Don Bosco Institute of Technology

Department of Information Technology

STTP on Open source Technologies for Engineering Education

Date: 4/1/2018

Django Workshop

1. Creating Simple App

To Create simple Pages using models, view and templaes methods.

1. Initial Setup

- create and enter a new virtual environment

```
vaishali@vaishali:~/Hello$ virtualenv -p python3 env
vaishali@vaishali:~/Hello$ source env/bin/activate
```

- install Django

```
(env) vaishali@vaishali:~/Hello$ pip install django
```

- create a new Django project

```
(env) vaishali@vaishali:~/Hello$ django-admin.py startproject
Hello_project
(env) vaishali@vaishali:~/Hello$ cd Simple_project
```

- create a new app

```
(env) vaishali@vaishali:~/Hello/Simple_project$ ./manage.py
startapp pages
```

- perform a migration to set up the database

```
(env) vaishali@vaishali:~/Hello/Simple_project$ ./manage.py
migrate
(env) vaishali@vaishali:~/Hello/Simple_project$ ./manage.py
runserver
```

- update `settings.py`

```
    # blog_project/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'pages',
]
```

2. Templates

Every web framework needs a convenient way to generate HTML files. In Django, the approach is to use templates so that individual HTML files can be served by a view to a webpage specified by the URL. Let us create a new template folder

```
(env) $ mkdir templates

(env) $ touch templates/home.html

Then we can make a few changes in setting and home.html

# simple_project/settings.py

TEMPLATES = [
    {
        ...
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        ...
    },
]

<!-- templates/home.html -->
<h1>Homepage.</h1>
```

3. Configration of views and urls

```
# pages/views.py
from django.views.generic import TemplateView


class HomePageView(TemplateView):
    template_name = 'home.html'
```

we've capitalized our view now since it's a Python class. The `TemplateView` already contains all the logic needed to display our template, we just need to specify its name.

```
# simple_project/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('pages.urls')),
]
```

Creating page level urls

```
(env) $ touch pages/urls.py
```

```
# pages/urls.py
from django.urls import path

from . import views

urlpatterns = [
    path('', views.HomePageView.as_view(), name='home'),
]
```

If you start up the web server with `./manage.py runserver` and navigate to http://127.0.0.1:8000/ you can see a new homepage.

4. Add an About Page

```
(env) $ touch templates/about.html
```

```
<!-- templates/about.html -->

<h1>About page.</h1>
```

Create a new view for the page.

```
# pages/views.py

from django.views.generic import TemplateView


class HomePageView(TemplateView):
    template_name = 'home.html'


class AboutPageView(TemplateView):
    template_name = 'about.html'
```

And then connect it to a url at about/.

```
# pages/urls.py

from django.urls import path

from . import views
```

```
urlpatterns = [
    path('', views.HomePageView.as_view(), name='home'),
    path('about/', views.AboutPageView.as_view(), name='about'),
]
```

Start up the web server with ./manage.py runserver, navigate to http://127.0.0.1:8000/about, and you can see our new “About page”.

5. Extending Templates

The real power in templates is their ability to be extended. In most of the web sites, there is content that is repeated on every page (header, footer, etc). Wouldn’t it be nice if we, could have *one canonical place* for our header code that would be inherited by all other templates.

```
(env) $ touch templates/base.html
```

```
<!-- pages/base.html -->

<header>
  <a href="{% url 'home' %}">Home</a> | <a href="{% url 'about'
%}">About</a>
</header>

{% block content %}
{% endblock %}
```

At the bottom we’ve added a block tag called content. Blocks can be overwritten by child templates via inheritance.

Let’s update our home.html and about.html to extend the base.html template.

```
<!-- templates/home.html -->

{% extends 'base.html' %}

{% block content %}
<h1>Homepage.</h1>
{% endblock %}

<!-- templates/about.html -->

{% extends 'base.html' %}

{% block content %}
<h1>About page.</h1>
{% endblock %}
```

Now if we start up the server with ./manage.py runserver and open up our webpages again at http://127.0.0.1:8000/ and http://127.0.0.1:8000/about 'll see the header is magically included in *both* locations.

