


# A Simple Forum - Part I

Forums are one of the cornerstones of the Web — they are integral for organizing niche communities for anything from Orchids to DIY audio components. After reading this tutorial, you'll be able to make one!

## Defining the Model

Let's start by defining the model classes (in forum/models.py):

```
from django.db import models
from django.contrib.auth.models import User
from django.contrib import admin
from string import join
from settings import MEDIA_ROOT

class Forum(models.Model):
    title = models.CharField(max_length=60)
    def __unicode__(self):
        return self.title

class Thread(models.Model):
    title = models.CharField(max_length=60)
    created = models.DateTimeField(auto_now_add=True)
    creator = models.ForeignKey(User, blank=True, null=True)
    forum = models.ForeignKey(Forum)

    def __unicode__(self):
        return unicode(self.creator) + " - " + self.title

class Post(models.Model):
    title = models.CharField(max_length=60)
    created = models.DateTimeField(auto_now_add=True)
    creator = models.ForeignKey(User, blank=True, null=True)
    thread = models.ForeignKey(Thread)
    body = models.TextField(max_length=10000)

    def __unicode__(self):
        return u"%s - %s - %s" % (self.creator, self.thread, self.title)

    def short(self):
        return u"%s - %s\n%s" % (self.creator, self.title, self.created.strftime("%b
    short.allow_tags = True

### Admin

class ForumAdmin(admin.ModelAdmin):
    pass

class ThreadAdmin(admin.ModelAdmin):
    list_display = ["title", "forum", "creator", "created"]
    list_filter = ["forum", "creator"]

class PostAdmin(admin.ModelAdmin):
    search_fields = ["title", "creator"]
    list_display = ["title", "thread", "creator", "created"]

admin.site.register(Forum, ForumAdmin)
admin.site.register(Thread, ThreadAdmin)
admin.site.register(Post, PostAdmin)
```

As always, you should run *manage.py syncdb; manage.py runserver* to add tables and start Django.

If you ever used a forum site, you already know that front page will show a list of available forums, each forum page will show a list of threads with the latest on top and each thread page will show a list of posts sorted with the latest showing up at the bottom. The forum page will also let you add a

new topic and the thread page will let you post a reply.

That's the most basic forum functionality in a nutshell and that will be our first task. To begin, let's outline the url, function and template naming scheme: frontpage — */forum/*, *main()* and *list.html*; forum — */forum/{id}/*, *forum()* and *forum.html*; thread: */forum/thread/{id}/*, *thread()* and *thread.html*.

Your *urlconf* lines and main listing view should be as follows:

```
(r"", "main"),
(r"^forum/(\d+)/$", "forum"),
(r"^thread/(\d+)/$", "thread"),
```

```
from django.core.urlresolvers import reverse
from settings import MEDIA_ROOT, MEDIA_URL

def main(request):
    """Main listing."""
    forums = Forum.objects.all()
    return render_to_response("forum/list.html", dict(forums=forums, user=request.use
```

Forum listings usually show the total number of posts as well as the author and subject of the latest post. That's something we can easily add to our models:

```
class Forum(models.Model):
    # ...

    def num_posts(self):
        return sum([t.num_posts() for t in self.thread_set.all()])

    def last_post(self):
        if self.thread_set.count():
            last = None
            for t in self.thread_set.all():
                l = t.last_post()
                if l:
                    if not last: last = l
                    elif l.created > last.created: last = l
            return last

class Thread(models.Model):
    # ...

    def num_posts(self):
        return self.post_set.count()

    def num_replies(self):
        return self.post_set.count() - 1

    def last_post(self):
        if self.post_set.count():
            return self.post_set.order_by("created")[0]
```

When a *ForeignKey* or a *Many-to-Many* relationship is created, the related model instances get an automatic handle to the original objects — in our case, *Thread* will have *self.post_set* QuerySet object (keep in mind it's not a list and can't be used as one!), and Forum will have *self.thread_set* object.

