Django Class Notes

Clarusway

Django Authentication System - 1 (Built-In)

Nice to have VSCode Extentions:

- Djaneiro - Django Snippets (Be carefull about other conflicting extentions!)

Needs

- Python, add the path environment variable
- pip
- virtualenv

Summary

- Create project
- Secure your project
  - .gitignore
  - django-decouple
- Create app
- Login Admin Site
  - Migrate
  - Create superuser
- Add users programmatically
- Adding users with auth
  - Modify project url pattern
  - Create url on app

    - Create View
    - Create Login template
    - Modify view to add new users
- Password Change

Core

There is Django's authentication system in its default configuration.

This configuration has evolved to serve the most common project needs, handling a reasonably wide range of tasks, and has a careful implementation of passwords and permissions.

For projects where authentication needs differ from the default, Django supports extensive extension and customization of authentication.

User objects are the core of the authentication system. They typically represent the people interacting with your site and are used to enable things like restricting access, registering user profiles, associating content with creators etc.

The primary attributes of the default user are:

- username
- password
- email
- first_name
- last_name

The authentication that comes with Django is good enough for most common cases, but you may have needs not met by the out-of-the-box defaults.

Django Authentication System

According to Django the authentication system aims to be very generic, and so does not provide some features provided in other web authentication systems. Solutions for some common problems are available as third-party packages. For example, throttling of login attempts and authentication against third parties (e.g. OAuth).

The necessary configuration was all done for us when we created the app using the django-admin startproject command. The database tables for users and model permissions were created when we first called python manage.py migrate.

```
INSTALLED_APPS = [
    ...
    'django.contrib.auth',  # Core authentication framework and its default
models.
    'django.contrib.contenttypes',  # Django content type system (allows
permissions to be associated with models).
    ....

MIDDLEWARE = [
```

```
    ...
    'django.contrib.sessions.middleware.SessionMiddleware',  # Manages sessions
across requests
    ...
    'django.contrib.auth.middleware.AuthenticationMiddleware',  #  Associates
users with requests using sessions.
    ....
```

Create project

- Create a working directory, name it as you wish, cd to new directory
- Create virtual environment as a best practice:

```
python3 -m venv env # for Windows or
python -m venv env # for Windows
virtualenv env # for Mac/Linux or;
virtualenv yourenv -p python3 # for Mac/Linux
```

- Activate scripts:

```
.\env\Scripts\activate  # for Windows
source env/bin/activate  # for MAC/Linux
```

- See the (env) sign before your command prompt.
- Install django:

```
pip install django
```

- See installed packages:

```
pip freeze

# you will see:
asgiref==3.3.4
Django==3.2.4
pytz==2021.1
sqlparse==0.4.1

# If you see lots of things here, that means there is a problem with your virtual
env activation.
# Activate scripts again
```

- Create requirements.txt same level with working directory, send your installed packages to this file, requirements file must be up to date:

```
pip freeze > requirements.txt
```

- Create project:

```
django-admin startproject authenticate
django-admin startproject authenticate .
# With . it creates a single project folder.
# Avoiding nested folders
# Alternative naming:
django-admin startproject main .
```

- Various files has been created!
- Check your project if it's installed correctly:

```
python manage.py runserver
py -m manage.py runserver
```

- (Optional) Change the name of the project main directory as src to distinguish from subfolder with the same name!

```
# optional
mv .\clarusway\ src
```

- Lets create first application:
- Go to the same level with manage.py file:

```
cd .\src\
```

Secure your project

.gitignore

Add standard .gitignore file to the project root directory.

Do that before adding your files to staging area, else you will need extra work to unstage files to be able to ignore them.

python-decouple

- To use python decouple in this project, first install it:

```
pip install python-decouple
```

- For more information about python-decouple
- Import the config object on settings.py file:

```
from decouple import config
```

- Create .env file on root directory. We will collect our variables in this file.

```
SECRET_KEY = o5o9...
```

- Retrieve the configuration parameters in settings.py:

```
SECRET_KEY = config('SECRET_KEY')
```

- Now you can send you project to the github, but be sure you added a .gitignore file which has .env on it.

Create app

- Start app

```
python manage.py startapp user_example

# Alternative naming:
python manage.py startapp home
```

- Go to settings.py and add the app to the INSTALLED_APPS:

```
'user_example'
'user_example.apps.UserExampleConfig'
```

Login Admin Site

Authentication support is bundled as a Django contrib module in django.contrib.auth.