2. Createing Message Board app

1. Initial Setup

2. Create a database model

Our first task is to create a database model where we can store and display posts from our users. Django will turn this model into a database table for us.

```
# posts/models.py

from django.db import models


class Post(models.Model):
    text = models.TextField()
```

3. Activating models

The next step is to build the actual database with the migrate command based on the migrations file.

```
(env) $ ./manage.py makemigrations posts
(env) $ ./manage.py migrate posts
```

4. Django Admin

Django provides us with a robust admin interface for interacting with our database.

```
(env) $ ./manage.py createsuperuser

Username (leave blank to use 'wsv'): wsv
Email:
Password:
Password (again):
Superuser created successfully.
```

Restart the Django server with ./manage.py runserver and in your browser go to http://127.0.0.1:8000/admin/. You should see the admin's login screen:

Login by entering the username and password you just created. You should see the Django admin homepage next:

We need to explicitly tell Django what to display in the admin. Fortunately we can change fix this easily by opening the posts/admin.py

```
# posts/admin.py

from django.contrib import admin

from .models import Post

admin.site.register(Post)
```

Now refresh your browser and see the changes

Now let’s create our first message board post for our database. Click on the + Add button opposite Post. Enter your own text in the Text form field.

Within the posts/models.py file, add a new function __str__ as follows:

```
# posts/models.py

from django.db import models


class Post(models.Model):
    text = models.TextField()

    def __str__(self):
        """A string representation of the model."""
        return self.text[:50]
```

If you refresh your Admin page in the browser, you’ll see it’s changed to a much more descriptive and helpful representation of our database entry.

5. Views/Templates/URLs

In order to display our database content on our homepage, we have to wire up our views, templates, and URLConfs.

```
# posts/views.py

from django.views.generic import ListView
from .models import Post


class HomePageView(ListView):
    model = Post
    template_name = 'home.html'
```

Our view is complete which means we still need to configure our URLs and make our template.

```
(env) $ mkdir templates

(env) $ touch templates/home.html
```

Then update the DIRS field in our settings.py file so that Django knows to look in this templates folder.

```
# settings.py

TEMPLATES = [
    {
        ...
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        ...
    },
]
```

```
<!-- templates/home.html -->

<h1>Message board homepage</h1>
<ul>
  {% for post in object_list %}
    <li>{{ post }}</li>
  {% endfor %}
</ul>
```

The last step is to set up our URLConfs. Let’s start with the project-level urls.py file where we simply include our posts and add include on the second line.

```python
# mb_project/urls.py

from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('posts.urls')),
]
```

Then create an app-level urls.py file.

```
(env) $ touch posts/urls.py
```

And update it like so:

```python
# posts/urls.py
from django.urls import path

from . import views

urlpatterns = [
    path('', views.HomePageView.as_view(), name='home'),
]
```

Restart the server with ./manage.py runserver and navigate to our homepage http://127.0.0.1:8000/ which now lists out our message board posts.

3. Creating Blog App

To create a basic blog application that displays a list of all blog posts on the homepage, and a dedicated page for each blog post,

1. Initial Setup

- create and enter a new virtual environment

```
vaishali@vaishali:~/Blog$ virtualenv -p python3 env
vaishali@vaishali:~/Blog$ source env/bin/activate
```

- install Django

```
(env) vaishali@vaishali:~/Blog$ pip install django
```

- create a new Django project

```
(env) vaishali@vaishali:~/Blog$ django-admin.py
startproject blog_project

(env) vaishali@vaishali:~/Blog$ cd blog_project
```

- create a new app

```
(env) vaishali@vaishali:~/Blog/blog_project$ ./manage.py
startapp blogs
```

- perform a migration to set up the database

```
(env) vaishali@vaishali:~/Blog/blog_project$ ./manage.py
migrate
(env) vaishali@vaishali:~/Blog/blog_project$ ./manage.py
runserver
```

- update settings.py

- # blog_project/settings.py

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blogs',
]
```

2. Database Models

Here each post has a title, text, and an author. We can turn this into a database model by opening the blogs/models.py file

```
# blogs/models.py
from django.db import models


class Post(models.Model):
    author = models.ForeignKey(
        'auth.User',
        on_delete=models.CASCADE,
    )
    title = models.CharField(max_length=200)
    text = models.TextField()

    def __str__(self):
        return self.title
```

At the top we’re importing the class models and then creating a subclass of models.Model called Post. Using this subclass functionality we automatically have access to everything within django.db.models.Models and can add additional fields and methods as desired.

