Learning Objectives

- **Differentiate between one-step and two-step activation**
- **Add login page and a profile users see once logged in**
- **Create two-step activation for your blog**
- **Send users an activation email**
- **Set a window for which users must activate their account**
- **Remove users who have not activated their accounts**

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the `blango` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a `blango` directory appear in the file tree.

You are now ready for the next assignment.

Introduction and Logging In

Introduction

We're missing something that's vital for any website with user content: a way for users to register! Django provides views for logging in and out of your application, but not for registering. We'll use a third-party library called Django Registration to set up a registration system. This helps us by providing URLs, views and forms to support registration, so we only need to wire these up and write a few templates to get a registration system.

Django Registration supports both a "one-step" workflow, and a "two-step activation" workflow. With the one-step workflow, a user registers with their username and password (or in the case of Blango, email address and password). The user is created and active straight away, and the user is logged in.

The two-step activation workflow requires users to validate their email address before logging in. This is to help prevent automated users from registering for the site.

▼ **Valid email**
Making sure a user has a valid email address is good for other reasons too. We know with a valid email address that we have a way to contact a user if we need to. Also, some mail servers will start rejecting emails if the sender sends out too many messages that don't get delivered. If you don't first validate the addresses you might run into this problem.

After the user registers with a username, email address and password, they're sent a validation email with a link that contains a key. Once they visit the link, the email address is validated and they can log in. Once again, note that with Blango, they wouldn't have to provide a username, just an email address and password.

▼ **Two-step activation**
The key for the two-step activation workflow actually contains the username in an encoded format. By decoding the key and checking when the user signed up request can be validated without any extra data needing to be stored in the database.

We'll return to registration a little bit later.

So far, to log in to Blango, we've had to go through the Django Admin site. This is obviously not ideal, especially since it limits logging in to admin/staff users only! We'll start by setting up Django's built-in login system, to give a better user experience. You might have used this before, if you've built a Django site where users can log in, so we won't go over the process in great detail. We'll just quickly do the minimum required to allow a user to log in.

Try It Out

First we're going to start in the `base.html` template. The title of the page has been stuck at *Hello, world!* for too long, so let's make it customizable in each template. Just change the `<title>` element from:

```
<title>Hello, world!</title>
```

to

```
<title>{% block title %}Welcome to Blango{% endblock %}</title>
```

Then we'll add a Bootstrap Navbar. Insert this HTML in the `<body>` but above the `{% block content %}` tag.

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
    <div class="container-fluid">
        <a class="navbar-brand" href="/">Blango Home</a>
        <div class="d-flex">
            {% if request.user.is_active %}
                <a class="nav-link" href="{% url "profile"
        %}">Profile</a>
                <a class="nav-link" href="{% url "logout"
        %}">Log Out</a>
            {% else %}
                <a class="nav-link" href="{% url "login" %}">Log
        In</a>
            {% endif %}
        </div>
    </div>
</nav>
```

This will give us a link to the profile page or to log out, for a logged-in user. For anonymous users, they'll get a link to the log-in page.

Profile View

Profile View

Next we'll set up the `blango_auth` templates directory. Create a `templates` directory inside the `blango_auth` directory, and then a `blango_auth` directory inside that.

We're going to have a `profile` view that users will see after logging in. It will just render a `profile.html` template. Create a file called `profile.html` inside the `blango_auth` directory you just created and add this content to it:

Open profile.html

```
{% extends "base.html" %}
{% load blog_extras %}
{% block title %}Blango Profile{% endblock %}
{% block content %}
{% row %}
    {% col %}
        <p>Logged in as {{ request.user }}.</p>
        <p><a href="{% url "logout" %}">Log Out</a></p>
    {% endcol %}
{% endrow %}
{% endblock content %}
```

While we're working on templates, we need to add a `registration/login.html` template that Django will render to show the login form. Create a directory called `registration` inside the `blango_auth/templates` directory. Then, inside that, create a file called `login.html`. Paste this content into it:

Open login.html