## Front Page

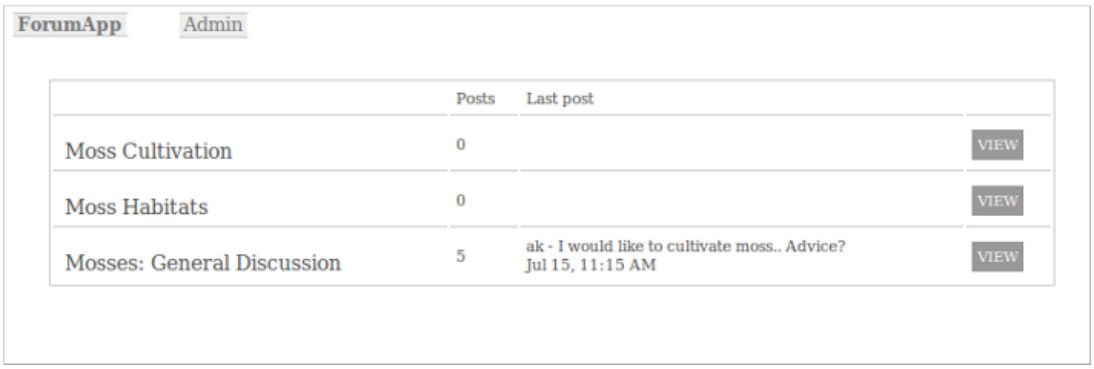
Here is the main loop of the template I used for main listing:

```
<!-- Forums  -->
<div id="list">
<table border="0" cellpadding="4" width="100%">
    <tr>
        <td></td>
        <td>Posts</td>
        <td>Last post</td>
        <td></td>
    </tr>

    {% for forum in forums %}
    <tr>
        <td {% if forloop.last %}class="last"{% endif %}>
        <div class="title"> <a href="{% url forum.views.forum forum.pk %}">{{ forum.t
            </div></td>
        <td {% if forloop.last %}class="last"{% endif %}>{{ forum.num_posts }}</td>
```

```
        <td {% if forloop.last %}class="last"{% endif %}>
            {{ forum.last_post.short|linebreaksbr }}</td>
        <td {% if forloop.last %}class="last"{% endif %}>
            <a class="button" href="{% url forum.views.forum forum.pk %}">VIEW</a>
        </td>
    </tr>

    {% endfor %}
</div>
```

The *forloop.last* construct is used here to assign the css class that won't have a bottom border — the other rows will have one as you will soon see in the screenshot. Take a note of

*linebreaksbr* filter that converts new lines to *<br />* tags.




## Forum View

Now for the forum view.. Again, both view and template are called *forum*:

```
def add_csrf(request, ** kwargs):
    d = dict(user=request.user, ** kwargs)
    d.update(csrf(request))
    return d

def mk_paginator(request, items, num_items):
    """Create and return a paginator."""
    paginator = Paginator(items, num_items)
    try: page = int(request.GET.get("page", '1'))
    except ValueError: page = 1

    try:
        items = paginator.page(page)
    except (InvalidPage, EmptyPage):
        items = paginator.page(paginator.num_pages)
    return items


def forum(request, pk):
    """Listing of threads in a forum."""
    threads = Thread.objects.filter(forum=pk).order_by("-created")
    threads = mk_paginator(request, threads, 20)
    return render_to_response("forum/forum.html", add_csrf(request, threads=threads, p
```

I've separated functions that add csrf dictionary and create paginator since we'll need to use these often.

The template is fairly close to the main listing but now we need pagination and a button to add a new topic:

```
<!-- Threads  -->
<a id="new_topic" class="buttont" href=
"{% url forum.views.post 'new_thread' pk %}">Start New Topic</a>
<br />
<br />

<div id="list">
<table border="0" cellpadding="4" width="100%">
    <tr>
        <td>Topics</td>
        <td>Replies</td>
        <td>Last post</td>
        <td></td>
    </tr>
```

```
    {% for thread in threads.object_list %}
    <tr>
        <td {% if forloop.last %}class="last"{% endif %}>
        <div class="title"> <a href="{% url forum.views.thread thread.pk %}">{{ thread
            </div></td>
        <td {% if forloop.last %}class="last"{% endif %}>{{ thread.num_replies }}</td>
        <td {% if forloop.last %}class="last"{% endif %}>
            {{ thread.last_post.short|linebreaksbr }}</td>
        <td {% if forloop.last %}class="last"{% endif %}>
            <a class="button" href="{% url forum.views.thread thread.pk %}">VIEW</a>
        </td>
    </tr>

    {% endfor %}
</table>
</div>

<!-- Next/Prev page links  -->
{% if threads.object_list and threads.paginator.num_pages > 1 %}
<div class="pagination">
    <span class="step-links">
        {% if threads.has_previous %}
            <a href= "?page={{ threads.previous_page_number }}">previous &lt;&lt; </a>
        {% endif %}

        <span class="current">
             Page {{ threads.number }} of {{ threads.paginator.num_pages }}
        </span>

        {% if threads.has_next %}
            <a href="?page={{ threads.next_page_number }}"> &gt;&gt; next</a>
        {% endif %}
    </span>
</div>
{% endif %}
```