```
(env) $ ./manage.py make migrations blogs
(env) $ ./manage.py migrate blogs
```

3. Creating Admin

Creating a superuser

```
(env) $ ./manage.py createsuperuser
Username (leave blank to use 'username'):
Email:
Password:
Password (again):
Superuser created successfully.
```

Now start running the Django server again with the command ./manage.py runserver and open up the Django admin at http://127.0.0.1:8000/admin/. Login with your new superuser account.

update blogs/admin.py to see the posts

```
# blogs/admin.py
from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

If you refresh the page you’ll see the update.
Let’s add two blog posts so we have some sample data to work with.
Click on the + Add button next to Posts to create a new entry.

Now that our database model is complete we need to create the necessary views, URLs, and templates so we can display the information on our web application.

4. Creating URLs

We want to display our blog posts on the homepage so, we’ll first configure our project-level URLConfs and then our app-level URLConfs to achieve this.

On the command line quit the existing server with `Control-c` and create a new `urls.py` file within our `blogs`:

```
(env) $ touch blogs/urls.py
```

```
# blogs/urls.py
from django.urls import path

from . import views

urlpatterns = [
    path('', views.BlogListView.as_view(), name='home'),
]
```

We also should update our project-level urls.py file so that it knows to forward all requests directly to the blog app.

```
# blog_project/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('blogs.urls')),
]
```

5. creating Views

We’re going to use class-based views . In our views file, add the code below to display the contents of our Post model using ListView.

# blogs/views.py
```
from django.views.generic import ListView

from . models import Post


class BlogListView(ListView):
    model = Post
    template_name = 'home.html'
```

On the top two lines we import ListView and our database model Post. Then we subclass ListView and add links to our model and template. This saves us a lot of code versus implementing it all from scratch.

6. Templates

With our URLConfs and Views now complete, now we are creating templates! we can inherit from other templates to keep our code clean. Thus we’ll start off with a `base.html` file and a `home.html` file that inherits from it. Then later when we add templates for creating and editing blog posts, they too can inherit from `base.html`.

Start by creating our project-level `templates` directory with the two template files.

```
(env) $ mkdir templates
(env) $ touch templates/base.html
(env) $ touch templates/home.html
```

Then update `settings.py` so Django knows to look there for our templates.

# blog_project/settings.py
```
TEMPLATES = [
    {
        ...
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        ...
    },
]
```

Update the base.html template as follows.

```
<!-- templates/base.html -->
<html>
  <head>
    <title>Django blog</title>
  </head>
  <body>
    <header>
      <h1><a href="/">Django blog</a></h1>
    </header>
    <div class="container">
      {% block content %}
      {% endblock content %}
    </div>
  </body>
</html>

<!-- templates/home.html -->
{% extends 'base.html' %}

{% block content %}
  {% for post in object_list %}
    <div class="post-entry">
      <h2><a href="">{{ post.title }}</a></h2>
      <p>{{ post.text }}</p>
    </div>
  {% endfor %}
{% endblock content %}
```

At the top we note that this template extends base.html and then wrap our desired code with content blocks. Then we use the Django Templating Language to set up a simple *for loop* for each blog post. Note that object_list comes from ListView and contains all the objects in our view.

If you start the Django server again ./manage.py runserver and refresh http://127.0.0.1:8000/ we can see it’s working.

3. Blog app with forms

1.Creating form

Forms are very complicated. Any time you are accepting user input there are security concerns (XSS Attacks), proper error handling is required, and there are UI considerations around how to alert the user to problems with the form as well as redirects on success.

Fortunately for us Django Forms provide a rich set of tools to handle common use cases working with forms.

update our base template to display a link to a page for entering new blog posts

```
<!-- templates/base.html -->
{% load staticfiles %}
<html>
  <head>
    <title>Django blog</title>
    <link href="https://fonts.googleapis.com/css?
family=Source+Sans+Pro:400" rel="stylesheet">
    <link rel="stylesheet" href="{% static 'css/base.css' %}">
  </head>
  <body>
    <div class="container">
      <header>
        <div class="nav-left">
          <h1><a href="/">Django blog</a></h1>
        </div>
        <div class="nav-right">
          <a href="{% url 'post_new' %}">+ New Blog Post</a>
        </div>
      </header>
      {% block content %}
      {% endblock content %}
    </div>
  </body>
</html>
```

Let’s add a new URLConf for post_new now:

```
 # blog_app/urls.py
from django.urls import path

from . import views

urlpatterns = [
    path('', views.BlogListView.as_view(), name='home'),
    path('post/<int:pk>/', views.BlogDetailView.as_view(),
name='post_detail'),
    path('post/new/', views.BlogCreateView.as_view(),
name='post_new'),
]
```

Our url will start with post/new/, the view is called BlogCreateView, and the url will be named post_new.

