# Django Comments Documentation

*Release 1.8*

**Django Software Foundation and contributors**

**Jan 07, 2021**

# Contents

Django used to include a comments framework; since Django 1.6 it's been separated to a separate project. This is that project.

This framework can be used to attach comments to any model, so you can use it for comments on blog entries, photos, book chapters, or anything else.

CHAPTER 1

# Contents

## 1.1 Quick start guide

To get started using the `comments` app, follow these steps:

1. Install the comments app by running `pip install django-contrib-comments`.
2. Enable the "sites" framework by adding `'django.contrib.sites'` to `INSTALLED_APPS` and defining `SITE_ID`.
3. Install the comments framework by adding `'django_comments'` to `INSTALLED_APPS`.
4. Run `manage.py migrate` so that Django will create the comment tables.
5. Add the comment app's URLs to your project's `urls.py`:

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

6. Use the *comment template tags* below to embed comments in your templates.

You might also want to examine *the available settings*.

To take full advantage of the moderation system, you may want to add some comment-enabling fields to specified models. See *Comment moderation* for details.

### 1.1.1 Comment template tags

You'll primarily interact with the comment system through a series of template tags that let you embed comments and generate forms for your users to post them.

Like all custom template tag libraries, you'll need to load the custom tags before you can use them:

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

Once loaded you can use the template tags below.

### Specifying which object comments are attached to

Django's comments are all "attached" to some parent object. This can be any instance of a Django model. Each of the tags below gives you a couple of different ways you can specify which object to attach to:

1. Refer to the object directly – the more common method. Most of the time, you'll have some object in the template's context you want to attach the comment to; you can simply use that object.

   For example, in a blog entry page that has a variable named `entry`, you could use the following to load the number of comments:

   ```
   {% get_comment_count for entry as comment_count %}.
   ```

2. Refer to the object by content-type and object id. You'd use this method if you, for some reason, don't actually have direct access to the object.

   Following the above example, if you knew the object ID was `14` but didn't have access to the actual object, you could do something like:

   ```
   {% get_comment_count for blog.entry 14 as comment_count %}
   ```

   In the above, `blog.entry` is the app label and (lower-cased) model name of the model class.

### Displaying comments

To display a list of comments, you can use the template tags *render_comment_list* or *get_comment_list*.

### Quickly rendering a comment list

The easiest way to display a list of comments for some object is by using *render_comment_list*:

```
{% render_comment_list for [object] %}
```

For example:

```
{% render_comment_list for event %}
```

This will render comments using a template named `comments/list.html`, a default version of which is included with Django.

### Rendering a custom comment list

To get the list of comments for some object, use *get_comment_list*:

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

For example:

```
{% get_comment_list for event as comment_list %}
{% for comment in comment_list %}
    ...
{% endfor %}
```

This returns a list of *Comment* objects; see *the comment model documentation* for details.

### Linking to comments

To provide a permalink to a specific comment, use *get_comment_permalink*:

```
{% get_comment_permalink comment_obj [format_string] %}
```

By default, the named anchor that will be appended to the URL will be the letter 'c' followed by the comment id, for example 'c82'. You may specify a custom format string if you wish to override this behavior:

```
{% get_comment_permalink comment "#c%(id)s-by-%(user_name)s"%}
```

The format string is a standard python format string. Valid mapping keys include any attributes of the comment object.

Regardless of whether you specify a custom anchor pattern, you must supply a matching named anchor at a suitable place in your template.

For example:

```
{% for comment in comment_list %}
    <a name="c{{ comment.id }}"></a>
    <a href="{% get_comment_permalink comment %}">
        permalink for comment #{{ forloop.counter }}
    </a>
    ...
{% endfor %}
```

**Warning:** There's a known bug in Safari/Webkit which causes the named anchor to be forgotten following a redirect. The practical impact for comments is that the Safari/webkit browsers will arrive at the correct page but will not scroll to the named anchor.