By default, the required configuration is already included in the settings.py generated by django-admin startproject, these consist of two items listed in your INSTALLED_APPS setting:

- 'django.contrib.auth' contains the core of the authentication framework, and its default models.
- 'django.contrib.contenttypes' is the Django content type system, which allows permissions to be associated with models you create.

and these items in your MIDDLEWARE setting:

- SessionMiddleware manages sessions across requests.
- AuthenticationMiddleware associates users with requests using sessions.

With these settings in place, running the command manage.py migrate creates the necessary database tables for auth related models and permissions for any models defined in your installed apps.

- We need to create users. Go to manage.py level directory:

```
python manage.py migrate
```

- In order to login, we need to create a user who can login to the admin site. Run the following command:

```
python manage.py createsuperuser  # or with the parameters
python manage.py createsuperuser --username rafe --email admin@mail.com
```

- Enter your desired username, email adress, and password twice.
- Run server.
- Go to http://127.0.0.1:8000/admin/ You should see the admin's login screen.
- After you login, you should see a few types of editable content: groups and users. They are provided by django.contrib.auth, the authentication framework shipped by Django.

You already created your first user as a superuser.

Our superuser is already authenticated and has all permissions, so we'll need to create a test user to represent a normal site user.

We can use the admin site to create groups and users, as it is one of the quickest ways to do so.

First lets create a new group for our website members. We don't need any permissions for the group, so just press SAVE.

Create a regular user.