Now let’s create our view by importing a new generic class called `CreateView` and then subclass it to create a new view called `BlogCreateView`.

```
# blog_app/views.py
from django.views.generic import ListView, DetailView
from django.views.generic.edit import CreateView
from . models import Post


class BlogListView(ListView):
    model = Post
    template_name = 'home.html'


class BlogDetailView(DetailView):
    model = Post
    template_name = 'post_detail.html'


class BlogCreateView(CreateView):
    model = Post
    template_name = 'post_new.html'
    fields = '__all__'
```

Within `BlogCreateView` we specify our database model `Post`, the name of our template `post_new.html`, and all fields with `'__all__'` since we only have two: `title` and `author`.

The last step is to create our template, which we will call `post_new.html`.

```
(env) $ touch templates/post_new.html
```

```
<!-- templates/post_new.html -->
{% extends 'base.html' %}

{% block content %}
    <h1>New post</h1>
    <form action="" method="post">{% csrf_token %}
      {{ form.as_p }}
      <input type="submit" value="Save" />
    </form>
{% endblock %}
```

Let's breakdown what we've done:

- On the top line we inherit from our base template.
- Use HTML `<form>` tags with the method POST since we're *sending* data. If we were receiving data from a form, for example in a search box, we would use GET.
- Add a {% csrf_token %} which Django provides to protect our form from cross-site scripting attacks. **You should use it for all your Django forms.**
- Then to output our form data we use `{{ form.as_p }}` which renders it within paragraph `<p>` tags.
- Finally specify an input type of submit and assign it the value "Save"

To view our work, start the server with `./manage.py runserver` and go to the homepage at http://127.0.0.1:8000/.

```
Let's just send a user back to the homepage after success since
that will show the list of all their blogs.
```

We can follow Django's suggestion and add an `get_absolute_url` to our model. Open the `models.py` file. Add a new import on the second line for reverse and a new method `get_absolute_url`.

```
# blog_app/models.py
from django.db import models
from django.urls import reverse


class Post(models.Model):
    author = models.ForeignKey(
        'auth.User',
        on_delete=models.CASCADE,
    )
    title = models.CharField(max_length=200)
    text = models.TextField()

    def get_absolute_url(self):
        return reverse('home')

    def __str__(self):
        return self.title
```

2. Update Form

The process for creating an update form so users can edit blog posts should feel familiar. We'll again use a built-in Django class-based generic view, UpdateView, and create the requisite template, url, and view.

To start, let’s add a new link to `post_detail.html` so that the option to edit a blog post appears on an individual blog page.

```
<!-- templates/post_detail.html -->
{% extends 'base.html' %}

{% block content %}
  <div class="post-entry">
    <h2>{{ object.title }}</h2>
    <p>{{ object.text }}</p>
  </div>

  <a href="{% url 'post_edit' post.pk %}">+ Edit Blog Post</a>
{% endblock content %}

<!-- templates/post_edit.html -->
{% extends 'base.html' %}

{% block content %}
    <h1>Edit post</h1>
    <form action="" method="post">{% csrf_token %}
      {{ form.as_p }}
    <input type="submit" value="Update" />
</form>
{% endblock %}
```

We again use HTML <form></form> tags, Django’s csrf_token for security, form.as_p to display our form fields with paragraph tags, and finally give it the value “Update” on the submit button.