### Counting comments

To count comments attached to an object, use *get_comment_count*:

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

For example:

```
{% get_comment_count for event as comment_count %}

<p>This event has {{ comment_count }} comments.</p>
```

### Displaying the comment post form

To show the form that users will use to post a comment, you can use *render_comment_form* or *get_comment_form*

### Quickly rendering the comment form

The easiest way to display a comment form is by using *render_comment_form*:

```
{% render_comment_form for [object] %}
```

For example:

```
{% render_comment_form for event %}
```

This will render comments using a template named `comments/form.html`, a default version of which is included with Django.

### Rendering a custom comment form

If you want more control over the look and feel of the comment form, you may use *get_comment_form* to get a `form object` that you can use in the template:

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

A complete form might look like:

```
{% get_comment_form for event as form %}
<table>
  <form action="{% comment_form_target %}" method="post">
    {% csrf_token %}
    {{ form }}
    <tr>
      <td colspan="2">
        <input type="submit" name="submit" value="Post">
        <input type="submit" name="preview" value="Preview">
      </td>
    </tr>
  </form>
</table>
```

Be sure to read the *notes on the comment form*, below, for some special considerations you'll need to make if you're using this approach.

### Getting the comment form target

You may have noticed that the above example uses another template tag – *comment_form_target* – to actually get the `action` attribute of the form. This will always return the correct URL that comments should be posted to; you'll always want to use it like above:

```
<form action="{% comment_form_target %}" method="post">
```

### Redirecting after the comment post

To specify the URL you want to redirect to after the comment has been posted, you can include a hidden form input called `next` in your comment form. For example:

```
<input type="hidden" name="next" value="{% url 'my_comment_was_posted' %}" />
```

### Providing a comment form for authenticated users

If a user is already authenticated, it makes little sense to display the name, email, and URL fields, since these can already be retrieved from their login data and profile. In addition, some sites will only accept comments from authenticated users.

To provide a comment form for authenticated users, you can manually provide the additional fields expected by the Django comments framework. For example, assuming comments are attached to the model "object":

```
{% if user.is_authenticated %}
    {% get_comment_form for object as form %}
    <form action="{% comment_form_target %}" method="POST">
    {% csrf_token %}
    {{ form.comment }}
    {{ form.honeypot }}
    {{ form.content_type }}
    {{ form.object_pk }}
    {{ form.timestamp }}
    {{ form.security_hash }}
    <input type="hidden" name="next" value="{% url 'object_detail_view' object.id %}"␣
↪/>
    <input type="submit" value="Add comment" id="id_submit" />
    </form>
{% else %}
    <p>Please <a href="{% url 'auth_login' %}">log in</a> to leave a comment.</p>
{% endif %}
```

The honeypot, content_type, object_pk, timestamp, and security_hash fields are fields that would have been created automatically if you had simply used `{{ form }}` in your template, and are referred to in *Notes on the comment form* below.

Note that we do not need to specify the user to be associated with comments submitted by authenticated users. This is possible because the *Built-in Comment Models* that come with Django associate comments with authenticated users by default.

In this example, the honeypot field will still be visible to the user; you'll need to hide that field in your CSS:

```
#id_honeypot {
    display: none;
}
```

If you want to accept either anonymous or authenticated comments, replace the contents of the "else" clause above with a standard comment form and the right thing will happen whether a user is logged in or not.

### Notes on the comment form

The form used by the comment system has a few important anti-spam attributes you should know about:

- It contains a number of hidden fields that contain timestamps, information about the object the comment should be attached to, and a "security hash" used to validate this information. If someone tampers with this data – something comment spammers will try – the comment submission will fail.

  If you're rendering a custom comment form, you'll need to make sure to pass these values through unchanged.

- The timestamp is used to ensure that "reply attacks" can't continue very long. Users who wait too long between requesting the form and posting a comment will have their submissions refused.

- The comment form includes a "honeypot" field. It's a trap: if any data is entered in that field, the comment will be considered spam (spammers often automatically fill in all fields in an attempt to make valid submissions).

  The default form hides this field with a piece of CSS and further labels it with a warning field; if you use the comment form with a custom template you should be sure to do the same.