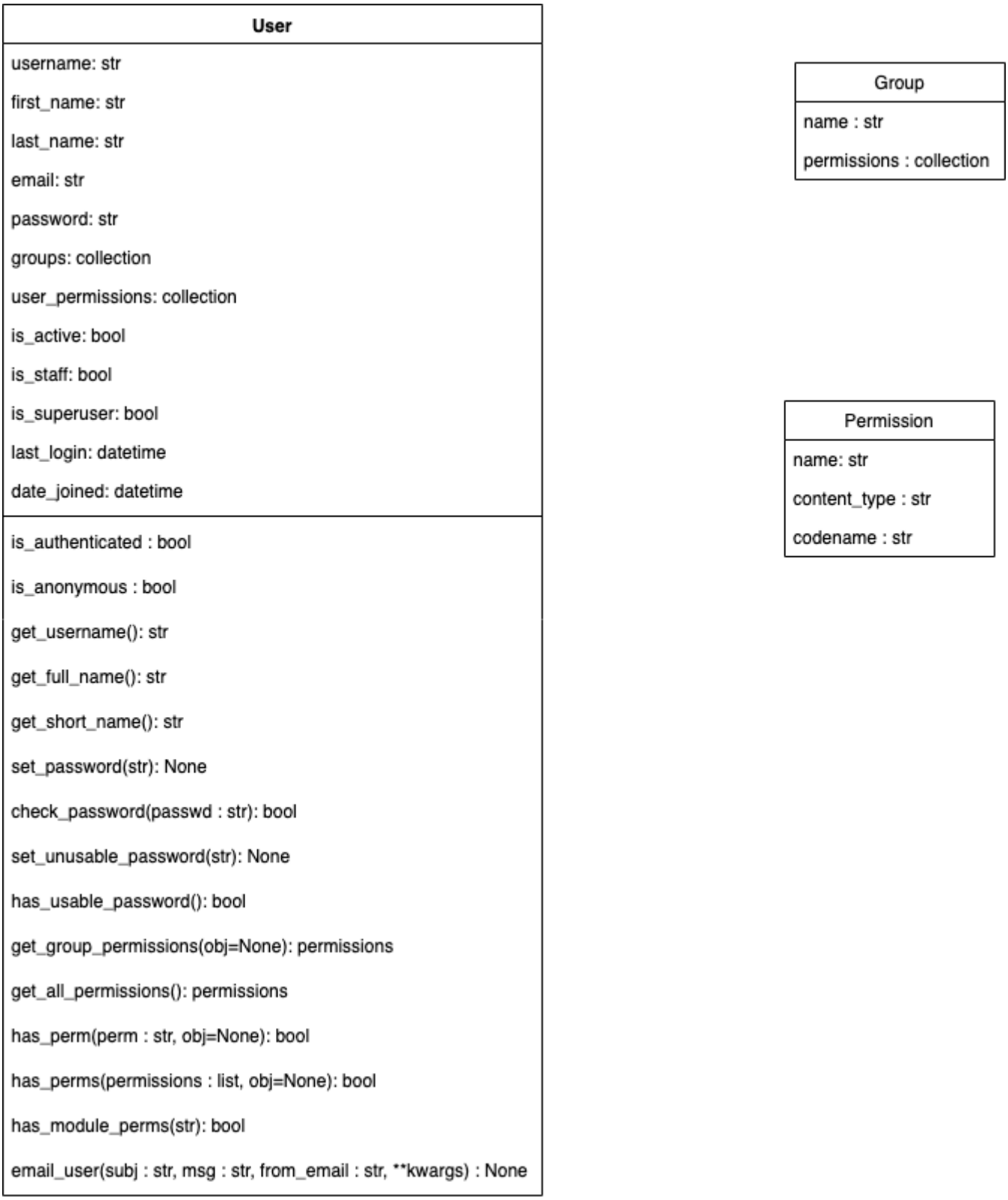
The Django Authentication Models

Django.contrib.auth.models has

- User,

- Permission,
- Group Models, which serves to associate a user with some persisted data about that user along with any groups and permissions they have.

Below you can see the class diagrams for User as well as Permission and Group.



User Model

Add users programmatically

https://docs.djangoproject.com/en/4.0/topics/auth/default/

The most direct way to create users is to use the included create_user() helper function:

```
python manage.py shell

from django.contrib.auth.models import User

# Create user and save to the database
# create_user(username, email=None, password=None, **extra_fields)
# Creates, saves and returns a User.
user1 = User.objects.create_user('myusername', 'myemail@crazymail.com',
'mypassword')
# The username and password are set as given.
# The domain portion of email is automatically converted to lowercase, and the
returned User object will have is_active set to True.
# If no password is provided, set_unusable_password() will be called.
# The extra_fields keyword arguments are passed through to the User's __init__
method to allow setting arbitrary fields on a custom user model.
# Another ecample with extra fields:
user2 = User.objects.create_user('john', email='lennon@thebeatles.com',
password='johnpassword', is_staff=True)
# Or
user2.is_staff=True
user2.save()

# Update fields and then save again
user2.first_name = 'John'
user2.last_name = 'Citizen'
user2.save()
```

Django does not store raw (clear text) passwords on the user model, but only a hash (see documentation of how passwords are managed for full details). Because of this, do not attempt to manipulate the password attribute of the user directly. This is why a helper function is used when creating a user.

```
python manage.py changepassword <username>

# You can also change a password programmatically, using set_password():
from django.contrib.auth.models import User

user3 = User.objects.get(username='john')
user3.set_password('newpassword')
user3.save()
```

If you have the Django admin installed, you can also change user's passwords on the authentication system's admin pages.

Django also provides views and forms that may be used to allow users to change their own passwords.

Changing a user's password will log out all their sessions.

Add users with auth

We want to allow adding regular users to our app.

- Go to authenticate/urls.py and add:

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

urlpatterns = [
    path('admin/', admin.site.urls),
    ### And using some urls which Django give us about authentication:
    path('accounts/', include('django.contrib.auth.urls'))
]
```

If you look at django.contrib.auth.urls you can see the default views that are defined. That would be login, logout, password_change and password_reset.

Default URLconf

- Navigate to the http://127.0.0.1:8000/accounts/ URL (note the trailing forward slash!) and Django will show an error that it could not find this URL, and listing all the URLs it tried. From this you can see the URLs that will work.
- Using the above method adds the following URLs with names in square brackets, which can be used to reverse the URL mappings. You don't have to implement anything else — the above URL mapping automatically maps the below mentioned URLs.

```
accounts/ login/ [name='login']
accounts/ logout/ [name='logout']
accounts/ password_change/ [name='password_change']
accounts/ password_change/done/ [name='password_change_done']
accounts/ password_reset/ [name='password_reset']
accounts/ password_reset/done/ [name='password_reset_done']
accounts/ reset/<uidb64>/<token>/ [name='password_reset_confirm']
accounts/ reset/done/ [name='password_reset_complete']
```

- Try to navigate to the login URL (http://127.0.0.1:8000/accounts/login/). This will fail, but with an error that tells you that we're missing the required template (registration/login.html) on the template search path.
- We will create a registration directory on the search path and then add the login.html file later.
- Add user_example to the project urls.py.