```
{% extends "base.html" %}
{% load crispy_forms_tags blog_extras %}
{% block title %}Log In to Blango{% endblock %}
{% block content %}
{% row "justify-content-center" %}
    {% col "col-md-6" %}
    {% if next %}
        {% if user.is_authenticated %}
        <p>Your account doesn't have access to this page. To
        proceed,
        please login with an account that has access.</p>
        {% else %}
        <p>Please login to see this page.</p>
        {% endif %}
    {% endif %}
    {% endcol %}
{% endrow %}

{% row "justify-content-center" %}
    {% col "col-md-6" %}
        <form method="post" action="{% url "login" %}">
        {% csrf_token %}
        {{ form|crispy }}
        <button type="submit" class="btn btn-primary">Log
        In</button>
        <input type="hidden" name="next" value="{{ next }}">
        </form>

        <p><a href="{% url 'password_reset' %}">Lost password?
        </a></p>
    {% endcol %}
{% endrow %}
{% endblock %}
```

We're adding the class `justify-content-center` which will center the columns in the middle of the screen, which looks a bit nicer. The columns have the class `col-md-6` so they will be 50% width above the medium breakpoint but full width below it.

Next we need a `profile` view that renders the `profile.html` template. Open up `blango_auth/views.py`, and insert this code:

Open blango_auth/views.py

```
from django.contrib.auth.decorators import login_required
from django.shortcuts import render


@login_required
def profile(request):
    return render(request, "blango_auth/profile.html")
```

Note the use of the `login_required` decorator so that we know we have a logged-in user in that view.

Finally, we need to set up the URL routes. Open `urls.py`. Import the `blango_auth.views` module:

Open urls.py

```
import blango_auth.views
```

And then add these two URL routes inside `urlpatterns`. First, `account/` to `django.contrib.auth.urls`:

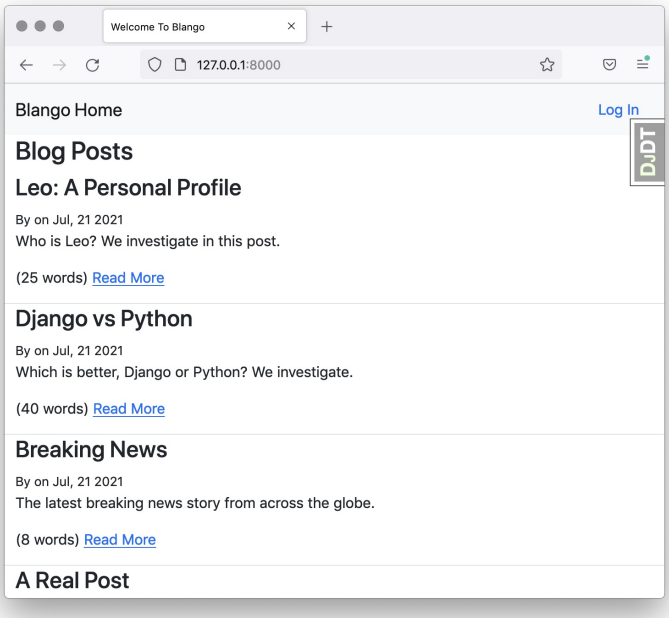
```
path("accounts/", include("django.contrib.auth.urls")),
```

Then `accounts/profile/` to our profile view, with the name `profile`.