## Thread View

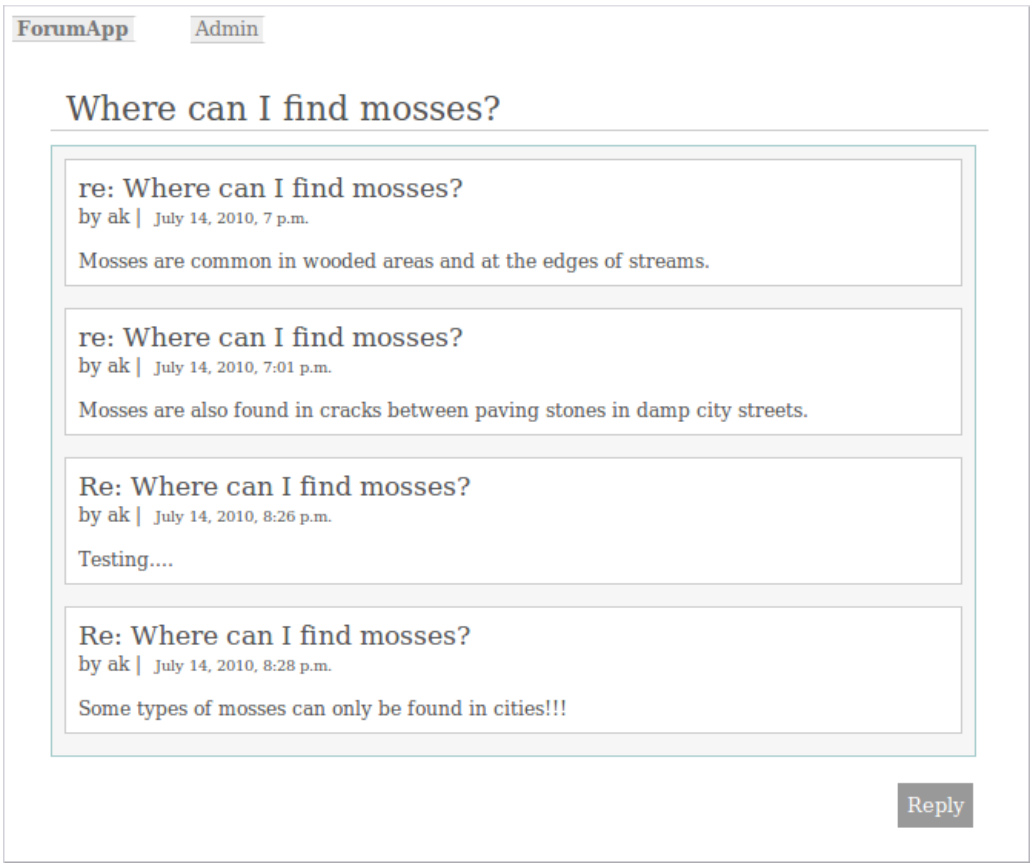
At last, our thread view (both function and template are called *thread*):

```
def thread(request, pk):
    """Listing of posts in a thread."""
    posts = Post.objects.filter(thread=pk).order_by("created")
    posts = mk_paginator(request, posts, 15)
    title = Thread.objects.get(pk=pk).title
    return render_to_response("forum/thread.html", add_csrf(request, posts=posts, pk=p
        title=title, media_url=MEDIA_URL))
```

As you can see, we're sorting in the opposite order compared to the forum view.

```
<!-- Posts  -->
<div class="ttitle">{{ title }}</div>
<div id="list">

    {% for post in posts.object_list %}
        <div class="post">
            <span class="title">{{ post.title }}</span><br />
            by {{ post.creator }} | <span class="date">{{ post.created }}</span> <br />
            {{ post.body }} <br />
        </div>
    {% endfor %}
</div>

<!-- Next/Prev page links
    ...
-->

<a class="button" href="{% url forum.views.post 'reply' pk %}">Reply</a>
```

I skipped pagination since it's the same as in previous listing. Here's what we have so far:


## Posting Replies and New Topics

Of course, we also need to add a way to post replies and new threads. I'll use the same template for both and call it *post.html* and the method names will be: *post()* to show the form and *new_thread()* and *reply()* to submit; urls will be: */forum/post/(new_thread|reply)/{id}/* and */forum/new_thread/{id}/* and */forum/reply/{id}/*. I've added these *urlconf* lines:

```
(r"^post/(new_thread|reply)/(\d+)/$", "post"),
(r"^reply/(\d+)/$", "reply"),
(r"^new_thread/(\d+)/$", "new_thread"),
```

...and *post()*:

```
def post(request, ptype, pk):
    """Display a post form."""
    action = reverse("dbe.forum.views.%s" % ptype, args=[pk])
    if ptype == "new_thread":
        title = "Start New Topic"
        subject = ''
    elif ptype == "reply":
        title = "Reply"
        subject = "Re: " + Thread.objects.get(pk=pk).title

    return render_to_response("forum/post.html", add_csrf(request, subject=subject,
        action=action, title=title))
```

When adding a new Topic, we'll need both the subject and body text — I will silently return to forum

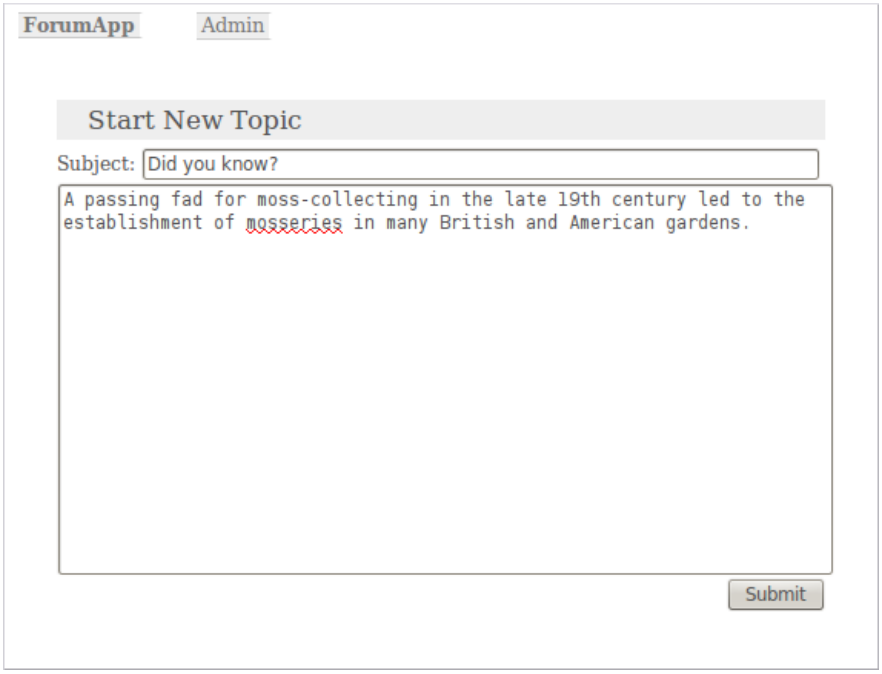
listing otherwise to keep things simple for the tutorial (although normally you should show an error and highlight required fields).

```
def new_thread(request, pk):
    """Start a new thread."""
    p = request.POST
    if p["subject"] and p["body"]:
        forum = Forum.objects.get(pk=pk)
        thread = Thread.objects.create(forum=forum, title=p["subject"], creator=reques
        Post.objects.create(thread=thread, title=p["subject"], body=p["body"], creator
    return HttpResponseRedirect(reverse("dbe.forum.views.forum", args=[pk]))
```

Here we have the *reply()* function which is very similar to *new_thread()*:

```
def reply(request, pk):
    """Reply to a thread."""
    p = request.POST
    if p["body"]:
        thread = Thread.objects.get(pk=pk)
        post = Post.objects.create(thread=thread, title=p["subject"], body=p["body"],
            creator=request.user)
    return HttpResponseRedirect(reverse("dbe.forum.views.thread", args=[pk]) + "?page=
```

This is an outrage! Firefox does not know what a mossery is? I shall be writing a stern letter. (Also: passing? It may be so, for now, but it will surely come back!)