Now to our view. We need to import `UpdateView` on the second-from-the-top line and then subclass it in our new view `BlogUpdateView`.

```
# blog_app/views.py
from django.views.generic import ListView, DetailView
from django.views.generic.edit import CreateView, UpdateView
from . models import Post


class BlogListView(ListView):
    model = Post
    template_name = 'home.html'


class BlogDetailView(DetailView):
    model = Post
    template_name = 'post_detail.html'
```

```
class BlogCreateView(CreateView):
    model = Post
    template_name = 'post_new.html'
    fields = '__all__'


class BlogUpdateView(UpdateView):
    model = Post
    fields = ['title', 'text']
    template_name = 'post_edit.html'
```

Notice that in BlogUpdateView we are explicitly listing the fields we want to use ['title', 'text'] rather than using '__all__'. This is because we assume that the author of the post is not changing; we only want the title and text to be editable.

The final step is to update our `urls.py` file as follows:

```
# blog_app/urls.py
from django.urls import path

from . import views

urlpatterns = [
    path('', views.BlogListView.as_view(), name='home'),
    path('post/<int:pk>/', views.BlogDetailView.as_view(),
name='post_detail'),
    path('post/new/', views.BlogCreateView.as_view(),
name='post_new'),
    path('post/<int:pk>/edit/',
        views.BlogUpdateView.as_view(), name='post_edit'),
]
```

At the top we add our view BlogUpdateView to the list of imported views, then created a new url pattern for /post/pk/edit and given it the name post_edit.
Now run your server and go to http://127.0.0.1:8000

4. Creating login Logout Page

Whenever we create a new project, by default Django installs the `auth` app, which provides us with a User object containing:

- username
- password
- email
- first_name
- last_name

We will use this **`User` object** to implement login, logout, and signup in our blog application.

1. Login

Django provides us with a default view for a login page via LoginView.

1. All we need to add are a project-level urlpattern for the auth system,

```
# blog_project/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/', include('django.contrib.auth.urls')),
    path('', include('blogs.urls')),
]
```

2. A login template,

by default Django will look within a templates folder called `registration` for a file called `login.html` for a login form.

```
(env) $ mkdir templates/registration
(env) $ touch templates/registration/login.html

<!-- templates/registration/login.html -->
{% extends 'base.html' %}

{% block content %}
<h2>Login</h2>
<form method="post">
  {% csrf_token %}
  {{ form.as_p }}
  <button type="submit">Login</button>
</form>
{% endblock content %}
```

3. a small update to our `settings.py` file.

```
# settings.py
LOGIN_REDIRECT_URL = '/'
```

now start up the Django server again with `./manage.py runserver` and navigate to our login page at http://127.0.0.1:8000/accounts/login/

4. Updating Homepage for login/logout Status

1. To disply the login status is_authenticated attribute can be used.Update the `base.html` file with new code starting beneath the closing `</header>` tag.

```
<!-- templates/base.html -->
...
</header>
{% if user.is_authenticated %}
  <p>Hi {{ user.username }}!</p>
  <p><a href="{% url 'logout' %}">logout</a></p>
{% else %}
  <p>You are not logged in.</p>
  <a href="{% url 'login' %}">login</a>
{% endif %}
```

`No need to create` *`view`* `as itis provided to us by the Django auth app.We do need to specify where to redirect a user upon logout though.`

`2. a small update to our settings.py file.`

```
# blog_project/settings.py
LOGIN_REDIRECT_URL = '/'
LOGOUT_REDIRECT_URL = '/'
```

Refresh the page to check login/logout status.

5. Creating Signup Page

Django provides us with a form class, UserCreationForm, to make things easier. By default it comes with three fields: `username`, `password1`, and `password2`.

1. Create an app accounts

```
(blog) $ ./manage.py startapp accounts
```

2. Add the new app to the INSTALLED_APPS setting in our settings.py file.

3. Add a new app to the project-level url

```
# blog_project/urls.py
.
.
urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/', include('django.contrib.auth.urls')),
    path('accounts/', include('accounts.urls')),
    path('', include('blogs.urls')),
]
```

4. create our accounts/urls.py file.

```
(blog) $ touch accounts/urls.py
```

```
# acounts/urls.py
from django.urls import path

from . import views

urlpatterns = [
    path('signup/', views.SignUpView.as_view(), name='signup'),
]
```

5. Create a View.The view which uses the built-in UserCreationForm and generic CreateView.

```
 # accounts/views.py
from django.contrib.auth.forms import UserCreationForm
from django.urls import reverse_lazy
from django.views import generic


class SignUpView(generic.CreateView):
    form_class = UserCreationForm
    success_url = reverse_lazy('login')
    template_name = 'signup.html'
```

6. Create signup.html to project-level templates folder

```
<!-- templates/signup.html -->
{% extends 'base.html' %}

{% block content %}
<h2>Sign up</h2>
<form method="post">
  {% csrf_token %}
  {{ form.as_p }}
  <button type="submit">Sign up</button>
</form>
{% endblock %}
```

Its all about to create Signup Page!