```
path("accounts/profile/", blango_auth.views.profile,
        name="profile"),
```

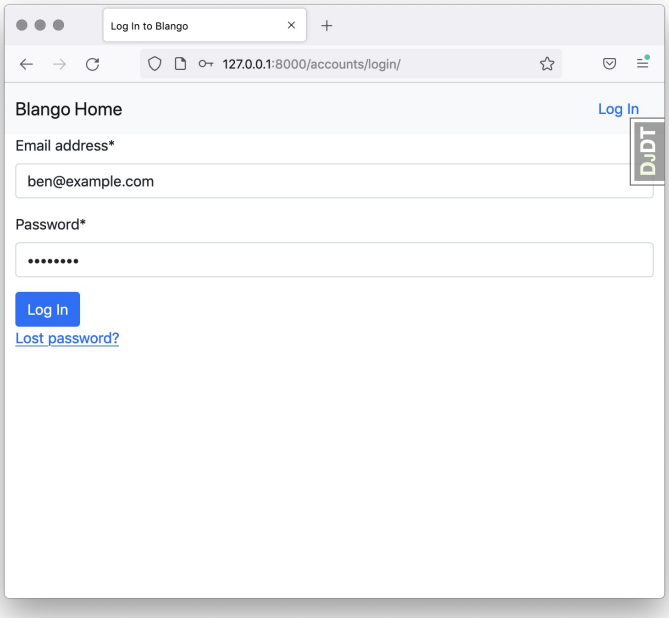
View Blog

We're ready to test it out. Head to the Blango main page and you should see the navbar, with either *Profile* and *Log Out* links, if you're logged in, or a *Log In* link if you're not.



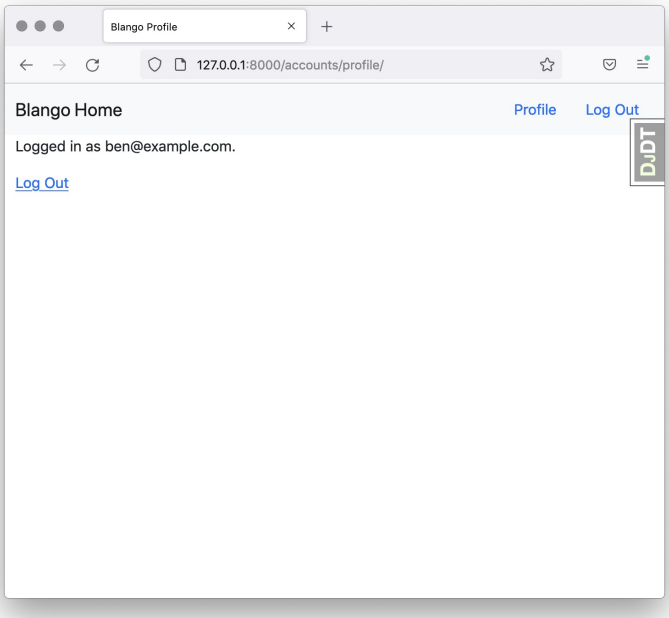
login link

Log out, if you're logged in, then return to the main page and click the **Log In** link. You'll see the login page.



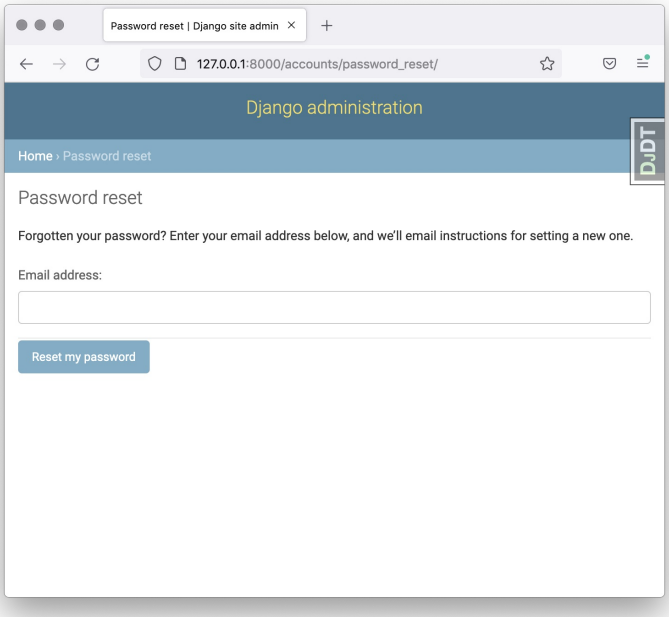
login form

Try logging in – you should be able to log in as both an admin or non-admin user. Once logged in you'll see the profile page which shows the email address of the current user.



profile

As we mentioned, we have just done the minimum to get a login system working. If you click on **Log Out** you'll see the standard Django Admin logged out page. Similarly, if you click the **Lost password?** link on the login page, you'll see the standard Django Admin Password Reset page.



django default logout page

▼ **Overriding the look**

If you want to override the look of these pages, you can do so by following

the guide in the Django Authentication Documentation. For our purposes, it doesn't matter that these pages don't look quite right. If you do create your own templates, you'll have to move `blango_auth` to be before `django.contrib.admin` in the `INSTALLED_APPS` setting. If you don't do this, then Django will load the Admin apps's templates as they have the same name. Moving `blango_auth` earlier in the list gives it precedence.

That's our minimum Django login system setup. We'll now return back to Django Registration.

One-Step Workflow

Django Registration One-Step Workflow

We've already discussed the different types of workflow for Django Registration. Since the one-step workflow is pretty easy to implement we'll just give a brief overview of how to set it up, but **we won't actually implement it** in Blango.

First, start by installing `django-registration` with `pip`. Then, add a URL pattern to include Django Registration's one step URLs:

```
path('accounts/',
        include('django_registration.backends.one_step.urls')),
```

You'll then need to add some templates for Django Registration to render. They are:

- `django_registration/registration_form.html`: This template is used to show the registration form, which is passed to the template in the `form` context variable.
- `django_registration/registration_complete.html`: This template is rendered and shown after the registration is successful.
- `django_registration/registration_closed.html`: Django Registration has the ability to disable registration by adding `REGISTRATION_OPEN = False` to your `settings.py`. If that's done, then the `registration_closed.html` template will be shown instead of the registration form.

If you're using the built-in Django User model that's all that needs to be done. However, if you've used a custom User model, like in Blango, you'll have to also create a `RegistrationForm` subclass and set the model class to the custom User model.

For example:

```
from django_registration.forms import RegistrationForm

from blango_auth.models import User


class BlangoRegistrationForm(RegistrationForm):
    class Meta(RegistrationForm.Meta):
        model = User
```

Then a specific URL pattern needs to be set up to point to `RegistrationView`, so that the custom form can be passed to the view. For example, we'd add these two URL patterns instead:

```
from django_registration.backends.one_step.views import
        RegistrationView
from blango_auth.forms import BlangoRegistrationForm

urlpatterns = [
    …
    path(
        "accounts/register/",
        RegistrationView.as_view(form_class=BlangoRegistrationForm),

        name="django_registration_register",
    ),
    path("accounts/",
        include("django_registration.backends.one_step.urls")),
    …
]
```

If you have a very custom User model then you might need to make some more changes, and the Django Registration Custom User Models Documentation has more information. But in most cases, what we've seen is all that's necessary.

Now let's look at the two-step activation workflow.

Two-Step Activation Part 1

Two-Step Activation Workflow Part 1

Implementing the two-step activation workflow isn't that much extra work than the one-step workflow. We just have to add a few more templates. `registration_form.html` and `registration_closed.html` have the same purpose as in the one-step workflow. `registration_complete.html` is a little different. Instead of being shown when a user's account is active, it's shown after the user has completed the register form, and should have a message letting them know they should check their email to complete registration.

The other templates are:

- `django_registration/activation_failed.html`: Used when a user visits the validation link but it fails, for example if the key is invalid or has expired. The template will have access to the context variable `activation_error`, a dictionary containing error context information. We'll look at this in more detail later.
- `django_registration/activation_complete.html`: Rendered after activation is complete, it should let the user know that they can now log in.
- `django_registration/activation_email_subject.txt`: This will be rendered to create the subject of the email sent to the user. The template is passed a number of context variables that we'll look at later.
- `django_registration/activation_email_body.txt`: This is rendered to create the body of the email, it receives the same context variables at the email activation subject template.

We also have to set up the URL patterns, which can be as simple as adding a single rule:

```
path('accounts/',
        include('django_registration.backends.activation.urls'))
        ,
```

Otherwise, we follow the one-step example for working with a custom User model.