The comments app also depends on the more general Cross Site Request Forgery protection that comes with Django. As described in the documentation, it is best to use `CsrfViewMiddleware`. However, if you are not using that, you will need to use the `csrf_protect` decorator on any views that include the comment form, in order for those views to be able to output the CSRF token and cookie.

## 1.2 The comment models

**class** `django_comments.models.`**Comment**

The comment model. Has the following fields:

**content_object**
A `GenericForeignKey` attribute pointing to the object the comment is attached to. You can use this to get at the related object (i.e. `my_comment.content_object`).

Since this field is a `GenericForeignKey`, it's actually syntactic sugar on top of two underlying attributes, described below.

**content_type**
A `ForeignKey` to `ContentType`; this is the type of the object the comment is attached to.

**object_pk**
A `TextField` containing the primary key of the object the comment is attached to.

**site**
A `ForeignKey` to the `Site` on which the comment was posted.

**user**
A `ForeignKey` to the `User` who posted the comment. May be blank if the comment was posted by an unauthenticated user.

**user_name**
The name of the user who posted the comment.

**user_email**
The email of the user who posted the comment.

**user_url**
The URL entered by the person who posted the comment.

**comment**
The actual content of the comment itself.

**submit_date**
The date the comment was submitted.

**ip_address**
The IP address of the user posting the comment.

**is_public**
`False` if the comment is in moderation (see *Generic comment moderation*); If `True`, the comment will be displayed on the site.

**is_removed**
`True` if the comment was removed. Used to keep track of removed comments instead of just deleting them.

## 1.3 Signals sent by the comments app

The comment app sends a series of signals to allow for comment moderation and similar activities. See the introduction to signals for information about how to register for and receive these signals.

### 1.3.1 comment_will_be_posted

`django_comments.signals.`**`comment_will_be_posted`**

Sent just before a comment will be saved, after it's been sanity checked and submitted. This can be used to modify the comment (in place) with posting details or other such actions.

If any receiver returns `False` the comment will be discarded and a 400 response will be returned.

This signal is sent at more or less the same time (just before, actually) as the `Comment` object's `pre_save` signal.

Arguments sent with this signal:

**sender** The comment model.

**comment** The comment instance about to be posted. Note that it won't have been saved into the database yet, so it won't have a primary key, and any relations might not work correctly yet.

**request** The `HttpRequest` that posted the comment.

### 1.3.2 comment_was_posted

`django_comments.signals.`**`comment_was_posted`**

Sent just after the comment is saved.

Arguments sent with this signal:

**sender** The comment model.

**comment** The comment instance that was posted. Note that it will have already been saved, so if you modify it you'll need to call `save()` again.

**request** The `HttpRequest` that posted the comment.

### 1.3.3 comment_was_flagged

`django_comments.signals.`**`comment_was_flagged`**

Sent after a comment was "flagged" in some way. Check the flag to see if this was a user requesting removal of a comment, a moderator approving/removing a comment, or some other custom user flag.

Arguments sent with this signal:

**sender** The comment model.

**comment** The comment instance that was posted. Note that it will have already been saved, so if you modify it you'll need to call `save()` again.

**flag** The `django_comments.models.CommentFlag` that's been attached to the comment.

**created** `True` if this is a new flag; `False` if it's a duplicate flag.

**request** The `HttpRequest` that posted the comment.

## 1.4 Customizing the comments framework

If the comment framework doesn't quite fit your needs, you can extend the comment app's behavior to add custom data and logic. The comments framework lets you extend the in comment model, the comment form, and the various comment views.

The *COMMENTS_APP* setting is where this customization begins. Set *COMMENTS_APP* to the name of the app you'd like to use to provide custom behavior. You'll use the same syntax as you'd use for `INSTALLED_APPS`, and the app given must also be in the `INSTALLED_APPS` list.