5.Social Authentication with django-allauth

Django comes with a robust built-in authentication system for users but it does not provide support for third-party (social) authentication via services like Github, Gmail, or Facebook. Fortunately the excellent 3rd party django-allauth package does in just a few steps.

1. Creating a New Setup:

- vaishali@vaishali:~/Social$ virtualenv -p python3 env
- vaishali@vaishali:~/Social$ source env/bin/activate
- (env) vaishali@vaishali:~/Social$ pip install django
- (env) vaishali@vaishali:~/Social$ django-admin.py startproject socialauth
- (env) vaishali@vaishali:~/Social/socialauth$ ./manage.py migrate
- (env) vaishali@vaishali:~/Social/socialauth$ ./manage.py runserver

2.django-allauth

Now we can install `django-allauth` and configure our project.

- (env) vaishali@vaishali:~/Social/socialauth$ pip install django-allauth
- We need to update our settings.py file.

```
# socialauth/settings.py
INSTALLED_APPS = [
  ...
  'django.contrib.sites',
  'allauth',
  'allauth.account',
  'allauth.socialaccount',
  'allauth.socialaccount.providers.github',
]
```

Then at the bottom of settings.py we need to specify that we’re using the allauth backend, add a SITE_ID since allauth uses the built-in sites app, and configure a redirect to the homepage upon successful login.

# socialauth/settings.py
```
AUTHENTICATION_BACKENDS = (
    "django.contrib.auth.backends.ModelBackend",
    "allauth.account.auth_backends.AuthenticationBackend",
)

SITE_ID = 1

LOGIN_REDIRECT_URL = '/'
```

- The django-allauth package is installed so now we need to add it to our urls.

```
# socialauth/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/', include('allauth.urls')),
]
```

- **migrate our changes to update the existing database.**

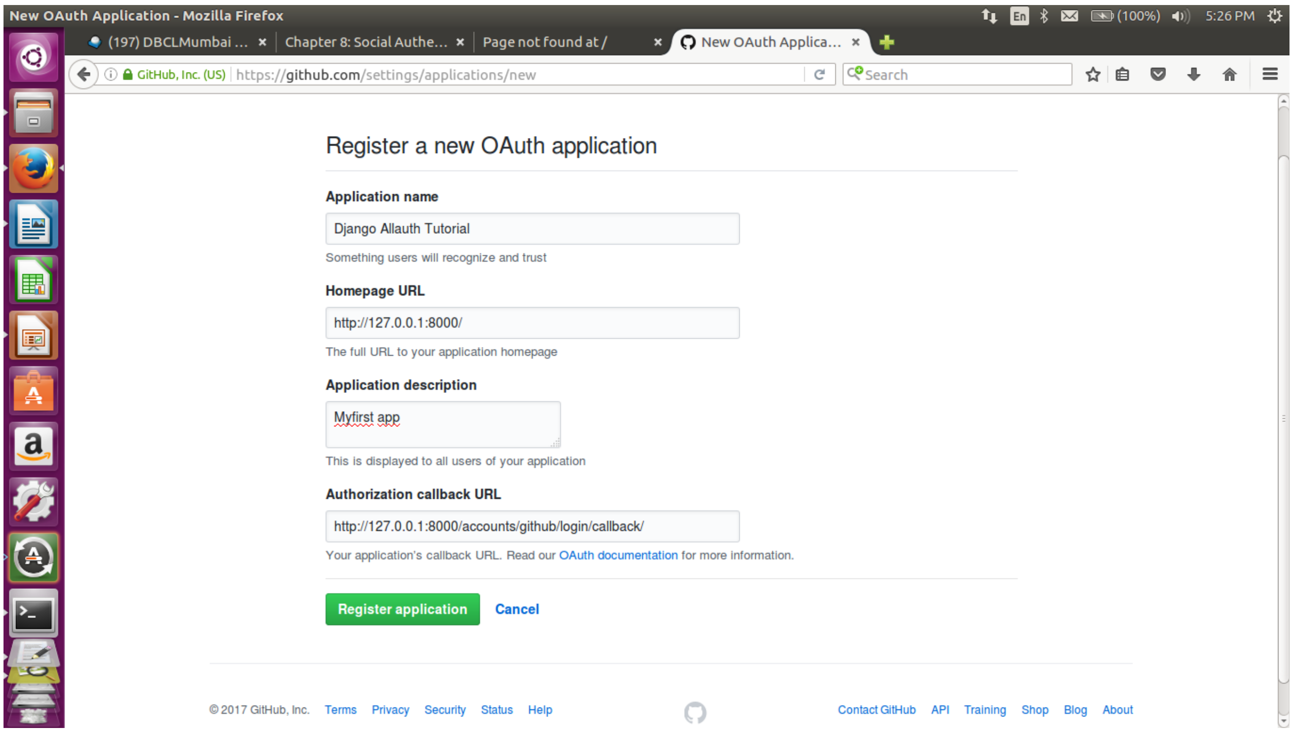
```
(env) vaishali@vaishali:~/Social/socialauth$
./manage.py migrate
```

