Unit 4

Classwork: The classwork for this unit should be saved in a new folder: **your_repo/unit_4/**
Homework: Homework should be saved here: **your_repo/homework/unit_4/**

*** Day 01 ***

Authentication/Sessions

We have learned many important features about Django, HTML, JS and Databases since the first day of the first semester.

Now, we will learn more advanced topics in Django and also we will learn a JS framework, Vue.js, for frontend, like Django is a Python framework for backend. With Vue.js, you will be able to implement more advanced JS without coding too much.

Let's start by learning how to authenticate a user and use a session after the authentication process. So far, we did log in into our application. Our application was open to any user who requested our page. Some webpages are entirely public, some are entirely private, and others are both (public and private functionalities in the same web page). So let's see how we can authenticate a user in Django.

First, let's take a look at the table auth_user:

- You have a column id, this will be the id of the user in the database.

- You have a column password. This column is encrypted, that means, you cannot figure out the password just by looking at the string. This is the page that explains how django is using the password: https://docs.djangoproject.com/en/5.0/topics/auth/passwords/ . "By default, Django uses the PBKDF2 algorithm with a SHA256 hash, a password stretching mechanism recommended by NIST (National Institute of Standards and Technology). This should be sufficient for most users: it’s quite secure, requiring massive amounts of computing time to break it."

In the **INSTALLED_APPS** in **settings.py**, you should have 'django.contrib.auth'. It was installed by Django when you did the startproject command. This contains the core of the authentication framework and its default models (users, permission, group….)

In the middleware, you should have 'django.contrib.auth.middleware.AuthenticationMiddleware' and 'django.contrib.sessions.middleware.SessionMiddleware'. The authenticationmiddleware will associate the user into the request using the session. The sessiomiddleware will manage sessions into the request. You will notice that the sessionmiddleware is first in the middleware, to create the session object into the request, and then the authenticationmiddleware is going to use this session. If the sessionmiddleware is not before the authenticationmidlleware it will throw some errors:

See https://github.com/django/django/blob/main/django/contrib/auth/middleware.py where you will find the following:

```
class AuthenticationMiddleware(MiddlewareMixin):
        def process_request(self, request):
                if not hasattr(request, "session"):
                        raise ImproperlyConfigured(
                                "The Django authentication middleware requires session "
                                "middleware to be installed. Edit your MIDDLEWARE setting to "
                                "insert "
                                "'django.contrib.sessions.middleware.SessionMiddleware' before "
                                "'django.contrib.auth.middleware.AuthenticationMiddleware'."
                        )
                request.user = SimpleLazyObject(lambda: get_user(request))
                request.auser = partial(auser, request)
```

First, you need to create a user. Usually, you create a first a super user to have control over all the application:

```
python manage.py createsuperuser
```

This will create a new user, go check the table auth_user. You will see in the password column, your password encrypted.

The next step could be to create a login view to attach the user to the current session. Check here if you want to see the Django documentation for a simple view to have a user logged in: https://docs.djangoproject.com/en/5.0/topics/auth/default/#how-to-log-a-user-in

The example on that web page is a simple view, but we will not use it. We are going to use some default implementation built by Django. But if one day you need to overwrite the login view, for example to implement a 2FA (two factor authentication) and use an OTP (one-time password) authentication app for only certain users, you can use a class view that will override the LoginView (https://github.com/django/django/blob/main/django/contrib/auth/views.py).

```
from django.contrib.auth.views import LoginView
from django.contrib.auth import login as auth_login
```

```
class CoreLoginView(LoginView):
  template_name = "core/login.html"

  def form_valid(self, form):
     """Security check complete. Log the user in."""
     user = form.get_user()
     auth_login(self.request, user)
     if user_need_to_go_to_otp:
         return redirect('otp_url')
    else:
         return else_where
```

For now, we are going to use the default authentication view build by django:
https://docs.djangoproject.com/en/5.0/topics/auth/default/#module-django.contrib.auth.views

In movie_theather/urls.py, add the path for accounts url:

```
urlpatterns = [
  path("admin/", admin.site.urls),
  path("accounts/", include("django.contrib.auth.urls")),
  path("movies/", include("movies.urls", namespace="movies")),
 …
]
```

Based on this documentation https://github.com/django/django/blob/main/django/contrib/auth/, we are going to make localhost:8000/accounts/login work.

What do you see in the view? Any useful information about how to make login work? Templates? Forms?

We can create a login template inside the core templates. I created the template at **core/templates/registrations/login.html**

For the redirection once the login has been successful, in the LoginView, we can see that the class is going to use the variable **settings.LOGIN_REDIRECT_URL**.

LOGIN_REDIRECT_URL should be hardcoded in settings.py (LOGIN_REDIRECT_URL = "/movies/movies/").

Please add the **base** template to set up a **login button**, if the user is not logged in or display it username if logged in.