One tiny usability improvement: we already have a link on every page that goes to frontpage (it's the one that says *ForumApp* in upper left corner), but a user won't be able to go back from thread page to the forum page easily. We need to add these lines to *thread()* and *thread.html* to create a backlink:

```
t = Thread.objects.get(pk=pk)
return render_to_response("forum/thread.html", add_csrf(request, posts=posts, pk=pk, t
                                                        forum_pk=t.forum.pk))
```

```
<a href="{% url forum.views.forum forum_pk %}">&lt;&lt; back to list of topics</a>
```

Continue to part II




# A Simple Forum - Part II

## Profile Picture

Forums would look a little boring without ubiquitous profile pics. We already know how to add images using Admin interface from the last tutorial, but doing the same from regular views involves a bit more work.

To get us started, let's set out the naming scheme we'll use: url, function and template will all be called *profile* and both url and function for saving the image will be called *save_profile*; model class will be *UserProfile*. We have to add a link to *fbase.html*:

```
{% if user.is_authenticated %}
    <a href="{% url forum.views.profile user.pk %}">Edit profile</a> {% endif %}
```

Here's our *UserProfile* class:

```
class UserProfile(models.Model):
    avatar = models.ImageField("Profile Pic", upload_to="images/", blank=True, null=Tr
    posts = models.IntegerField(default=0)
    user = models.ForeignKey(User, unique=True)

    def __unicode__(self):
        return unicode(self.user)
```

Don't forget to run *manage.py syncdb* to save the table!

We already used a ModelForm in MyBlog tutorial — that part should be familiar to you. In *profile()*, we're sending the image URL to the template if it already exists:

```
from PIL import Image as PImage
from os.path import join as pjoin

class ProfileForm(ModelForm):
    class Meta:
        model = UserProfile
        exclude = ["posts", "user"]

@login_required
def profile(request, pk):
    """Edit user profile."""
    profile = UserProfile.objects.get(user=pk)
    img = None

    if request.method == "POST":
        pf = ProfileForm(request.POST, request.FILES, instance=profile)
        if pf.is_valid():
            pf.save()
            # resize and save image under same filename
            imfn = pjoin(MEDIA_ROOT, profile.avatar.name)
            im = PImage.open(imfn)
            im.thumbnail((160,160), PImage.ANTIALIAS)
            im.save(imfn, "JPEG")
    else:
        pf = ProfileForm(instance=profile)

    if profile.avatar:
        img = "/media/" + profile.avatar.name
    return render_to_response("forum/profile.html", add_csrf(request, pf=pf, img=img))
```

*profile.html:*

```
<div id="rtitle">Edit Profile</div><br />
```

```
<p>
<form enctype="multipart/form-data" action="{% url forum.views.save_profile user.pk %}
    {% csrf_token %}
    {% if img %} <p><img border="0" alt="Profile Pic" src="{{ img }}" /></p> {% endif

    Profile Pic: {{ pf.avatar }}
    <input type="submit" value="Submit" id="submit" />
</form>
</p>
```

It's *very* important that the form tag should include the *enctype* property as shown. It's particularly important to remember because there won't be any error if you omit it — the file simply won't be saved (guess if I forgot it while writing this tutorial!).

I haven't talked in depth about django forms yet and I won't now, but I'd like to point out that Django gives you the choice between rendering the full form (except for the form opening, form closing and submit tags) and formatting each field separately. To have Django show the full form, I would just put *{{ pf }}* right before the submit tag. In this case, that would work almost as well as it does now except that there would be no space between semicolon and the image input field. Since we only have the single form field to show, it's very easy to add the Label and the field manually.

It's also very important not to forget to add the *request.FILES* argument because Django won't warn you or show any error. To keep things simple I'm assuming we'll be dealing with JPEG images although in a real forum you'd want to also handle PNG and GIF files.

As always, we have to add the *urlconf* line:

```
(r"^profile/(\d+)/$", "profile"),
```

That what we have so far, with this wonderful picture of Moss I got off Wikipedia uploaded and resized:


## Showing Profile Data in Posts

Of course, this is a bit useless if the profile pic doesn't show up in actual posts in a thread. Usually forums will also add a bit of additional information like the number of posts and date joined under the profile pic.