3. Github OAuth

OAuth is an open standard for authentication between systems. When a user logs into our site with their Github account, we will redirect them to Github which then sends us a token that represents the user.

To configure a new OAuth application on Github, go to https://github.com/settings/applications/new.

It’s important to fill out the *Application Name*, *Homepage URL*, and *Authorization callback URL*. The *Application description* is optional.


Register a new OAuth application

Application name: Django Allauth Tutorial
Something users will recognize and trust

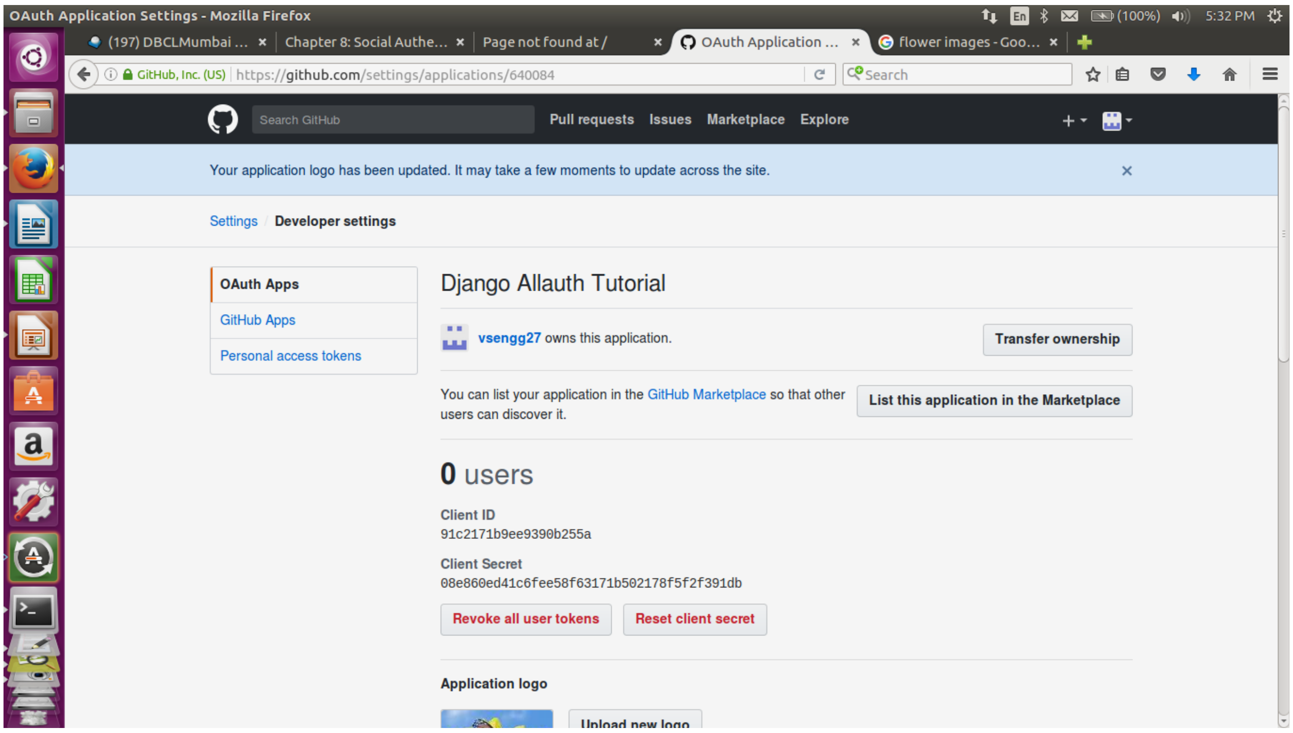
Homepage URL: http://127.0.0.1:8000/
The full URL to your application homepage

