.html`. Add the following HTML code before the script tags in the `base.html` in the `example/comp/templates` directory:

```
[...] around line 67, right before the first <script> tag...

    <div style="display:none">
      {% block hidden %}
      {% endblock %}
    </div>

[...]
```

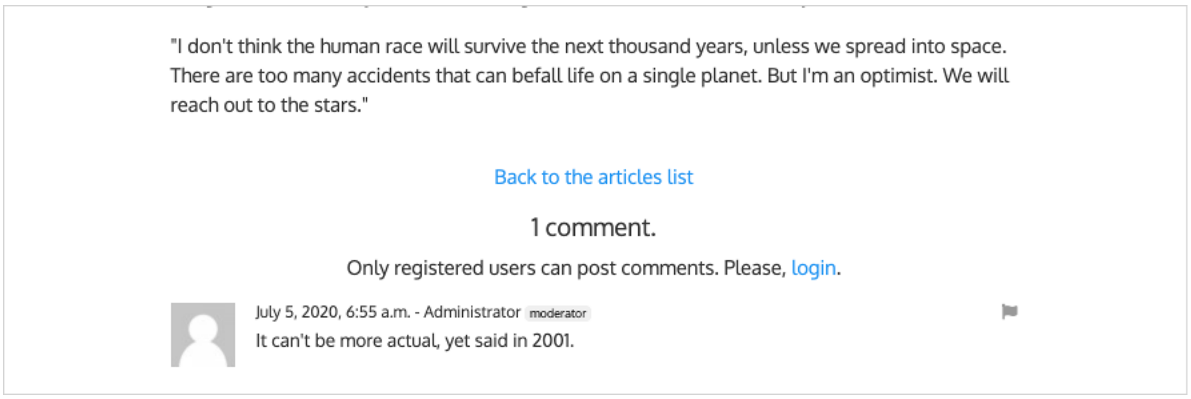
Add the following code to `templates/articles/article_detail.html`:

```
[...] around line 46, right before the {% block extra_js %}...

{% block hidden %}
  {% render_only_users_can_post_template object %}
{% endblock %}
```

And finally create the file `only_users_can_post.html` within the `comp/templates/django_comments_xtd` directory, with the following content in it:

```
<div id="only-users-can-post-{{ html_id_suffix }}">
  <p class="text-center">Only registered users can post comments. Please,
    <a href="{% url 'admin:login' %}?next={{ object.get_absolute_url }}">
↪login</a>.
  </p>
</div>
```

With all the changes already done, logout of the comp site and reload the article's page. You should see the message with the login link.


"I don't think the human race will survive the next thousand years, unless we spread into space. There are too many accidents that can befall life on a single planet. But I'm an optimist. We will reach out to the stars."

Back to the articles list

1 comment.

Only registered users can post comments. Please, login.

July 5, 2020, 6:55 a.m. - Administrator moderator

It can't be more actual, yet said in 2001.

Python Module Index

Index