```
from django_registration.backends.activation.views import
        RegistrationView


from blango_auth.forms import BlangoRegistrationForm


urlpatterns = [
    …
    path(
        "accounts/register/",
        RegistrationView.as_view(form_class=BlangoRegistrationForm),


        name="django_registration_register",
    ),
    path("accounts/",
        include("django_registration.backends.activation.urls"))
        ,
    …
]
```

Finally we need to specify how long the activation key is valid for, by setting the `ACCOUNT_ACTIVATION_DAYS`. For example, for a seven-day validity:

```
ACCOUNT_ACTIVATION_DAYS = 7
```

So now that we've seen *what* to do, let's add it to Blango.

Try It Out

Start by installing `django-registration` with `pip`.

```
pip3 install django-registration
```

Before we can see the emails that Django Registration is sending out, we need to configure an email backend. For development, we'll use the console email backend, which just prints emails out to the terminal.

To set this up, add this to your Django settings in the `Dev` class:

Open settings.py

```
EMAIL_BACKEND = "django.core.mail.backends.console.EmailBackend"
```

▼ **SMTP server**
If you have access to an SMTP server that you'd prefer to use, feel free to substitute that into your settings instead of the console backend. Bear in

mind that for testing you might go through a few email addresses so make sure you have access to them!

And while you're in `settings.py`, add the `ACCOUNT_ACTIVATION_DAYS` setting:

```
ACCOUNT_ACTIVATION_DAYS = 7
```

Next, we'll create the templates. We'll do this in the order they're used. First, create a directory called `django_registration` inside the `blango_auth/templates` directory. Inside this new directory we'll create all the files for Two-Step Activation Part 1. Start by creating `registration_form.html`.

Open registration_form.html

You can create the template in any way you choose, just make sure it has a `<form>` that POSTs. We'll use Django Crispy Forms to build this form later, so you can use the Cripsy tags to render the form if you like.

You should come up with something like this:

```
{% extends "base.html" %}
{% load crispy_forms_tags blog_extras %}
{% block title %}Register for Blango{% endblock %}
{% block content %}
{% row "justify-content-center" %}
    {% col "col-md-6" %}
        <h2>Register for Blango</h2>
        {% crispy form %}
    {% endcol %}
{% endrow %}
{% endblock %}
```

Next, create `registration_complete.html` in the `django_registration` directory, which is shown after the user registers. It should contain a message that the user needs to check their email now. It doesn't get passed any context variables. Once again, you can come up with your own content/design, but something like this should work:

Open registration_complete.html

```
{% extends "base.html" %}
{% load crispy_forms_tags blog_extras %}
{% block title %}Registration Successful{% endblock %}
{% block content %}
{% row "justify-content-center" %}
    {% col "col-md-6" %}
        <h2>Your registration was successful</h2>
        <p>Check your email to validate your account.</p>
    {% endcol %}
{% endrow %}
{% endblock %}
```

Now we'll create the templates to render the email. First create `activation_email_subject.txt` in the `django_registration` directory. The context variables that are available in this template are:

Open activation_email_subject.txt

- `activation_key`: The activation key for the new account, as a string.
- `expiration_days`: The number of remaining days which the account can be validated; matches the `ACCOUNT_ACTIVATION_DAYS` setting.
- `request`: The `HttpRequest` object in which the user registered.
- `scheme`: The scheme of the HTTP request as a string, either `http` or `https`.
- `site`: An object representing the site on which the user registered, if using `django.contrib.sites` (which Blango isn't).
- `user`: The newly-created user object.

You can come up with your own subject content using these (you don't have to use any if you don't want). But it could be something like this:

```
Activate your Blango account! You have {{ expiration_days }}
days!
```

Next, create the `activation_email_body.txt` file for the email body. It is passed the same context variables as the `activation_email_subject.txt`. You can build a link back to the activation page like this:

Open activation_email_body.txt

```
{{ scheme }}://{{ request.get_host }}{% url
"django_registration_activate" activation_key %}
```

The rest of the template is up to you. Something like this will work:

```
Hi,

You registered for Blango, but you need to activate your account
        within {{ expiration_days }} days.


To do that, please visit this page:

{{ scheme }}://{{ request.get_host }}{% url
        "django_registration_activate" activation_key %}


Thanks,
The Blango Team
```

When the user visits the activation link with a valid key, the `activation_complete.html` template will be rendered. Create this file now. It is not passed any template variables. It should just tell the user that they can now log in, and perhaps contain a link to the login page. This will do the trick:

Open activation_complete.html

```
{% extends "base.html" %}
{% load crispy_forms_tags blog_extras %}
{% block title %}Activation Complete{% endblock %}
{% block content %}
{% row "justify-content-center" %}
    {% col "col-md-6" %}
        <h2>Activation Complete</h2>
        <p>Your account is now activated! You can now log in and
        use Blango.</p>
        <p><a href="{% url "login" %}">Log In</a></p>
    {% endcol %}
{% endrow %}
{% endblock %}
```

If the activation were to fail, namely due to an invalid key, the `activation_failed.html` template is rendered; create this file. It is passed the context variable `activation_error`, which is a dictionary containing information about why the activation failed. The most useful keys are:

Open activation_failed.html

- `code`: The failure exception code, a string.
- `message`: A message/description of the exception.

Code will be one of the following:

- `already_activated`: Indicates the account has already been activated.
- `bad_username`: Indicates the username decoded from the activation key

is invalid (does not correspond to any user account).
- `expired`: Indicates the account/activation key has expired.
- `invalid_key`: Generic indicator that the activation key was invalid.

You should be able to just show the user `activation_error.message` in the template.

For example:

```
{% extends "base.html" %}
{% load crispy_forms_tags blog_extras %}
{% block title %}Activation Failed{% endblock %}
{% block content %}
{% row "justify-content-center" %}
    {% col "col-md-6" %}
        <h2>Activation Failed</h2>
        <p>Sorry, we couldn't activate your account.</p>
        <p>{{ activation_error.message }}</p>
    {% endcol %}
{% endrow %}
{% endblock %}
```

Two-Step Activation Part 2

Two-Step Activation Workflow Part 2

Finally, for completion, create the `registration_closed.html` template, to show a message to users that registration is not currently available. Such as:

Open registration_closed.html

```
{% extends "base.html" %}
{% load crispy_forms_tags blog_extras %}
{% block title %}Registration Closed{% endblock %}
{% block content %}
{% row "justify-content-center" %}
    {% col "col-md-6" %}
        <h2>Registration is currently closed</h2>
    {% endcol %}
{% endrow %}
{% endblock %}
```

While we're editing templates, let's add a link to the register page in the navbar of `base.html`. Open it, and add a link to the `django_registration_register` named URL (which we'll create later), just above the link to the login page in the navbar:

Open base.html

```
{% else %}
    <a class="nav-link" href="{% url
        "django_registration_register" %}">Register</a>
    <a class="nav-link" href="{% url "login" %}">Log In</a>
{% endif %}
```

We now need a custom form for registration. Create a `forms.py` file in the `blango_auth` directory (not the one in the `templates` directory). We saw earlier how to create the `BlangoRegistrationForm` and set the model to our custom User model. However we want to also use the Django Crispy Form helper to add a submit button to the form, with the text *Register*.

Putting that all together and the contents of `forms.py` is like this:

Open blango_auth/forms.py