I could do this from the template but I think it's clearer and easier when this code is in *models.py*:

```
class Post:
    # ...
    def profile_data(self):
        p = self.creator.userprofile_set.all()[0]
        return p.posts, p.avatar
```

In *thread.html*, we'll add this block of code:

```
<div class="ppic">
    {% with post.profile_data as pdata %}
        {% if pdata.1 %}
            <img src="{{ media_root }}{{ pdata.1 }}" /> <br />
        {% endif %}
    {{ post.creator }}<br />
    Posts: {{ pdata.0 }}<br />
    Joined: {{ post.creator.date_joined|date:"M d Y" }}
```

```
    {% endwith %}
</div>
```

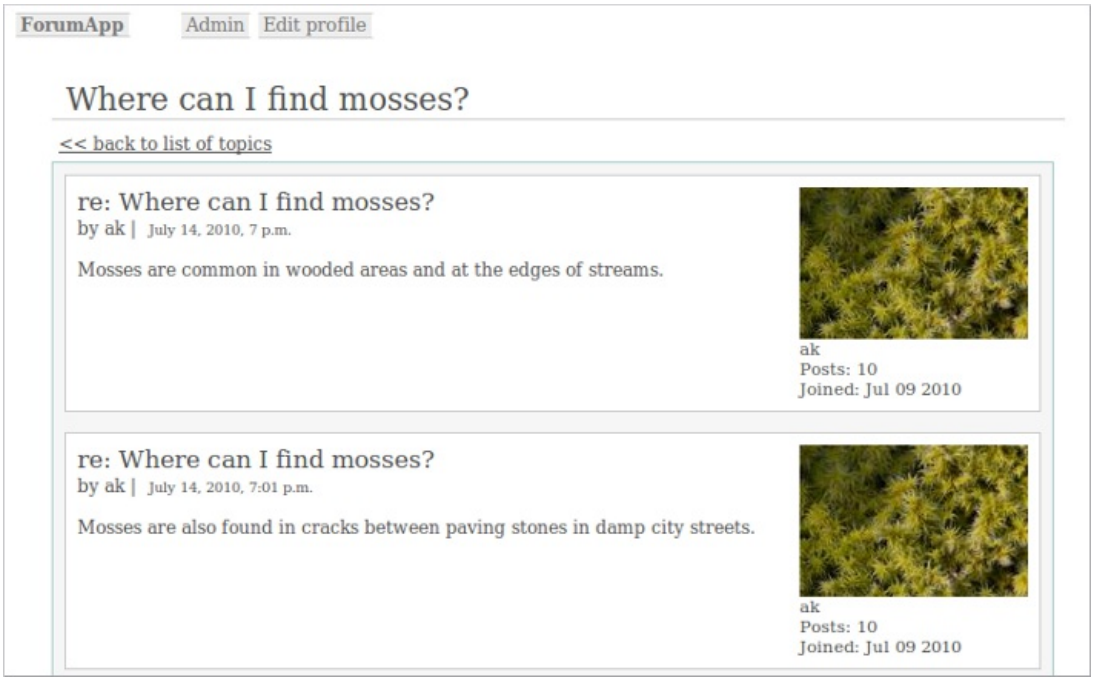
I'm using *with* tag to make code a little shorter and *date* filter to only show the date joined, because *date_joined* stores both date and time.

We also have to make sure we increment the *posts* counter every time a user replies or creates a thread:

```
def increment_post_counter(request):
    profile = request.user.userprofile_set.all()[0]
    profile.posts += 1
    profile.save()

def new_thread(request, pk):
    """Start a new thread."""
    p = request.POST
    if p["subject"] and p["body"]:
        forum = Forum.objects.get(pk=pk)
        thread = Thread.objects.create(forum=forum, title=p["subject"], creator=reques
        Post.objects.create(thread=thread, title=p["subject"], body=p["body"], creato
        increment_post_counter(request)
    return HttpResponseRedirect(reverse("dbe.forum.views.forum", args=[pk]))
```

The same call to *increment_post_counter()* should be added to *reply()* view as well.

Here's what we have now:




Continue to part III

Django by Example » search page || previous | next




# A Simple Forum - Part III

It's good to have a forum of your own, post new topics, reply to oneself, start flamewars. All of that is great, but what if.. you would like other users to be able to participate, as well? That will be our last task in this tutorial: user registration.

## User Registration

There is already an existing package for Django that handles user registration — all we have to do is set it up for our needs. If you are on Ubuntu / Debian, you can install it by running *apt-get install django-registration*.

Here is the outline of what we'll need to do to set things up for this package: we'll need to add it to *settings.py*, we'll add *register*, *login* and *logout* links to our own *fbase.html* template, add one *urlconf* line to the base *urls.py* module of our site and finally we'll add a new directory with registration templates.