```
path('', include("user_example.urls")),
```

- Create urls.py under user_example, and add:

```
from django.urls import path
from .views import home_view

urlpatterns = [
    path('', home_view, name="home"),
]
```

Create View

- Go to views.py in user_example directory
- Create home view by adding:

```
def home_view(request):
    return render(request, "user_example/home.html")
```

Create an HTML template

- Create user_example/templates/user_example directory and create a home.html file under it:

```
<h1>This is the home page!</h1>
```

- Run our project:

```
python manage.py runserver
```

- Go to http://localhost:8000 in your browser, and you should see the text “This is the home page!”, which you defined in the home view.
- Try to go to the accounts page, which we added url path to the project.
- Try accounts/ and see the options coming from django.contrib.auth.urls
- Try accounts/login. We did not create login page yet. Lets create it.

Login template

- Create templates/registration folder and login.html under this folder. This name is unique Django will look for this name.

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

{% block content %}

<form action="{% url 'login' %}" method="post">

    {% csrf_token %}

    {{ form.as_p }}

    <input type="submit" value="Login">

</form>

{% endblock %}
```

- This will display a form in which you can enter your username and password, and that if you enter invalid values you will be prompted to enter correct values when the page refreshes.

Navigate back to the login page (http://127.0.0.1:8000/accounts/login/) once you've saved your template

- If you log in using valid credentials, you'll be redirected to another page (by default this will be http://127.0.0.1:8000/accounts/profile/). The problem is that, by default, Django expects that upon logging in you will want to be taken to a profile page, which may or may not be the case. As you haven't defined this page yet, you'll get another error!
- Need to change login redirect url to fix this.
- Open the project settings.py and add the text below to the bottom. Now when you log in you should be redirected to the site homepage by default.

```
# Redirect to home URL after login (Default redirects to /accounts/profile/)
LOGIN_REDIRECT_URL = "/"
```

- From now on, when someone login the page, the page will be redirected to home page.

Enable views to add new users

- Add a new view to add registration.
- Go to views.py, create new view:

```
# First import UserCreationForm
from django.contrib.auth.forms import UserCreationForm

def register(request):
    form = UserCreationForm()
    context = {
        'form': form
```

```
    }
    return render(request, "registration/register.html", context)
```

- Before creating register page, lets add it to the url list of our app;

```
from django.urls import path
from .views import home_view, register

urlpatterns = [
    path('', home_view, name="home"),
    path('register', register, name='register')
]
```

- Create register.html under templates/registration folder.

```
<h1>Registration page</h1>

<form action="{% url 'register' %}" method="post">

    {% csrf_token %}

    {% if form.errors %}
        <p>There is something wrong what you entered!</p>
    {% endif %}

    {{ form.as_p }}
    {# as_p orders the scene #}

    <input type="submit" value="Register">

</form>
```

- Try to create new user
- It will send user info to register app, go to the admin page and check it!
- Cant see the new user, so we need to save new user to our users list using view
- If its a get request or post? For get requests, only showing the form is enough. But for post request need to add someting.
- If post and creates a user, need to save it. Use if block:

```
from django.shortcuts import redirect, render
from django.contrib.auth.forms import UserCreationForm

# add authenticate and login
from django.contrib.auth import authenticate, login

def home_view(request):
```

```
    return render(request, "user_example/home.html")

def register(request):

    if request.method == 'POST':
        # pass in post data when instantiate the form.
        form = UserCreationForm(request.POST)
        # if the form is ok with the info filled:
        if form.is_valid():
            form.save()
            # that creates a new user
            # after creation of the user, want to authenticate it
            username = form.cleaned_data['username']
            password = form.cleaned_data['password1']
            # inspect the page and see the first password is password1, import
authenticate

            # Django will verify this password is correct, put the password to the
db
            # hash it. authenticate() function takes care of that.

            # Use authenticate() to verify a set of credentials. It takes
credentials as keyword arguments, username and password for the default case,
checks them against each authentication backend, and returns a User object if the
credentials are valid for a backend.
            user = authenticate(username=username, password=password)

            # want user to login right after registered, import login
            login(request, user)
            # want to redirect to home page, import redirect
            return redirect('home')

    else:
        form = UserCreationForm()

    context = {
        'form': form
    }

    return render(request, "registration/register.html", context)
```