```
from crispy_forms.helper import FormHelper
from crispy_forms.layout import Submit
from django_registration.forms import RegistrationForm

from blango_auth.models import User


class BlangoRegistrationForm(RegistrationForm):
    class Meta(RegistrationForm.Meta):
        model = User

    def __init__(self, *args, **kwargs):
        super(BlangoRegistrationForm, self).__init__(*args,
        **kwargs)
        self.helper = FormHelper()
        self.helper.add_input(Submit("submit", "Register"))
```

Finally, let's configure `urls.py`. Start by adding these imports:

Open urls.py

```
from django_registration.backends.activation.views import
        RegistrationView
from blango_auth.forms import BlangoRegistrationForm
```

Then add a URL mapping to the `RegistrationView`, so that we can specify `BlangoRegistrationForm` as the form class. Inside `urlpatterns`:

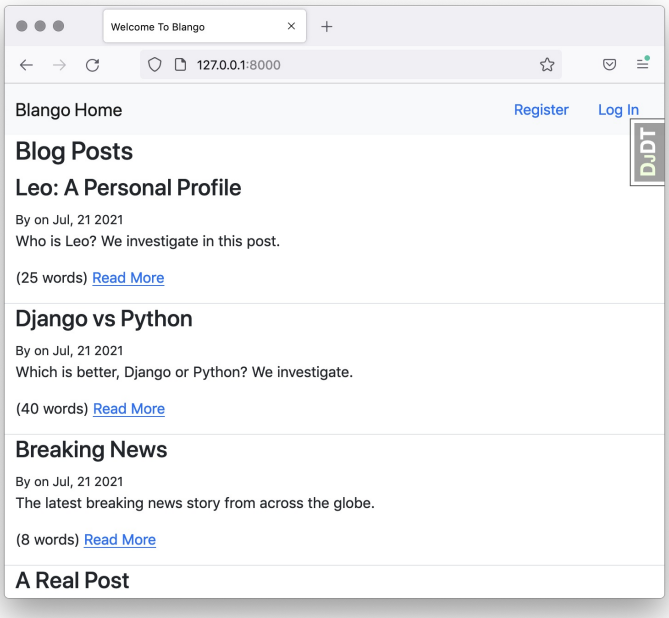
```
path(
    "accounts/register/",
    RegistrationView.as_view(form_class=BlangoRegistrationForm),

    name="django_registration_register",
),
```

Then, add a mapping to include the two step activation URLs. Also inside `urlpatterns`:

```
path("accounts/",
        include("django_registration.backends.activation.urls"))
        ,
```

That's it, start the Django dev server if it's not already running, and try it out. Visit the main page and you'll see the *Register* link in the top-right corner.

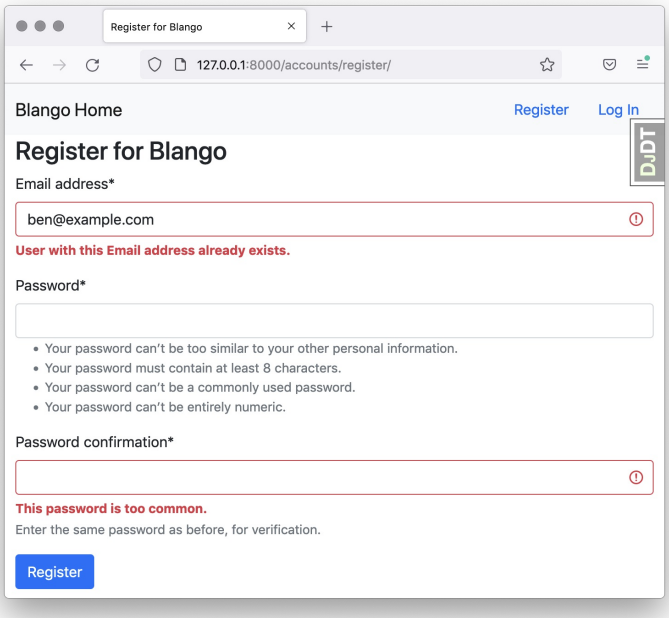
View Blog



register link

Click it to go to the registration page.

Try registering with some bad data to see that Django Registration provides validation for us:



registration validation

Then try with some good data, and you should see the registration successful page. If you look in your console/Terminal now, you should see that the email has “arrived”.

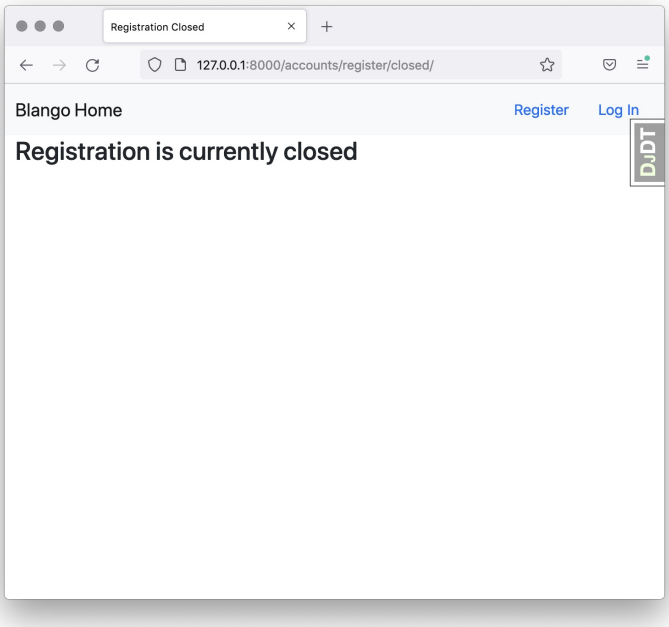
```
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: Activate your Blango account! You have 7 days!
From: webmaster@localhost
To: buzz@example.com
Date: Tue, 27 Jul 2021 09:47:01 -0000
Message-ID: <162737922168.98234.15162000611058500426@BensMBP.local>

Hi,

You registered for Blango, but you need to activate your account within 7 days.

To do that, please visit this page:

http://127.0.0.1:8000/accounts/activate/ImJ1enpAZXhhbXBsZS5jb20i:1m8Jfx:thOXnxkZJ3vreFXJBlXa72qP7_-l48AopxmpdH97jZs/

Thanks,
The Blango Team
```

email in console

Visit that link and you should get the success page. If you visit it again, you’ll get a message that the account has already been activated. You can try also putting in a random key in the URL and seeing the other message you get.

Try logging in as the user that you registered as, you should be able to, but you won’t be able to access the Django admin (this is to be expected).

Finally, try adding the setting `REGISTRATION_OPEN = False` to `settings.py`. Then log out and visit the registration page. You should see the *Registration is currently closed* message.



registration closed

Remove the `REGISTRATION_OPEN` setting after you’ve tested this out (it defaults to `True`).

As you can see, the two-step activation workflow is a big of a lengthy process, to set up all the templates, but it's fairly simple, and a good way to add an extra layer of protection to your user registration process. But if people sign up, and never validate their email address, what happens?

Cleaning Up Inactive Users

Django Registration doesn't have support for automatically cleaning up users that have registered but not validated, but it's fairly easy to do. You can query for the users that joined more than `ACCOUNT_ACTIVATION_DAYS` days ago, but are still not active, then delete them. For example:

```
from datetime import timedelta

from django.conf import settings
from django.utils import timezone

from blango_auth.models import User
User.objects.filter(
    is_active=False,
    date_joined__lt=timezone.now() -
        timedelta(days=settings.ACCOUNT_ACTIVATION_DAYS)
).delete()
```

This could be turned into a management command and then set up to be executed periodically.

In the next section we'll look at how to integrate authentication with social accounts using Django Allauth.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish django registration"
```

- Push to GitHub:

```
git push
```