Here are the changes you'll need to do:

*settings.py*:

```
INSTALLED_APPS = (
    # ...
    'dbe.forum',
    'registration',
)
```

*urls.py* (note: this is the base *urlconf* file located under *dbe/*, **not** under *dbe/forum/*):

```
(r'^accounts/', include('registration.urls')),
```

New links in *fbase.html*:

```
{% if not user.is_authenticated %}<a href="/accounts/login/?next=/forum/">login</a> |
    href="/accounts/register/">register</a>{% endif %}

{% if user.is_authenticated %}<a href="/accounts/logout/?next=/forum/">logout</a>
    {% endif %}
```

Files under *templates/registration/*:

*activate.html*:

```
<html>
    <head>
        <title>Activation</title>
    </head>
    <body>
        <h2>Your account is now active!</h2>
        <a href="/accounts/login/?next=/forum/">Login</a>
    </body>
</html>
```

*activation_complete.html*:

```
<html>
    <head>
        <title>Activation complete</title>
    </head>
    <body>
        <h4>Activation complete! Your account is now active.</h4>
```

```
    </body>
</html>
```

*activation_email.txt*:

```
Your activation key: {{ activation_key }}

Please use this link to activate your account:

http://localhost:8000/accounts/activate/{{ activation_key }}/

After activation you can login to the site:

http://localhost:8000/accounts/login/
```

*activation_email_subject.txt*:

```
Activate your account
```

*login.html*:

```
{% extends "admin/base_site.html" %}
{% load i18n %}

{% block extrastyle %}{% load adminmedia %}{{ block.super }}
    <link rel="stylesheet" type="text/css" href="{% admin_media_prefix %}css/login.cs
{% endblock %}

{% block bodyclass %}login{% endblock %}

{% block content_title %}{% endblock %}

{% block breadcrumbs %}{% endblock %}

{% block content %}
{% if error_message %}
<p class="errornote">{{ error_message }}</p>
{% endif %}
<div id="content-main">
    <form action="{{ app_path }}" method="post" id="login-form">{% csrf_token %}
  <div class="form-row">
    <label for="id_username">{% trans 'Username:' %}</label>
        <input type="text" name="username" id="id_username" />
  </div>
  <div class="form-row">
    <label for="id_password">{% trans 'Password:' %}</label>
        <input type="password" name="password" id="id_password" />
    <input type="hidden" name="this_is_the_login_form" value="1" />
  </div>
  <div class="submit-row">
    <label> </label><input type="submit" value="{% trans 'Log in' %}" />
  </div>
</form>

<script type="text/javascript">
document.getElementById('id_username').focus()
</script>
</div>
{% endblock %}
```

*registration_complete.html*:

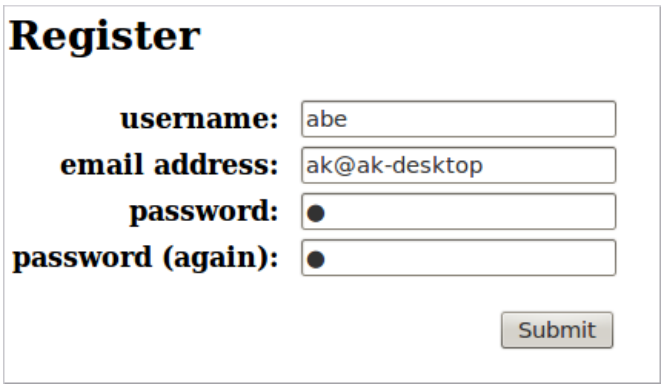
```
<html>
    <head>
        <title>Registration Complete</title>
    </head>
    <body>
        <h2>Registration Complete!</h2>
    </body>
</html>
```

*registration_form.html*:

```
<html>
    <head>
        <title>Register</title>
    </head>
    <body>
        <h2>Register</h2>
```

```
        <form action="/accounts/register/" method="POST">{% csrf_token %}
            <table>
            {{ form.as_table }}
            </table>
        <p>
            <input type="submit" value="Submit" />
        </p>
        </form>
    </body>
</html>
```

Screenshots of registration form and confirmation email:




```
i:Exit  -:PrevPg  <Space>:NextPg v:View Attachm.  d:Del  r:Reply  j:Next ?:Help
Date: Mon, 19 Jul 2010 15:06:37 -0000
From: webmaster@localhost
To: ak@ak-desktop.org
Subject: Activate your account

Your activation key: 2b647128c2e095f214a0560985700c18b38d647a

Please use this link to activate your account:

http://localhost:8000/accounts/activate/2b647128c2e095f214a0560985700c18b38d647a/

After activation you can login to the site:

http://localhost:8000/accounts/login/
```