For example, if you wanted to use an app named `my_comment_app`, your settings file would contain:

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

COMMENTS_APP = 'my_comment_app'
```

The app named in *COMMENTS_APP* provides its custom behavior by defining some module-level functions in the app's `__init__.py`. The *complete list of these functions* can be found below, but first let's look at a quick example.

### 1.4.1 An example custom comments app

One of the most common types of customization is modifying the set of fields provided on the comment model. For example, some sites that allow comments want the commentator to provide a title for their comment; the comment model has no field for that title.

To make this kind of customization, we'll need to do three things:

1. Create a custom comment `Model` that adds on the "title" field.
2. Create a custom comment `Form` that also adds this "title" field.
3. Inform Django of these objects by defining a few functions in a custom *COMMENTS_APP*.

So, carrying on the example above, we're dealing with a typical app structure in the `my_comment_app` directory:

```
my_comment_app/
    __init__.py
    models.py
    forms.py
```

In the `models.py` we'll define a `CommentWithTitle` model:

```
from django.db import models
from django_comments.abstracts import CommentAbstractModel

class CommentWithTitle(CommentAbstractModel):
    title = models.CharField(max_length=300)
```

Most custom comment models will subclass the `CommentAbstractModel` model. However, if you want to substantially remove or change the fields available in the `CommentAbstractModel` model, but don't want to rewrite the templates, you could try subclassing from `BaseCommentAbstractModel`.

Next, we'll define a custom comment form in `forms.py`. This is a little more tricky: we have to both create a form and override `CommentForm.get_comment_create_data()` to return deal with our custom title field:

```
from django import forms
from django_comments.forms import CommentForm
from my_comment_app.models import CommentWithTitle

class CommentFormWithTitle(CommentForm):
    title = forms.CharField(max_length=300)

    def get_comment_create_data(self, **kwargs):
        # Use the data of the superclass, and add in the title field
        data = super().get_comment_create_data(**kwargs)
        data['title'] = self.cleaned_data['title']
        return data
```

Django provides a couple of "helper" classes to make writing certain types of custom comment forms easier; see *`django_comments.forms`* for more.

Finally, we'll define a couple of methods in `my_comment_app/__init__.py` to point Django at these classes we've created:

```
def get_model():
    from my_comment_app.models import CommentWithTitle
    return CommentWithTitle