Let's figure it out. There are some hints here:
https://docs.djangoproject.com/en/5.0/topics/auth/default/

Check the template base.html.
We need also to set up the variable **settings.LOGOUT_REDIRECT_URL.**

Let's create a home page, in the core app. Check the app (core/views.py, template, movie_theater.urls)
And let's set up the LOGOUT_REDIRECT_URL = "/" in settings.py

So now, we can login and logout. You can see in the database, the table django_session.
When you login, you will see a new line in the table, with the session_key the same value as the sessionId Cookie. Let's check it please.

The session_data column is going to be encrypted (using the settings.SECRET_KEY). In this session_data, django will store the value of the user.id if the session is related to a user.

Once you log out, the session in django_session will be deleted.

This is now the only usage of the cookie we will have from now on. We do not need to add cookies, as we have the database now, and we can store information about the session in the session_data.

*** Day 02 ***

Restricting Access

So far, the users have full access to the web pages in our projects. Even if a user is logged out, they can see everything in the app.

In a classic view (function, no class based), you can use a decorator.

What a decorator is in Python?

A decorator will permit you to modify or add a check to the function's result.

Let's see a decorator example:

```
def uppercase_decorator(type):
    def decorator(function):
        def wrapper(*args, **kwargs):
            print(*args)
            func = function(*args, **kwargs)
            if type == "capitalize":
                make_uppercase = func.capitalize()
            else:
                make_uppercase = func.upper()
            return make_uppercase
        return wrapper
    return decorator
```

Let's apply the decorator:

```
@uppercase_decorator("capitalize")
def say_hi():
    return 'hello there'

say_hi()
```

If you need to set up arguments in the decorator function, you can do the following:

```
def uppercase_decorator(function):
    def wrapper():
        func = function()
        make_uppercase = func.upper()
        return make_uppercase

    return wrapper
```

To apply the decorator function directly, you can do simply like this:

```
def say_hi():
    return 'hello there'

decorate = uppercase_decorator(say_hi)
decorate()
```

And using the @:

```
@uppercase_decorator
def say_hi():
    return 'hello there'

say_hi()
```

You can look a this page for a detailed explanation:
https://book.pythontips.com/en/latest/decorators.html

Exercises (Homework):

1. Write a python decorator that will measure the execution time of a function.(required)

2. Write a python decorator that will check the calls over an api over a period of time and throw an error if the number of calls exceeds the limit. (optional)

This is the decorator with arguments and the api call:

```
@rate_limits(max_calls=6, period=10)
def api_call():
   print("API call executed successfully...")
```

Django Login Decorator:

Django has a decorator that checks if a user is logged in.

Let's take a look at it on the django code:
https://github.com/django/django/blob/main/django/contrib/auth/decorators.py

We see in the function login_required (user_passes_test), if the user is not authenticated, the request is going to redirect to settings.LOGIN_URL or to login_url if it was passed as an argument on the decorator function login_required.

To try we are going to add the following line in core/views.py:

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

Let's create a view function just to test this decorator login_required.

First, create the view in core/views.py (do not add the decorator yer):

```
def test_view(request):
        return render(request, "core/test.html")
```

Add a simple template core/templates/core/test.html:

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

{% block content %}
   Hey!!! This is a test!
{% endblock content %}
```

Test it: http://localhost:8000/test

Then, add the decorator login_required:

```
@login_required
def test_view(request):
        return render(request, "core/test.html")
```

Let's reload the test page: this time the app redirects us to the login page, because this page is asking to be logged in. Log in to test that you can access the test page. Notice that in the login url, the app adds a next with the test url part. Once you are logged in, django will redirect you to the test page thanks to this next parameter.

Class based views login required:

Let's replicate this for the class based views. For class based view, obviously we are going to use a class. We will use django.contrib.auth.mixins import LoginRequiredMixin, so do not forget to add that line to your import section at the beginning of the file.

Let's add this to the MovieList view.

class MovieListView(LoginRequiredMixin, ListView):

Then try to access the page without being logged in.

Let's go to the page to understand how the django developers implemented this class:
https://github.com/django/django/blob/main/django/contrib/auth/mixins.py

Here you can see a diagram of the Listview class:
https://www.brennantymrak.com/articles/listviewdiagram
Dispatch function is one of the first function calls by the class.

Le's see the code for ListView:
https://github.com/django/django/blob/main/django/views/generic/list.py

Which one is the ListView's parent class (where the dispatch function is being called)?

The BaseListView is View, let's see this class:
https://github.com/django/django/blob/main/django/views/generic/base.py

Remember, in the urls, we set them as **.as_view()** in the class, to connect the view with the application.

Let's see the **as_view** function in the view class, after some checkings and parameters initialization, as_view returns self.dispatch(request, *args, **kwargs)
And the class LoginRequiredMixin, overrides the dispatch function, checking if the user is logged in, if yes, it calls the parent dispatch function (with the super()), if not it is going to redirect to the login page.