Application description: Myfirst app
This is displayed to all users of your application

Authorization callback URL: http://127.0.0.1:8000/accounts/github/login/callback/
Your application's callback URL. Read our OAuth documentation for more information.

The *Application name* is what the user will see is requesting permission to access their Github account. The *Homepage URL* is as described. The *Authorization callback URL* takes a particular form for each integration as defined in the django-allauth docs.

After hitting the “Register application” button you’ll be redirected to the following page.


OAuth Application Settings - Mozilla Firefox

https://github.com/settings/applications/640084

Your application logo has been updated. It may take a few moments to update across the site.

Settings / Developer settings

Django Allauth Tutorial

vsengg27 owns this application.

You can list your application in the GitHub Marketplace so that other users can discover it.

0 users

Client ID
91c2171b9ee9390b255a

Client Secret
08e860ed41c6fee58f63171b502178f5f2f391db

Application logo

Pay particular attention to the **Client ID** and **Client Secret**. We'll be using those shortly. Also note that in the real-world, you'd never want to publicly reveal either of these keys!

4. Django admin

We need to configure the admin portion of our Django project. Create a new superuser so we can login!

```
(env) vaishali@vaishali:~/Social/socialauth$ ./manage.py
createsuperuser
```

- Now we can start the server again with ./manage.py runserver and then navigate to the admin page http://127.0.0.1:8000/admin.
- We need to make two updates to the admin for django-allauth to work correctly. First go to the *Sites* portion and set the domain name to 127.0.0.1.
- Next go back to the admin homepage and click on the *add* button for Social Applications on the bottom. This is where you configure each third-party OAuth.

We’re using Github so that’s the Provider. We add a name and then the **Client ID** and **Secret ID** from Github. Final step is to add our site to the Chosen sites on the bottom. Then click save.

5. Creating Homepage

- Creating a new project-level views.py file on the command line:

  # socialauth/views.py

```
from django.views.generic import TemplateView


class HomePageView(TemplateView):
    template_name = 'home.html'
```

- Now update settings.py file to tell Django to look in a templates folder.

  # socialauth/settings.py

```
TEMPLATES = [
    ...
    'DIRS': [os.path.join(BASE_DIR, 'templates')],
    ...
]
```

- Create this new project-level templates folder and an home.html file within it.

  (env) vaishali@vaishali:~/Social/socialauth$ mkdir templates

  (env) vaishali@vaishali:~/Social/socialauth$ touch templates/home.html

  <!-- templates/home.html -->

```
{% load socialaccount %}

<h1>Django Allauth Tutorial</h1>
{% if user.is_authenticated %}
<p>Welcome {{ user.username }} !!!</p>
{% else %}
<a href="{% provider_login_url 'github' %}">Sign Up</a>
{% endif %}
```

- Finally update our urls.py to point to the new homepage.

```
    # socialauth/urls.py
from django.contrib import admin
from django.urls import path, include

from . import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/', include('allauth.urls')),
    path('', views.HomePageView.as_view(), name='home'),
]
```

- Now turn up the local server with ./manage.py runserver and navigate to the homepage at http://127.0.0.1:8000/admin/ to *logout* from the superuser account.
- Then navigate to the homepage at http://127.0.0.1:8000/ to see the logged out greeting.

6. Sign Up with Github

- Login with Github by clicking on the “Sign Up” link and you’ll be redirected to the Github authorize page.
- Click on the “Authorize” button and you’ll be redirected back to our homepage with a greeting for your Github account name.You can look at all users by navigating back to the admin panel at any time.
- django-allauth comes with a robust list of customizations we can add, including a logout link, requiring email confirmation, and much more.