def get_form():
    from my_comment_app.forms import CommentFormWithTitle
    return CommentFormWithTitle
```

The class imports have to be inside functions, as recent Django versions do not allow importing models in the application root `__init__.py` file.

**Warning:** Be careful not to create cyclic imports in your custom comments app. If you feel your comment configuration isn't being used as defined – for example, if your comment moderation policy isn't being applied – you may have a cyclic import problem.

If you are having unexplained problems with comments behavior, check if your custom comments application imports (even indirectly) any module that itself imports Django's comments module.

The above process should take care of most common situations. For more advanced usage, there are additional methods you can define. Those are explained in the next section.

### 1.4.2 Custom comment app API

The *django_comments* app defines the following methods; any custom comment app must define at least one of them. All are optional, however.

`django_comments.`**`get_model`**`()`
Return the `Model` class to use for comments. This model should inherit from `django_comments.abstracts.BaseCommentAbstractModel`, which defines necessary core fields.

The default implementation returns *django_comments.models.Comment*.

`django_comments.`**`get_form`**`()`
Return the `Form` class you want to use for creating, validating, and saving your comment model. Your custom comment form should accept an additional first argument, `target_object`, which is the object the comment will be attached to.

The default implementation returns *django_comments.forms.CommentForm*.

---

**Note:** The default comment form also includes a number of unobtrusive spam-prevention features (see *Notes on the comment form*). If replacing it with your own form, you may want to look at the source code for the default form and consider incorporating similar features.

---

`django_comments.`**`get_form_target`**`()`
Return the URL for POSTing comments. This will be the `<form action>` attribute when rendering your comment form.

The default implementation returns a reverse-resolved URL pointing to the `post_comment()` view.

---

**Note:** If you provide a custom comment model and/or form, but you want to use the default `post_comment()` view, you will need to be aware that it requires the model and form to have certain additional attributes and methods: see the `django_comments.views.post_comment()` view for details.

---

`django_comments.`**`get_flag_url`**`()`
Return the URL for the "flag this comment" view.

The default implementation returns a reverse-resolved URL pointing to the `django_comments.views.moderation.flag()` view.

`django_comments.`**`get_delete_url`**`()`
Return the URL for the "delete this comment" view.

The default implementation returns a reverse-resolved URL pointing to the `django_comments.views.moderation.delete()` view.

`django_comments.`**`get_approve_url`**`()`
Return the URL for the "approve this comment from moderation" view.

The default implementation returns a reverse-resolved URL pointing to the `django_comments.views.moderation.approve()` view.

## 1.5 Comment form classes

The `django_comments.forms` module contains a handful of forms you'll use when writing custom views dealing with comments, or when writing *custom comment apps*.

**class** django_comments.forms.**CommentForm**
The main comment form representing the standard way of handling submitted comments. This is the class used by all the views *django_comments* to handle submitted comments.

If you want to build custom views that are similar to django_comment's built-in comment handling views, you'll probably want to use this form.

### 1.5.1 Abstract comment forms for custom comment apps

If you're building a *custom comment app*, you might want to replace *some* of the form logic but still rely on parts of the existing form.

*CommentForm* is actually composed of a couple of abstract base class forms that you can subclass to reuse pieces of the form handling logic:

**class** django_comments.forms.**CommentSecurityForm**
Handles the anti-spoofing protection aspects of the comment form handling.

This class contains the content_type and object_pk fields pointing to the object the comment is attached to, along with a timestamp and a security_hash of all the form data. Together, the timestamp and the security hash ensure that spammers can't "replay" form submissions and flood you with comments.

**class** django_comments.forms.**CommentDetailsForm**
Handles the details of the comment itself.

This class contains the name, email, url, and the comment field itself, along with the associated validation logic.

## 1.6 Generic comment moderation

Django's bundled comments application is extremely useful on its own, but the amount of comment spam circulating on the Web today essentially makes it necessary to have some sort of automatic moderation system in place for any application which makes use of comments. To make this easier to handle in a consistent fashion, django_comments.moderation provides a generic, extensible comment-moderation system which can be applied to any model or set of models which want to make use of Django's comment system.

### 1.6.1 Overview

The entire system is contained within django_comments.moderation, and uses a two-step process to enable moderation for any given model:

1. A subclass of *CommentModerator* is defined which specifies the moderation options the model wants to enable.
2. The model is registered with the moderation system, passing in the model class and the class which specifies its moderation options.

A simple example is the best illustration of this. Suppose we have the following model, which would represent entries in a Weblog:

```
from django.db import models

class Entry(models.Model):
    title = models.CharField(maxlength=250)
    body = models.TextField()
```

(continues on next page)

(continued from previous page)

```
    pub_date = models.DateField()
    enable_comments = models.BooleanField()
```

Now, suppose that we want the following steps to be applied whenever a new comment is posted on an `Entry`:

1. If the `Entry`'s `enable_comments` field is `False`, the comment will simply be disallowed (i.e., immediately deleted).
2. If the `enable_comments` field is `True`, the comment will be allowed to save.
3. Once the comment is saved, an email should be sent to site staff notifying them of the new comment.

Accomplishing this is fairly straightforward and requires very little code:

```
from django_comments.moderation import CommentModerator, moderator

class EntryModerator(CommentModerator):
    email_notification = True
    enable_field = 'enable_comments'