One last thing we should do is automatic creation of a *UserProfile* whenever a new user is created. If *User* was a regular type of class defined in our *models.py*, you would already know how to override *save()* method to create a profile along with user. There's just one little problem.. *User* is a part of Admin — how can we change what it does when saving itself?! One way to do this would be to make a custom version of the Admin but there's an easier way using Django *signals*. The idea is very simple: every time a certain model is saved, a *post_save* signal is sent and associated function will be run:

```
from django.db.models.signals import post_save

def create_user_profile(sender, **kwargs):
    """When creating a new user, make a profile for him or her."""
    u = kwargs["instance"]
    if not UserProfile.objects.filter(user=u):
        UserProfile(user=u).save()

post_save.connect(create_user_profile, sender=User)
```

(This code should go into *models.py*). We have to check if *UserProfile* already exists because this function will run every time changes to *User* are saved.

## Automated Testing

Automated testing can be tremendously useful with anything beyond the most basic web app. The following code shows how to test basic functionality of the Forum app using Django's test client. This code should live in *tests.py* under *dbe/forum/*:

```
from django.test import TestCase
```

```
from django.test.client import Client
from django.contrib.auth.models import User
from django.contrib.sites.models import Site

from dbe.forum.models import *

class SimpleTest(TestCase):
    def setUp(self):
        f = Forum.objects.create(title="forum")
        u = User.objects.create_user("ak", "ak@abc.org", "pwd")
        Site.objects.create(domain="test.org", name="test.org")
        t = Thread.objects.create(title="thread", creator=u, forum=f)
        p = Post.objects.create(title="post", body="body", creator=u, thread=t)

    def content_test(self, url, values):
        """Get content of url and test that each of items in `values` list is present
        r = self.c.get(url)
        self.assertEquals(r.status_code, 200)
        for v in values:
            self.assertTrue(v in r.content)

    def test(self):
        self.c = Client()
        self.c.login(username="ak", password="pwd")

        self.content_test("/forum/", ['<a href="/forum/forum/1/">forum</a>'])
        self.content_test("/forum/forum/1/", ['<a href="/forum/thread/1/">thread</a>',

        self.content_test("/forum/thread/1/", ['<div class="ttitle">thread</div>',
               '<span class="title">post</span>', 'body <br />', 'by ak |'])

        r = self.c.post("/forum/new_thread/1/", {"subject": "thread2", "body": "body2"
        r = self.c.post("/forum/reply/2/", {"subject": "post2", "body": "body3"})
        self.content_test("/forum/thread/2/", ['<div class="ttitle">thread2</div>',
               '<span class="title">post2</span>', 'body2 <br />', 'body3 <br />'])
```

To run it, do: *manage.py test forum*. Django test framework creates a separate, blank set of database tables for testing — when the test is run, it's up to you to create all records beforehand, including user accounts.

Instead of using standard *create()* method to create a user, we have to use special *create_user()* method because the password is stored in hashed form; *create_user()* takes care of that automatically.

The *test()* function creates a client object that logs in and accesses views similarly to how a user's browser would use the App (technically, it works in a rather different way but we're not concerned with that here). You can use client object to make *GET* and *POST* requests and check if you get proper response status codes and contents.

Client object *post()* method accepts the url and a dictionary that will go into *request.POST* dictionary. In the code sample above, *r.contents* will have full HTML code of the page as sent to the browser, letting us test for presence of links, elements, data, etc.

This is a very basic example of testing, there's much more to it! — make sure you take a quick look through Django Documentation's Chapter on Testing.

## Forum Search

Adding an efficient full-text search is specific to the database you're using, although for small and light traffic forums you might get away with using *.filter(body__icontains=myquery)* — simple SQL search. Another option is to use a Google search box restricted to your domain:

```
<form method="get" action="http://www.google.com/search">
    <input type="text"   name="q" size="31" maxlength="255" value="" />
    <input type="submit" value="Google Search" />
    <input type="radio"  name="sitesearch" value="" /> The Web
    <input type="radio"  name="sitesearch" value="mydomain.com" checked /> My Site<br
</form>
```

That's all for *A Simple Forum* tutorial. Hope you enjoyed it!

Download Full Sources

The Moss photo was created by: KirinX

Django by Example » search page || previous | next