- Lets try to create a new user again
- It must redirect to homepage
- See the user on admin page, there are lots of options to modify user via admin page
- Lets show something to the user on homepage about registration process
- Go to home.html

```
<h1>This is the home page!</h1>


{% if user.is_authenticated %}
```

```
    <h2>Your name is:{{ user.username }}</h2>
{% else %}
    <h2>You are not logged in!</h2>
{% endif %}
```

- Refresh home page and see the result.

Password Change

First look at regular password change and password change done behaviour of Django defaults.

Implementing a built-in password change in Django is very easy. Django provides authentication and authorization. For changing password you need to get authenticated first.

In your urls.py, we need to import PasswordChangeView from Django Auth. By default, Django PasswordChangeView will render a standard template. But we need some customization and we'll tell PasswordChangeView to render a different-named template from registration/change-password.html.

"success_url" is the URL to redirect to after a successful password change. Defaults to 'password_change_done'.

```
from django.contrib.auth import views as auth_views

urlpatterns = [
    path(
        'change-password/',
        auth_views.PasswordChangeView.as_view(template_name='change-
password.html'),
    ),
]
```

The views have optional arguments you can use to alter the behavior of the view. For example, if you want to change the template name a view uses, you can provide the template_name argument. A way to do this is to provide keyword arguments in the URLconf, these will be passed on to the view.

Note that you don't have to write a view for that!

A basic template can be like this:

- Add registration/password_change.html

```
{% extends 'base.html' %}
{% block content %}

<h1>Password Change Page</h1>

<form action="" method="post">

    {% csrf_token %}
```

```
    {{ form.as_p }}
    <br>
    <input type="submit" value="Change Password">

</form>

{% endblock content %}
```

- Add url

```
from django.urls import path
from .views import home_view, register
from django.contrib.auth import views as auth_views

urlpatterns = [
    path('', home_view, name="home"),
    path('register', register, name='register'),
    path('change-password/',
auth_views.PasswordChangeView.as_view(template_name='change-password.html'),
name='change_password'),
]
```

Password Reset

It is easy to add a new password reset template. Add a url first:

```
path('reset-password/',
auth_views.PasswordResetView.as_view(template_name='registration/reset-
password.html'), name='reset-password'),
```

And a template:

```
{% extends 'base.html' %}
{% block content %}

<h1>Password Reset Page</h1>

<form action="" method="post">

    {% csrf_token %}
    {{ form.as_p }}
    <br>
    <input type="submit" value="Reset Password">

</form>

{% endblock content %}
```

We need one more thing, adding a email backend to our settings to see the reply mail of Django.

Adjust a mail backend for development:

We can set up several mail backends, including a SMTP server. For simple use cases let's look at Console and File backends for simplicity.

Console backend:

Instead of sending out real emails the console backend just writes the emails that would be sent to the standard output. By default, the console backend writes to stdout. You can use a different stream-like object by providing the stream keyword argument when constructing the connection.

To specify this backend, put the following in your settings:

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

Important note: make sure your user have a mail in the database.

File backend:

The file backend writes emails to a file. A new file is created for each new session that is opened on this backend.

To specify this backend, put the following in your settings:

```
EMAIL_BACKEND = 'django.core.mail.backends.filebased.EmailBackend'
EMAIL_FILE_PATH = BASE_DIR / "sent_emails"
```

Logout

If you want to redirect to the home page when logout, add this variable to the settings.py:

```
LOGOUT_REDIRECT_URL = '/'
```

The login_required decorator

Look at this link you may use this decorator as a part of auth system.

Login Required Decorator Documentation

Try it by creating a special page! Or simply add it to the home page.

```python
from django.contrib.auth.decorators import login_required

@login_required(login_url='/accounts/login/')
def my_view(request):
    pass
```

Next Steps

Using the built-in Django auth system is easy. But be ready to make some customization.

Search for the documentation in this class note whenever you have questions. Keep yourself up-to-date.

☺ **Happy Coding!** ✍

Clarusway