moderator.register(Entry, EntryModerator)
```

The *CommentModerator* class pre-defines a number of useful moderation options which subclasses can enable or disable as desired, and `moderator` knows how to work with them to determine whether to allow a comment, whether to moderate a comment which will be allowed to post, and whether to email notifications of new comments.

### Moderation options

**class** django_comments.moderation.**CommentModerator**

Most common comment-moderation needs can be handled by subclassing *CommentModerator* and changing the values of pre-defined attributes; the full range of options is as follows.

**auto_close_field**

If this is set to the name of a `DateField` or `DateTimeField` on the model for which comments are being moderated, new comments for objects of that model will be disallowed (immediately deleted) when a certain number of days have passed after the date specified in that field. Must be used in conjunction with *close_after*, which specifies the number of days past which comments should be disallowed. Default value is `None`.

**auto_moderate_field**

Like *auto_close_field*, but instead of outright deleting new comments when the requisite number of days have elapsed, it will simply set the `is_public` field of new comments to `False` before saving them. Must be used in conjunction with *moderate_after*, which specifies the number of days past which comments should be moderated. Default value is `None`.

**close_after**

If *auto_close_field* is used, this must specify the number of days past the value of the field specified by *auto_close_field* after which new comments for an object should be disallowed. Allowed values are `None`, 0 (which disallows comments immediately), or any positive integer. Default value is `None`.

**email_notification**

If `True`, any new comment on an object of this model which survives moderation (i.e., is not deleted) will generate an email to site staff. Default value is `False`.

**enable_field**

If this is set to the name of a `BooleanField` on the model for which comments are being moderated,

new comments on objects of that model will be disallowed (immediately deleted) whenever the value of that field is `False` on the object the comment would be attached to. Default value is `None`.

**moderate_after**
If *auto_moderate_field* is used, this must specify the number of days past the value of the field specified by *auto_moderate_field* after which new comments for an object should be marked non-public. Allowed values are `None`, 0 (which moderates comments immediately), or any positive integer. Default value is `None`.

Simply subclassing *CommentModerator* and changing the values of these options will automatically enable the various moderation methods for any models registered using the subclass.

### Adding custom moderation methods

For situations where the options listed above are not sufficient, subclasses of *CommentModerator* can also override the methods which actually perform the moderation, and apply any logic they desire. *CommentModerator* defines three methods which determine how moderation will take place; each method will be called by the moderation system and passed two arguments: `comment`, which is the new comment being posted, `content_object`, which is the object the comment will be attached to, and `request`, which is the `HttpRequest` in which the comment is being submitted:

`CommentModerator.`**`allow`**(*comment*, *content_object*, *request*)
Should return `True` if the comment should be allowed to post on the content object, and `False` otherwise (in which case the comment will be immediately deleted).

`CommentModerator.`**`email`**(*comment*, *content_object*, *request*)
If email notification of the new comment should be sent to site staff or moderators, this method is responsible for sending the email.

`CommentModerator.`**`moderate`**(*comment*, *content_object*, *request*)
Should return `True` if the comment should be moderated (in which case its `is_public` field will be set to `False` before saving), and `False` otherwise (in which case the `is_public` field will not be changed).

### Registering models for moderation

The moderation system, represented by `django_comments.moderation.moderator` is an instance of the class *Moderator*, which allows registration and "unregistration" of models via two methods:

`moderator.`**`register`**(*model_or_iterable*, *moderation_class*)
Takes two arguments: the first should be either a model class or list of model classes, and the second should be a subclass of `CommentModerator`, and register the model or models to be moderated using the options defined in the `CommentModerator` subclass. If any of the models are already registered for moderation, the exception `AlreadyModerated` will be raised.

`moderator.`**`unregister`**(*model_or_iterable*)
Takes one argument: a model class or list of model classes, and removes the model or models from the set of models which are being moderated. If any of the models are not currently being moderated, the exception `NotModerated` will be raised.

### Customizing the moderation system

Most use cases will work easily with simple subclassing of *CommentModerator* and registration with the provided *Moderator* instance, but customization of global moderation behavior can be achieved by subclassing *Moderator* and instead registering models with an instance of the subclass.

**class** django_comments.moderation.**Moderator**

In addition to the *moderator.register()* and *moderator.unregister()* methods detailed above, the following methods on *Moderator* can be overridden to achieve customized behavior:

**connect**()

Determines how moderation is set up globally. The base implementation in *Moderator* does this by attaching listeners to the *comment_will_be_posted* and *comment_was_posted* signals from the comment models.

**pre_save_moderation**(*sender*, *comment*, *request*, ***kwargs*)

In the base implementation, applies all pre-save moderation steps (such as determining whether the comment needs to be deleted, or whether it needs to be marked as non-public or generate an email).

**post_save_moderation**(*sender*, *comment*, *request*, ***kwargs*)

In the base implementation, applies all post-save moderation steps (currently this consists entirely of deleting comments which were disallowed).

## 1.7 Example of using the comments app

Follow the first three steps of the *quick start guide*.

Now suppose, you have an app (`blog`) with a model (`Post`) to which you want to attach comments. Let's also suppose that you have a template called `blog_detail.html` where you want to display the comments list and comment form.

### 1.7.1 Template

First, we should load the `comment` template tags in the `blog_detail.html` so that we can use its functionality. So just like all other custom template tag libraries:

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

Next, let's add the number of comments attached to the particular model instance of `Post`. For this we assume that a context variable `object_pk` is present which gives the `id` of the instance of `Post`.

The usage of the *get_comment_count* tag is like below:

```
{% get_comment_count for blog.post object_pk as comment_count %}
<p>{{ comment_count }} comments have been posted.</p>
```

If you have the instance (say `entry`) of the model (`Post`) available in the context, then you can refer to it directly:

```
{% get_comment_count for entry as comment_count %}
<p>{{ comment_count }} comments have been posted.</p>
```

Next, we can use the *render_comment_list* tag, to render all comments to the given instance (`entry`) by using the `comments/list.html` template:

```
{% render_comment_list for entry %}
```

Django will will look for the `list.html` under the following directories (for our example):

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

To get a list of comments, we make use of the *get_comment_list* tag. Using this tag is very similar to the *get_comment_count* tag. We need to remember that *get_comment_list* returns a list of comments and hence we have to iterate through them to display them:

```
{% get_comment_list for blog.post object_pk as comment_list %}
{% for comment in comment_list %}
<p>Posted by: {{ comment.user_name }} on {{ comment.submit_date }}</p>
...
<p>Comment: {{ comment.comment }}</p>
...
{% endfor %}
```

Finally, we display the comment form, enabling users to enter their comments. There are two ways of doing so. The first is when you want to display the comments template available under your `comments/form.html`. The other method gives you a chance to customize the form.

The first method makes use of the *render_comment_form* tag. Its usage too is similar to the other three tags we have discussed above:

```
{% render_comment_form for entry %}
```

It looks for the `form.html` under the following directories (for our example):

```
comments/blog/post/form.html
comments/blog/form.html
comments/form.html
```

Since we customize the form in the second method, we make use of another tag called *comment_form_target*. This tag on rendering gives the URL where the comment form is posted. Without any *customization*, *comment_form_target* evaluates to `/comments/post/`. We use this tag in the form's `action` attribute.

The *get_comment_form* tag renders a `form` for a model instance by creating a context variable. One can iterate over the `form` object to get individual fields. This gives you fine-grain control over the form:

```
{% for field in form %}
{% if field.name == "comment" %}
  <!-- Customize the "comment" field, say, make CSS changes -->
...
{% endfor %}
```

But let's look at a simple example:

```
{% get_comment_form for entry as form %}
<!-- A context variable called form is created with the necessary hidden
fields, timestamps and security hashes -->
<table>
  <form action="{% comment_form_target %}" method="post">
    {% csrf_token %}
    {{ form }}
    <tr>
      <td colspan="2">
        <input type="submit" name="submit" value="Post">
        <input type="submit" name="preview" value="Preview">
      </td>
    </tr>
  </form>
</table>
```

### 1.7.2 Flagging

If you want your users to be able to flag comments (say for profanity), you can just direct them (by placing a link in your comment list) to `/flag/{{ comment.id }}/`. Similarly, a user with requisite permissions (`"Can moderate comments"`) can approve and delete comments. This can also be done through the `admin` as you'll see later. You might also want to customize the following templates:

- `flag.html`
- `flagged.html`
- `approve.html`
- `approved.html`
- `delete.html`
- `deleted.html`

found under the directory structure we saw for `form.html`.

### 1.7.3 Feeds

Suppose you want to export a feed of the latest comments, you can use the built-in `LatestCommentFeed`. Just enable it in your project's `urls.py`:

```
from django.conf.urls import url
from django_comments.feeds import LatestCommentFeed

urlpatterns = [
# ...
    url(r'^feeds/latest/$', LatestCommentFeed()),
# ...
]
```

Now you should have the latest comment feeds being served off `/feeds/latest/`.

### 1.7.4 Moderation

Now that we have the comments framework working, we might want to have some moderation setup to administer the comments. The comments framework comes with *generic comment moderation*. The comment moderation has the following features (all of which or only certain can be enabled):

- Enable comments for a particular model instance.
- Close comments after a particular (user-defined) number of days.
- Email new comments to the site-staff.

To enable comment moderation, we subclass the `CommentModerator` and register it with the moderation features we want. Let's suppose we want to close comments after 7 days of posting and also send out an email to the site staff. In `blog/models.py`, we register a comment moderator in the following way:

```
from django.db import models
from django_comments.moderation import CommentModerator, moderator

class Post(models.Model):
    title   = models.CharField(max_length = 255)
```

(continues on next page)

(continued from previous page)

```
    content = models.TextField()
    posted_date = models.DateTimeField()

class PostModerator(CommentModerator):
    email_notification = True
    auto_close_field   = 'posted_date'
    # Close the comments after 7 days.
    close_after        = 7

moderator.register(Post, PostModerator)
```

The generic comment moderation also has the facility to remove comments. These comments can then be moderated by any user who has access to the `admin` site and the `Can moderate comments` permission (can be set under the `Users` page in the `admin`).

The moderator can `Flag`, `Approve` or `Remove` comments using the `Action` drop-down in the `admin` under the `Comments` page.

---

**Note:** Only a super-user will be able to delete comments from the database. `Remove Comments` only sets the `is_public` attribute to `False`.

---

## 1.8 Settings that `django_comments` understands

### 1.8.1 COMMENTS_HIDE_REMOVED

If `True` (default), removed comments will be excluded from comment lists/counts (as taken from template tags). Otherwise, the template author is responsible for some sort of a "this comment has been removed by the site staff" message.

### 1.8.2 COMMENT_MAX_LENGTH

The maximum length of the comment field, in characters. Comments longer than this will be rejected. Defaults to 3000.

### 1.8.3 COMMENTS_APP

An app which provides *customization of the comments framework*. Use the same dotted-string notation as in `INSTALLED_APPS`. Your custom *`COMMENTS_APP`* must also be listed in `INSTALLED_APPS`.

### 1.8.4 COMMENT_TIMEOUT

The maximum comment form timeout in seconds. The default value is `2 * 60 * 60` (2 hours).

## 1.9 Porting to `django_comments` from `django.contrib.comments`

To move from `django.contrib.comments` to `django_comments`, follow these steps:

1. Install the comments app by running `pip install django-contrib-comments`.
2. In `INSTALLED_APPS`, replace `'django.contrib.comments'` with `'django_comments'`.

```
INSTALLED_APPS = (
    ...
    'django_comments',
    ...
)
```

3. In your project's `urls.py`, replace the url include `django.contrib.comments.urls` with `'django_comments.urls'`:

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

4. If your project had *customized the comments framework*, then update your imports to use the `django_comments` module instead of `django.contrib.comments`. For example:

```
from django.contrib.comments.models import Comment # old
from django_comments.models import Comment # new

from django.contrib.comments.forms import CommentForm # old
from django_comments.forms import CommentForm # new
```

5. If your database schema already contains the tables and data for existing comments and you get an error like `django.db.utils.ProgrammingError: relation "django_comments" already exists` in your first subsequent migration, run `manage.py migrate django_comments --fake-initial`.

# Python Module Index

## d

# Index